百问网
www.100ask.net

ARM Cortex-A7

Embedded Development Platform

User Manual

Rev. 5.1

2022/09/13

手册属性

类别	嵌入式开发
文档名	嵌入式 Linux 应用开发完全手册_韦东山全系列视频文档全集
当前版本	5.1
适用型号	100ASK_IMX6ULL_Pro
编辑	百问科技文档编辑团队
审核	韦东山

更新记录

更新日期	更新内容	更新版本	审核
2022/08/12	前三篇格式整理完毕	V5.0	
2022/08/22	修正封面、2-2-2.5	v5.0	
2022/09/02	修正 Windows 下烧写镜像到 SD 卡	V5.1	
2022/09/13	正式发布 V5.1	V5.1	
2023/01/29	修正部分命令的错误	V5.1	

目录

手册属性 1
更新记录 1
目录 1
>>>>>>>>第一篇 学习路线、教程介绍、资料下载<<<<<<<< 1
第 1 章 嵌入式 Linux 的学习新路线 1
1.1 嵌入式 Linux 的组成 1
1.2 教程内容及学习顺序 1
1.3 其他专题内容 2
第 2 章 资料下载 3
2.1 有哪些资料 3
2.2 下载开发板配套资料 3
2.3 下载 GIT 仓库的教程配套资料 4
2.4 BSP 8
2.5 在线视频 8
2.6 资料的更新 9
>>>>>>>>第二篇 Ubuntu 基本操作<<<<<<<< 10
第 1 章 安装 VMware 运行 Ubuntu 10
1.1 安装 VMware 10
1.2 使用 VMware 打开 Ubuntu 14
第 2 章 Ubuntu 操作入门 19
2.1 Ubuntu 下打开终端 19
2.2 Ubuntu 系统初体验 21
2.3 Linux 常用命令 28
2.4 Ubuntu 下包管理 47
2.5 Ubuntu 的 gedit 编辑器 50
2.6 强烈建议使用纯英文的 ubuntu 52
2.7 默认不能使用 root 用户登录 52
>>>>>>>>第三篇 环境搭建与开发板操作<<<<<<<< 53
第 1 章 配置 VMware 使用双网卡 53
1.1 添加 NAT 网卡 53
1.2 添加桥接网卡 56
1.3 配置桥接网卡 57
1.4 三者互 ping 验证 67
1.5 开发板上网 69
第 2 章 安装软件 70
2.1 安装 Windows 软件 70

2.2 安装 Ubuntu 软件 70
2.3 使用 MobaXterm 远程登录 Ubuntu 71
2.4 使用 FileZilla 在 Windows 和 Ubuntu 之间传文件 73
2.5 编程示例：Ubuntu 上的 Hello 程序 74
2.6 下载 BSP 及配置工具链 75
2.7 使用 Source Insight 阅读 Linux 内核源码 79
第 3 章 IMX6ULL Pro 开发板基本操作 86
3.1 100ask_imx6ull pro 开发板硬件资源简介 86
3.2 100ask_imx6ull pro 开发板软件资源简介 87
3.3 启动方式选择 89
3.4 串口连接 89
3.5 通过串口操作开发板 92
3.6 配置桥接网卡 92
3.7 开发板挂载 Ubuntu 的 NFS 目录 92
3.8 使用 FileZilla 在 Windows 和开发板之间传文件 94
3.9 使用 TFTP 服务传输文件 95
第 4 章 开发板的第 1 个 APP 实验 98
4.1 获取程序 98
4.2 编译程序 99
4.3 上传程序到开发板并运行 99
第 5 章 开发板的第 1 个驱动实验 100
5.1 前提 100
5.2 编译内核 100
5.3 编译安装内核模块 101
5.4 安装内核和模块到开发板上 102
5.5 编译 led 驱动 102
5.6 在开发板安装驱动模块 103
5.7 执行测试程序 104
第 6 章 构建 bootloader、内核、文件系统 106
6.1 准备 106
6.2 解压编译 bootloader 106
6.3 编译 Linux Kernel 和模块 107
6.4 Buildroot 的用法 108
6.5 构建 IMX6ULL Pro 版的根文件系统 115
6.6 开发板使用 NFS 根文件系统 116
第 7 章 烧写整个系统或更新部分系统 119
7.1 怎么更新整个系统或某些部件 119
7.2 100ASK_IMX6ULL 启动开关 120

7.3 在 Windows 使用 USB 烧写工具 120
7.4 在 Ubuntu 下使用 USB 来烧写 127
7.5 更新根文件系统 129
7.6 使用板子的 Linux 系统来烧写 129
7.7 在 Windows 下烧写 SD/TF 卡 131
7.8 在 Ubuntu 下烧写 SD/TF 卡 136
7.9 老方法：使用 mfgtool 烧写整个系统到 EMMC 上 139
>>>>>>>>第四篇 嵌入式 Linux 应用开发基础知识<<<<<<<< 141
第 1 章 HelloWorld 背后没那么简单 141
1.1 交叉编译 hello.c 141
1.2 请回答这几个问题 142
第 2 章 GCC 编译器的使用 143
2.1 GCC 编译过程(精简版) 143
2.2 GCC 编译过程 145
2.3 GCC 总体选项(Overall Option) 146
2.4 警告选项(Warning Option)-Wall 148
2.5 调试选项(Debugging Option)-g 148
2.6 优化选项(Optimization Option) 148
2.7 链接器选项(Linker Option) 149
2.8 目录选项(Directory Option) 152
2.9 ld/objdump/objcopy 选项 153
第 3 章 Makefile 的使用 154
3.1 配套视频内容大纲 154
3.2 Makefile 规则 159
3.3 Makefile 文件里的赋值方法 160
3.4 Makefile 常用函数 160
第 4 章 文件 IO 166
4.1 文件从哪来？ 166
4.2 怎么访问文件？ 167
4.3 怎么知道这些函数的用法？ 169
4.4 系统调用函数怎么进入内核？ 171
4.5 内核的 sys_open、sys_read 会做什么？ 172
第 5 章 Framebuffer 应用编程 173
5.1 LCD 操作原理 173
5.2 涉及的 API 函数 174
5.3 Framebuffer 程序分析 177
5.4 上机实验 180

第 6 章 文字显示 181
6.1 字符的编码方式 181
6.2 ASCII 字符的点阵显示 187
6.3 中文字符的点阵显示 190
6.4 交叉编译程序：以 freetype 为例 196
6.5 使用 freetype 显示单个文字 199
6.6 使用 freetype 显示一行文字 207
第 7 章 输入系统应用编程 215
7.1 什么是输入系统 215
7.2 输入系统框架及调试 215
7.3 不使用库的应用程序示例 221
7.4 电阻屏和电容屏 227
7.5 tslib 231
第 8 章 网络通信 237
8.1 网络通信概述 237
8.2 网络编程主要函数介绍 240
8.3 TCP 编程 243
8.4 UDP 编程 243
第 9 章 多线程编程 244
9.1 线程的使用 244
9.2 线程的控制 254
9.3 总结 264
第 10 章 I2C 应用编程 267
10.1 I2C 视频介绍 267
10.2 I2C 协议 269
10.3 SMBus 协议 274
10.4 I2C 系统的重要结构体 280
10.5 无需编写驱动程序即可访问 I2C 设备 283
10.6 编写 APP 直接访问 EEPROM 290
第 11 章 Linux 串口应用编程 294
11.1 串口 API 294
11.2 串口收发实验 295
>>>>>>>>第五篇 嵌入式 Linux 驱动开发基础知识<<<<<<<< 300
第 1 章 Hello 驱动(不涉及硬件操作) 300
1.1 APP 打开的文件在内核中如何表示 300
1.2 打开字符设备节点时，内核中也有对应的 struct file 301
1.3 请猜猜怎么编写驱动程序 302

1.4 编写代码 302
1.5 Hello 驱动中的一些补充知识 307
第 2 章 硬件知识_LED 原理图 314
2.1 先来讲讲怎么看原理图 314
第 3 章 普适的 GPIO 引脚操作方法 316
3.1 GPIO 模块一般结构 316
3.2 GPIO 寄存器操作 316
第 4 章 具体单板的 GPIO 操作方法 317
4.1 IMX6ULL 的 GPIO 操作方法 317
第 5 章 最简单的 LED 驱动程序 326
5.1 LED 操作方法_基于 IMX6ULL 326
5.2 最简单的 LED 驱动程序编程_基于 IMX6ULL 326
第 6 章 LED 驱动程序框架 328
6.1 回顾字符设备驱动程序框架 328
6.2 对于 LED 驱动，我们想要什么样的接口？ 328
6.3 LED 驱动能支持多个板子的基础：分层思想 329
6.4 写代码 330
6.5 课后作业 333
第 7 章 具体单板的 LED 驱动程序 334
7.1 怎么写 LED 驱动程序？ 334
7.2 百问网 IMX6ULL 的 LED 驱动程序 335
第 8 章 驱动设计的思想 342
8.1 面向对象 342
8.2 分层 342
8.3 分离 342
8.4 写示例代码 343
8.5 课后作业 344
第 9 章 驱动进化之路：总线设备驱动模型 345
9.1 驱动编写的 3 种方法 345
9.2 在 Linux 中实现“分离”：Bus/Dev/Drv 模型 346
9.3 匹配规则 347
9.4 常用函数 347
9.5 怎么写程序 348
9.6 课后作业 348
第 10 章 LED 模板驱动程序的改造：总线设备驱动模型 349
10.1 原来的框架 349
10.2 要实现的框架 349

10.3 写代码 349
10.4 课后作业 353
第 11 章 驱动进化之路：设备树的引入及简明教程 354
11.1 设备树的引入与作用 354
11.2 设备树的语法 355
11.3 编译、更换设备树 361
11.4 内核对设备树的处理 362
11.5 platform_device 如何与 platform_driver 配对 365
11.6 没有转换为 platform_device 的节点，如何使用 367
11.7 内核里操作设备树的常用函数 367
11.8 怎么修改设备树文件 374
第 12 章 LED 模板驱动程序的改造：设备树 375
12.1 总结 3 种写驱动程序的方法 375
12.2 怎么使用设备树写驱动程序 376
12.3 开始编程 377
12.4 上机实验 378
12.5 调试技巧 378
12.6 课后作业 379
第 13 章 APP 怎么读取按键值 380
13.1 妈妈怎么知道孩子醒了 380
13.2 APP 读取按键的 4 种方法 381
第 14 章 查询方式的按键驱动程序_编写框架 385
14.1 LED 驱动回顾 385
14.2 按键驱动编写思路 385
14.3 编程：先写框架 386
14.4 测试 389
14.5 课后怎业 389
第 15 章 具体单板的按键驱动程序(查询方式) 390
15.1 GPIO 操作回顾 390
15.2 百问网 IMX6ULL 的按键驱动程序(查询方式) 390
第 16 章 GPIO 和 Pinctrl 子系统的使用 397
16.1 Pinctrl 子系统重要概念 397
16.2 GPIO 子系统重要概念 401
16.3 基于 GPIO 子系统的 LED 驱动程序 405
16.4 在 100ASK_IMX6ULL 上机实验 409
第 17 章 异常与中断的概念及处理流程 411
17.1 中断的引入 411

17.2 中断的处理流程 412
17.3 异常向量表 413
17.4 参考资料 413
第 18 章 Linux 系统对中断的处理 414
18.1 进程、线程、中断的核心：栈 414
18.2 18.2 Linux 系统对中断处理的演进 418
18.3 Linux 中断系统中的重要数据结构 425
18.4 在设备树中指定中断_在代码中获得中断 430
18.5 编写使用中断的按键驱动程序 434
18.6 IMX6ULL 设备树修改及上机实验 436
第 19 章 驱动程序基石 440
19.1 休眠与唤醒 440
19.2 POLL 机制 445
19.3 异步通知 453
19.4 阻塞与非阻塞 462
19.5 定时器 463
19.6 中断下半部 tasklet 467
19.7 工作队列 470
19.8 中断的线程化处理 474
19.9 mmap 478
>>>>>>>>第六篇 实战项目<<<<<<<< 491
>>>>>>>>第七篇 驱动大全<<<<<<<< 492
>>>>>>>>第八篇 其它应用<<<<<<<< 494
第 1 章 Qt 应用开发(仅供测试) 494
1.1 安装 Qtcreator 494
1.2 配置 QtCreator 开发环境 494
1.3 创建 Helloworld 项目 498
1.4 编译 Qt 程序 500
1.5 在开发板运行 Qt 程序 502
第 2 章 开发板板载系统应用、库、工具、使用 503
2.1 图形库和图形应用 503
2.2 压缩与解压缩应用 507
2.3 调试、性能分析、基准测试应用 508
2.4 开发工具应用 523
2.5 文件系统和闪存类应用 529
2.6 硬件处理类工具 533
2.7 开发语言和脚本语言 535
2.8 网络通信类应用 535

2.9 系统工具 540
2.10 库 552
注意事项与售后维修 553
技术支持与开发定制 1
资料获取与后续更新 1
版权声明 1

>>>>>>>>第一篇 学习路线、教程介绍、资料下载<<<<<<<<

具体操作视频链接：https://www.bilibili.com/video/BV12A411J7DG

第1章 嵌入式 Linux 的学习新路线

1.1 嵌入式 Linux 的组成

嵌入式 Linux 系统，就相当于一套完整的 PC 软件系统。

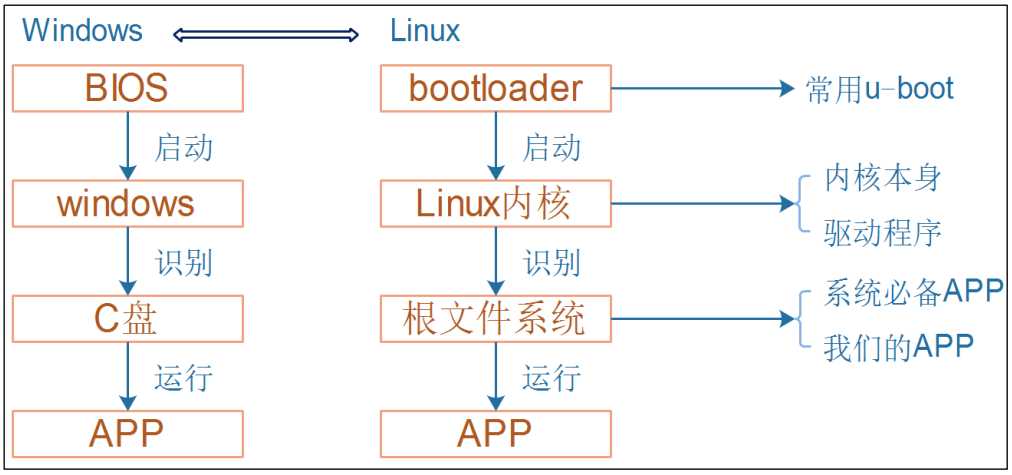


图 1.1 嵌入式 Linux 和 Windows 的差别

很多人喜欢从系统启动流程开始学习：先学习裸机，裸机集合起来就是 u-boot，再学习内核移植、驱动开发，接下来学习根文件系统，最后学习 APP 开发。

学习裸机需要 2、3 个月，学习 u-boot 也需要 2、3 个月，结果工作中 u-boot 基本不用改，并且 u-boot 比驱动开发还难！

按这套流程下来，学了后面忘了前面，最惨的是：不能快速上手工作，消耗学习热情！

入门讲究的是快速，入门之后再慢慢深入，

特别是对于急着找工作的学生，对于业余时间挑灯夜读的在职工程师，一定要快！

1.2 教程内容及学习顺序

本教程目前有如图 1.2 所示内容：

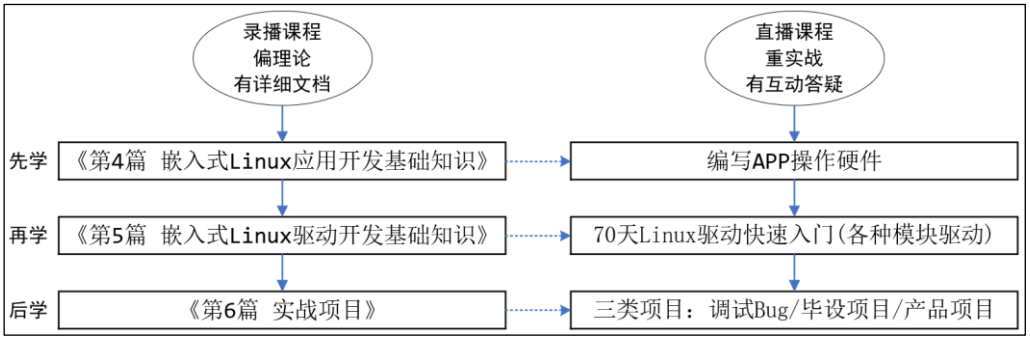


图 1.2 教程视频

在官网中，位置如下：

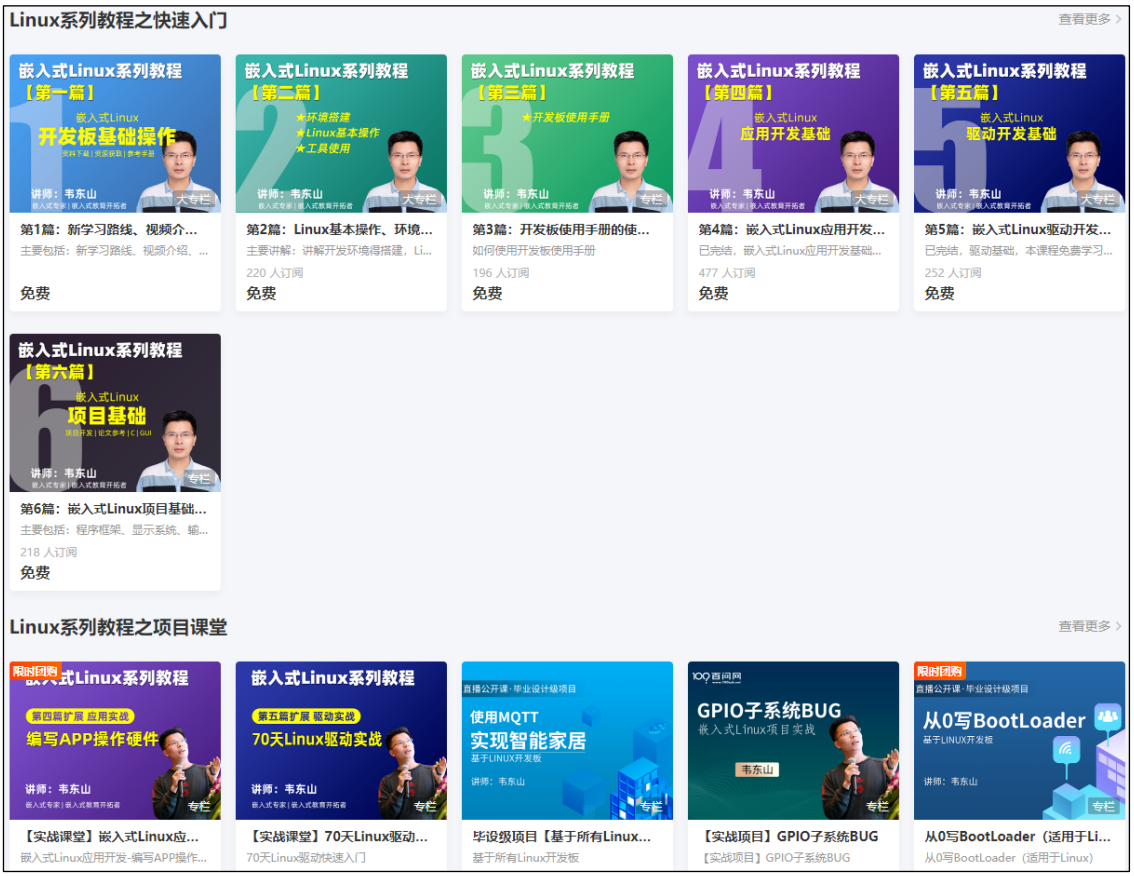


图 1.3 官网中的教程视频

对于初学者，下载好资料、搭建好开发环境后，按照上图顺序学习即可： 先应用，再驱动，最后做项目。应用、驱动、项目这三类课程，都有录播课程(篇理论)、直播课程(重实战)，先学录播课程，再学习直播课程。

1.3 其他专题内容

入门之后，如果你想学习某一个专题，我们会陆续推出各类专题视频。目前正在录制《驱动大全》，深入讲解各类驱动子系统。后续规划的专题有：u-boot、buildroot、调试。

第2章 资料下载

2.1 有哪些资料

所有资料分 4 类：

① 开发板配套资料(原理图、虚拟机、烧写工具等)，放在百度网盘
② 录制视频过程中编写的文档、源码、图片，放在 GIT 仓库
③ u-boot、linux 内核、buildroot 等比较大的源码，放在 GIT 仓库
④ 视频，在线观看，放在百问网、B 站等网站

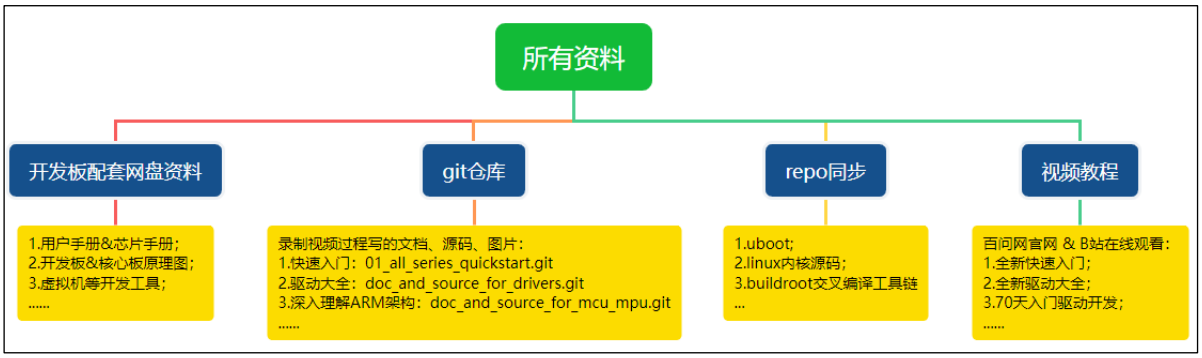


图 2.1 资料体系

2.2 下载开发板配套资料

打开官网：https://www.100ask.net/，在首页点击“资料下载” 跳转到下载中心。

在下载中心左侧选择开发板，点击后可以看到这一项，建议使用百度网盘下载，如图 2.2 所示

图 2.2 资料下载页

2.3 下载 GIT 仓库的教程配套资料

教程配套的资料放在 GIT 仓库里，随着视频的不断更新，使用 GIT 可以很方面地更新配套的资料：

https://e.coding.net/weidongshan/01_all_series_quickstart.git

注意，上述地址无法直接打开，必须使用 GIT 命令来下载。

下载成功后，内容如下图 2.3 所示：

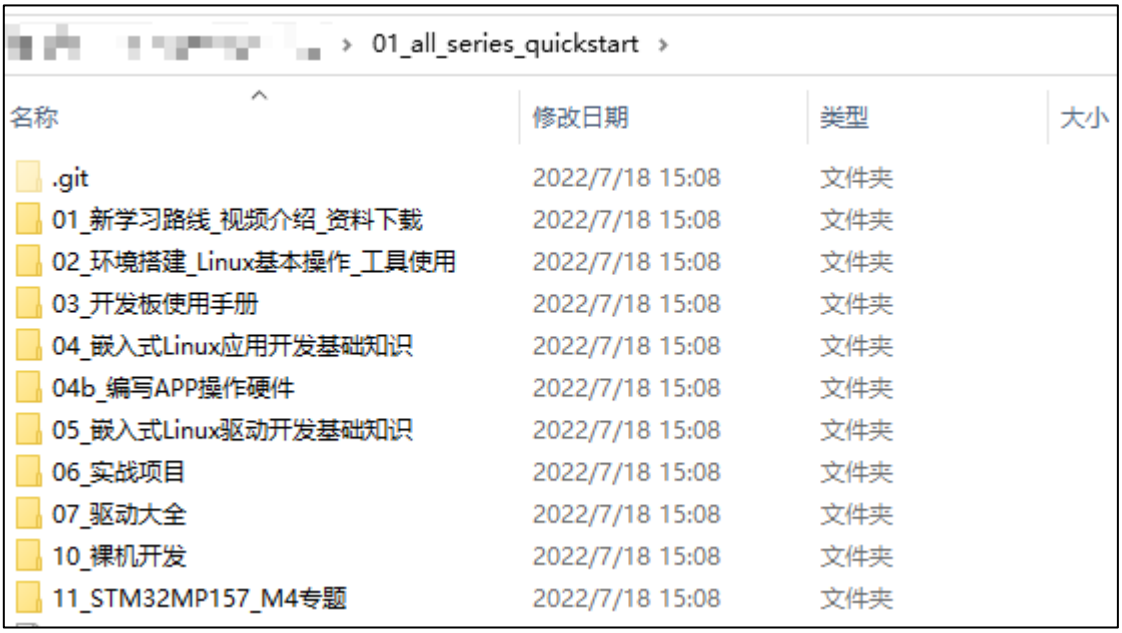


图 2.3 资料 Git 仓库目录结构

2.3.1 安装 GIT

在 Windows 下，GIT 名为 msysGit，从 https://gitforwindows.org/ 上下载安装文件，双击安装即可。

在 Ubuntu 下，执行以下命令即可，它会从网上下载安装 git（在我们发布的 Ubuntu 虚拟机里，已经安装有 git，无需再次安装）：

```
sudo apt-get install git
```

无论是 Windows 还是 Linux，GIT 命令的用法是相似的，对于 Windows，进入 Git 命令行的方法是在“开始”->“所有程序”->“Git”下启动 Git Bash。

Git Bash 中命令的用法跟 Linux 完全一样，比如也可以执行 cd、ls 等命令。

2.3.2 GIT 常用命令

表 2-1 Git 常用命令表

GIT 命令	说明	示例
clone	克隆，从远程下载仓库	git clone https://e.coding.net/weidongshan/01_all_series_quickstart.git
pull	拉取，从远程更新仓库	git pull origin

log	查看本地仓库的记录	git log，快捷键：f 前翻、b 后翻、q 退出
status	查看本地仓库状态， 比如有无修改， 修改有无提交进仓库里	git status
tag	查看标签， 或是打标签	git tag // 查看标签 git tag v2 // 打标签
checkout	提取出某个版本	使用 git log 查看版本，可以看到这样的版本号： commit 4eb78f0a27a85957e1d38a23c5b031cc2aa4b93f 这时就可以执行以下命令取出这个版本： git checkout 4eb78f0a27a85957e1d38a23c5b031cc2aa4b93f 执行上述命令后，当前目录里就是这个版本的源码； 要想提取出最新的代码，执行： git checkout master

如果只是使用 GIT 来下载资料，看后面的示例就可以了。如果要深入学习 GIT，用 GIT 来管理你的代码、协同开发，这有一个图形化介绍 GIT 的网站：

https://learngitbranching.js.org/?demo=&locale=zh_CN

2.3.3 下载和更新 GIT 仓库

要获取视频对应的资料，可以执行以下命令，这称为“克隆”，这会得到一个名为 01_all_series_quickstart 的目录：

```
git clone https://e.coding.net/weidongshan/01_all_series_quickstart.git
```

如果在你“克隆”之后，我们又更新了源码，你可以启动 git bash 后，使用 cd 命令进入 GIT 仓库目录，执行更新命令。

假设要进入 D:\abc\01_all_series_quickstart 目录，可以执行以下命令：

```
cd /D
cd abc
cd 01_all_series_quickstart
```

也可以执行一个命令直接进入该目录，注意目录分隔符是“/”而非“\”。

```
cd  /D/abc/01_all_series_quickstart
```

在 01_all_series_quickstart 目录下，执行以下命令更新 GIT 仓库：

```
git pull origin
```

2.3.4 日常使用示例

注意：建议下载源码后，复制到其他目录去修改；否则以后更新时可能会和你的本地修改产生冲突。

第 1 天：下载仓库

假设你要把仓库下载到 D 盘 abc 目录，如图 2.4 操作：

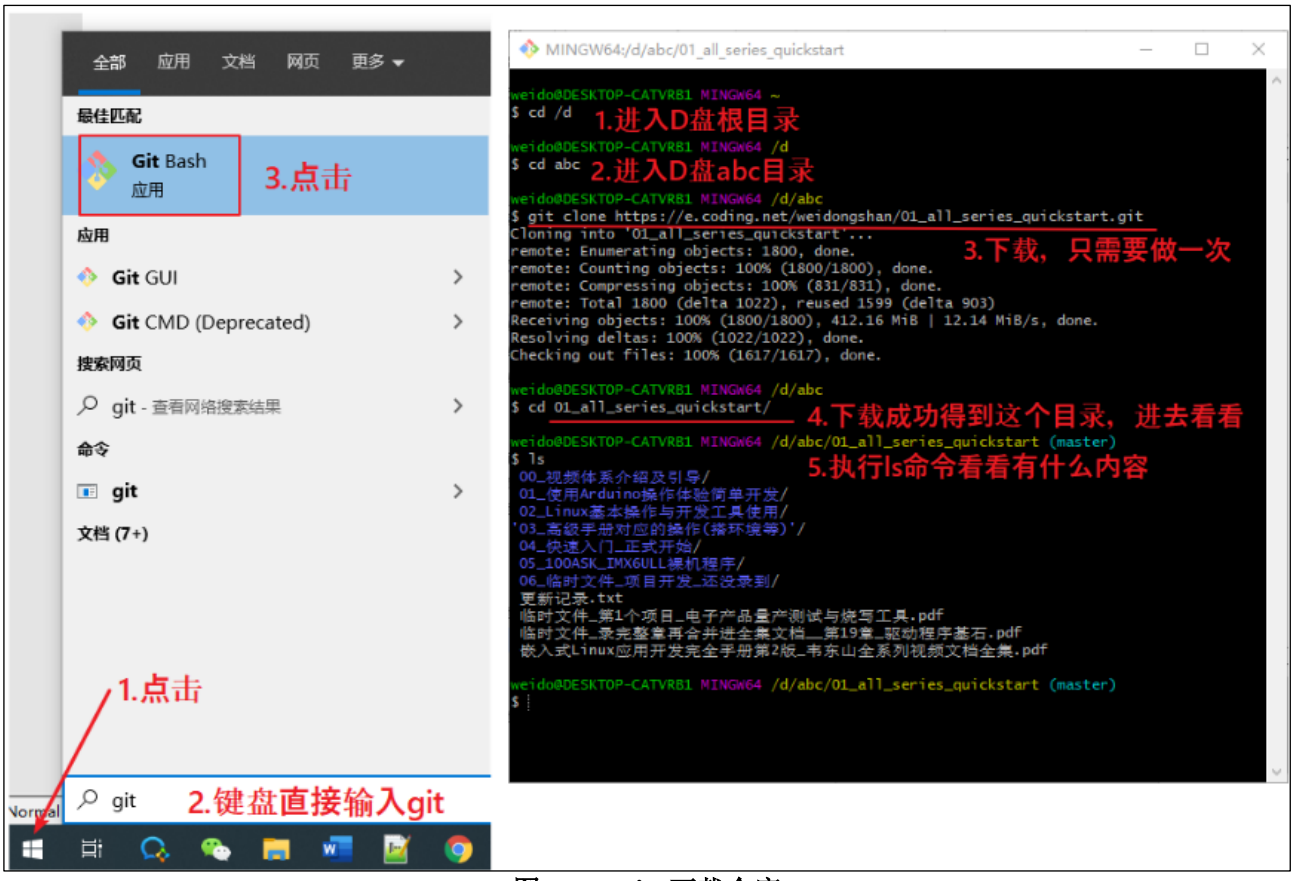


图 2.4 Git 下载仓库

① 进入 D 盘根目录:`cd /d`

② 进入 D 盘 abc 目录:`cd abc`

③ 下载（只需做一次）:

```
git clone https://e.coding.net/weidongshan/01_all_series_quickstart.git
```

④ 下载成功得到这个目录，进入这个目录看看:

```
cd 01_all_series_quickstart
```

⑤ 使用“ls”命令查看内容:

```
ls
```

第 2 天：查看有无更新，如图 2.5 所示：

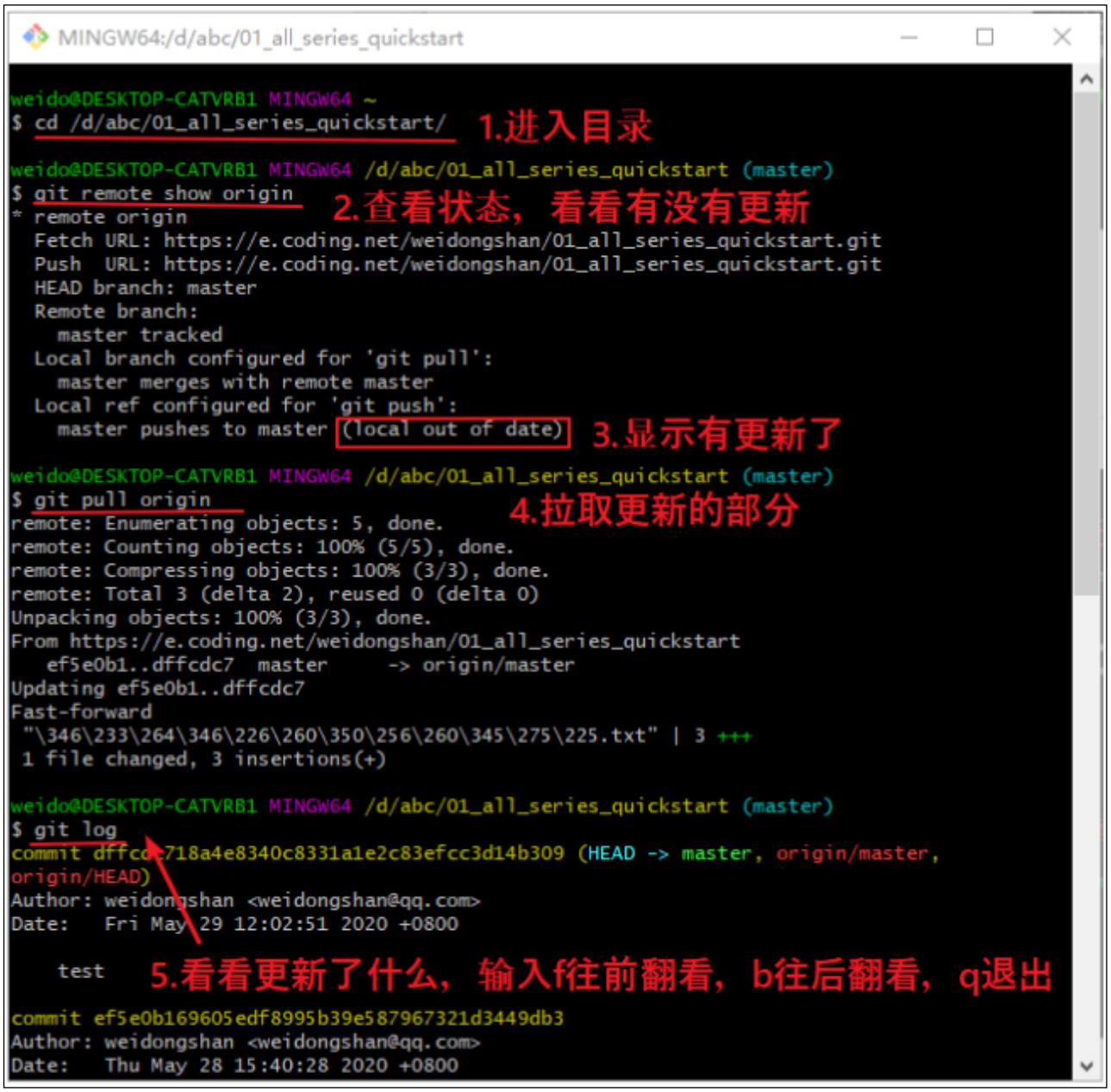


图 2.5 Git 仓库无更新

① 进入之前下载的 git 本地仓库目录：

```
cd /d/abc/01_all_series_quickstart
```

② 查看有无更新：

```
git remote show origin
```

③ 显示有更新：

④ 拉取更新的内容

```
git pull origin master
```

⑤ 查看更新了什么：

```
git log
```

注意：不执行“git remote show origin”查看状态，而是直接执行“git pull origin”也是可以的，后面这个命令会自动检查，有更新它就会下载更新

部分，没有更新也会提示你，如图 2.6 所示：

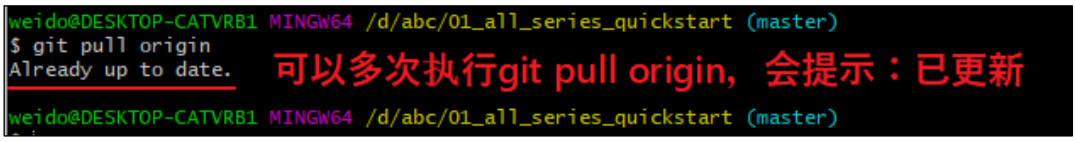


图 2.6 git pull origin

2.4 BSP

BSP，Board Support Package，指板级支持包，是构建嵌入式操作系统所需的引导程序(Bootload)、内核(Kernel)、根文件系统(Rootfs)和工具链(Toolchain)。

每种开发板的 BSP 都不一样，并且这些源码都非常庞大。我们把这些源码都放在 git 仓库里，使用 repo 来管理、下载。

作为初学者，你甚至都还没有安装 Ubuntu、还不会使用 Ubuntu，所以先别去下载它们。

2.5 在线视频

视频可以在线观看，百问网、B 站上都有：

- 百问网(https://www.100ask.net/)
- B 站(https://www.bilibili.com/)：搜“韦东山”。为方便大家使用，这是链接地址：

https://space.bilibili.com/275908810/channel/seriesdetail?sid=1714177

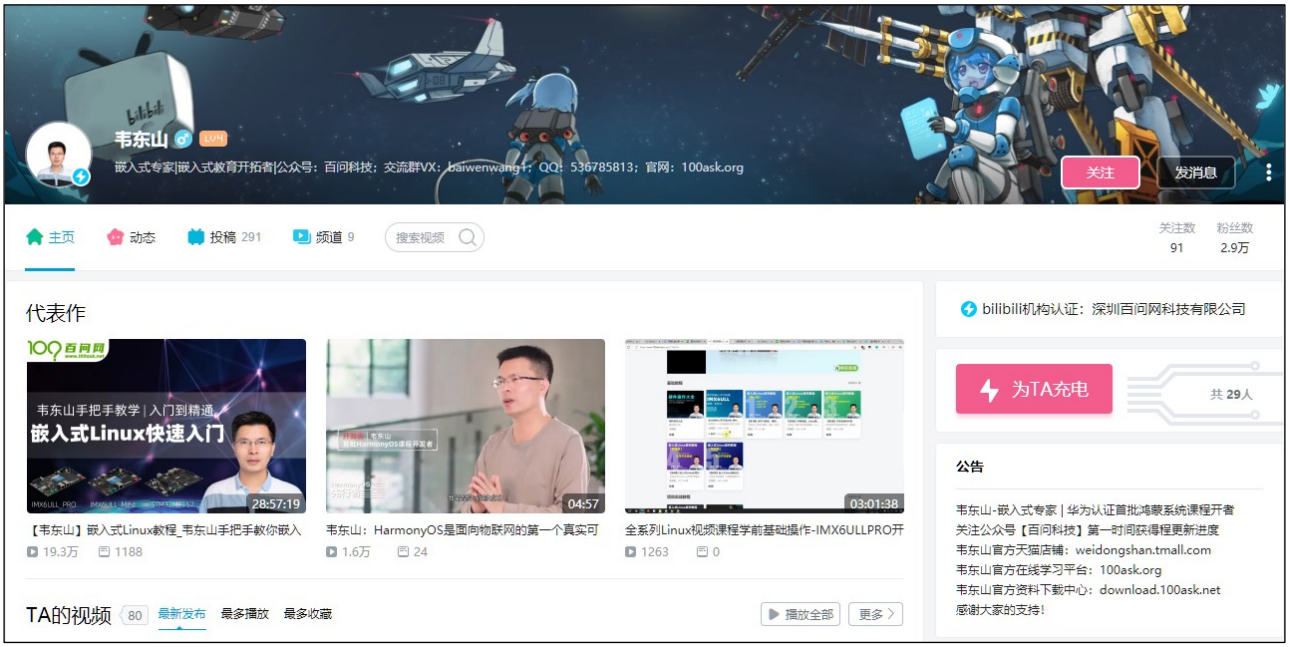


图 2.7 B 站首页

2.6 资料的更新

随着视频的录制，会发布更多的文档、源码，可以使用 GIT 查看更新信息。可以每天使用“git pull”查看有无更新，一般更新了 GIT 就表示视频也有了更新。也可以直接登录百问网(http://www.100ask.net)或是 B 站，查看视频是否更新了。我们也会在公众号发布视频更新信息，发布其他重大消息，可以关注公众号：

>>>>>>>>第二篇 Ubuntu 基本操作<<<<<<<<

具体操作视频链接：https://www.bilibili.com/video/BV19A411J7ci

第1章 安装 VMware 运行 Ubuntu

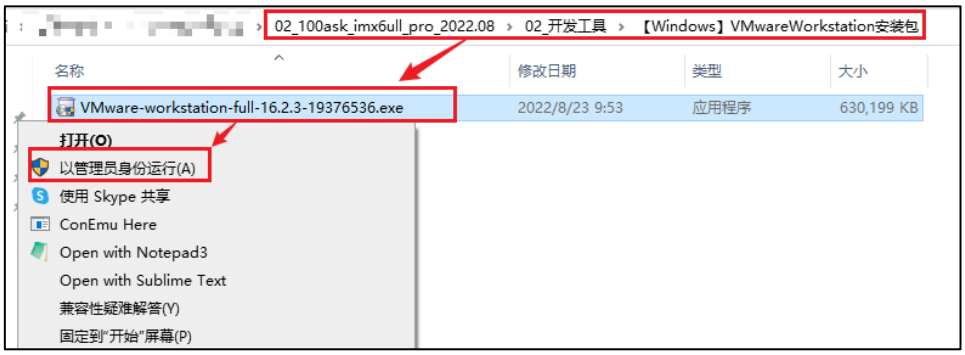
1.1 安装 VMware

Windows 下有很多虚拟机软件，目前市面上流行的有 VMware 和 VirtualBox。VMware 分为收费专业版 Workstation Pro 和非商用免费版 Workstation Player，推荐使用 Workstation Player。

首先从 VMware 官网（www.vmware.com）下载 Workstation Player 安装包，或者使用我们提供的安装包。

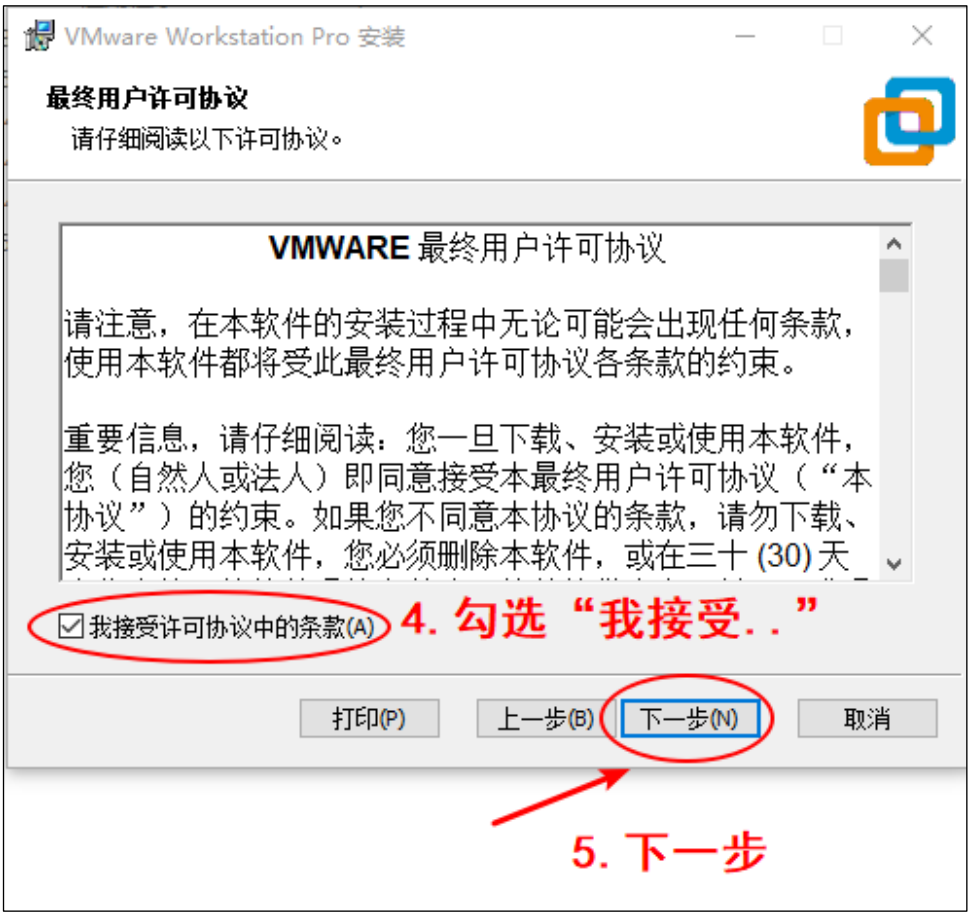
在 <开发板配套资料>02_开发工具\ 【Windows】VMwareWorkstation 安装包中，VMWare 安装软件是：VMware-workstation-full-16.2.3-19376536.exe。下面给出 VMWare 的安装步骤。

第 1 步：以管理员身份运行安装软件

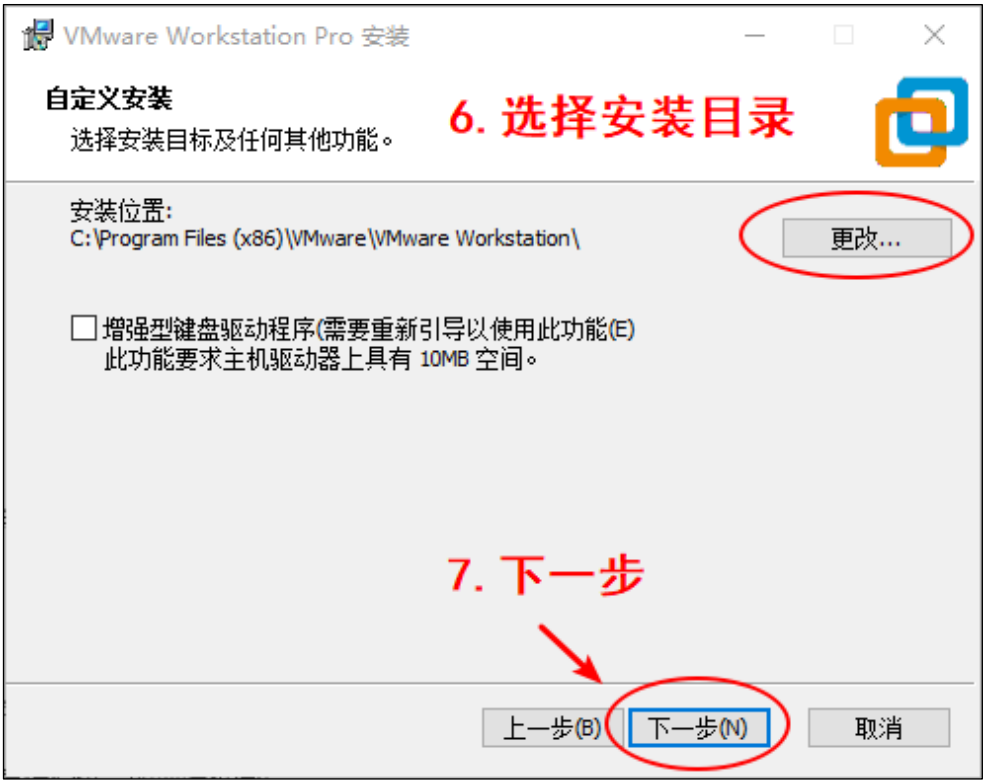


第 2 步：点击“下一步”

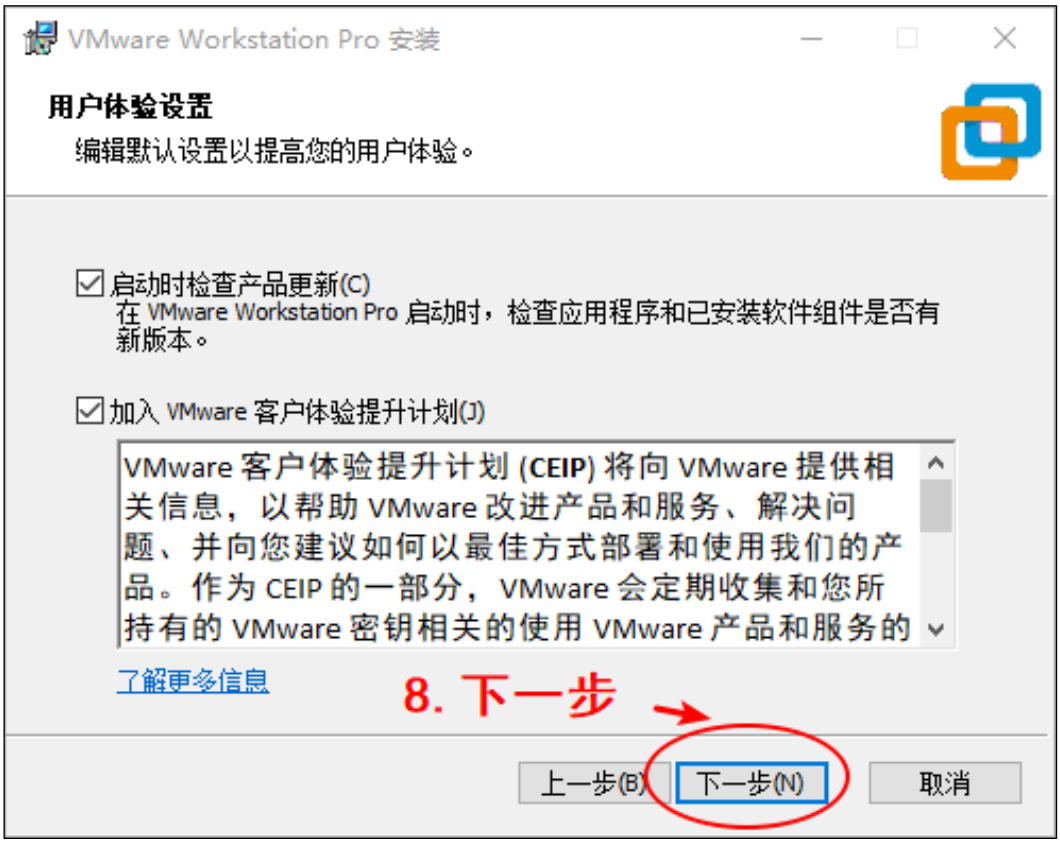
第 3 步：勾选“我接受”点击“下一步”



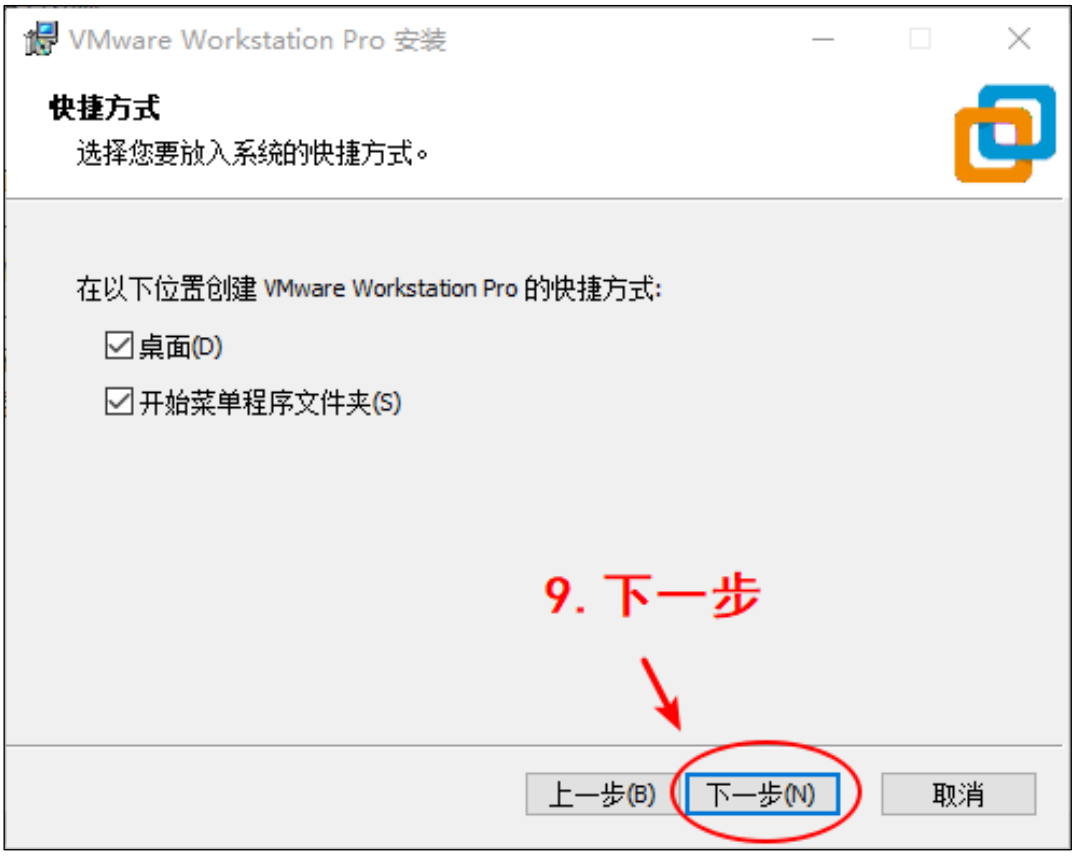
第 4 步：指定安装目录后点击“下一步”



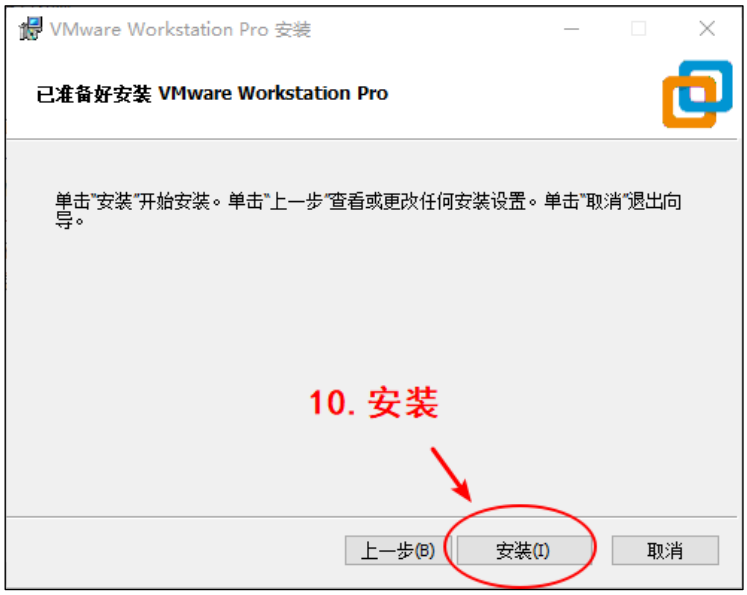
第 5 步：设置用户体验后点击“下一步”



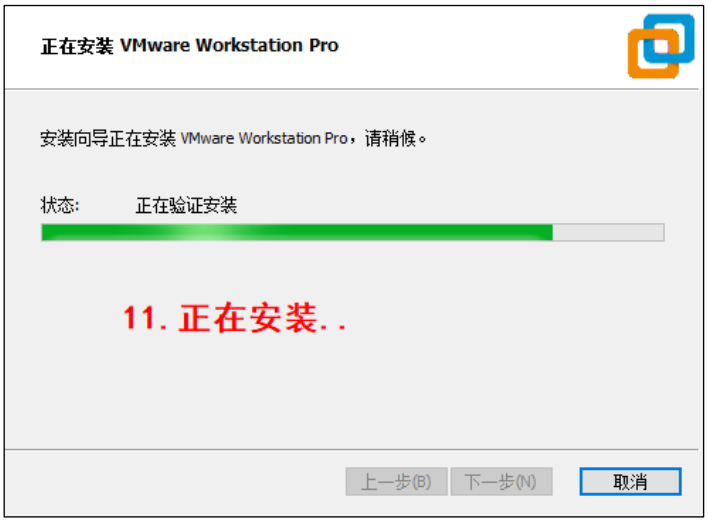
第 6 步：设置快捷方式后点击“下一步”



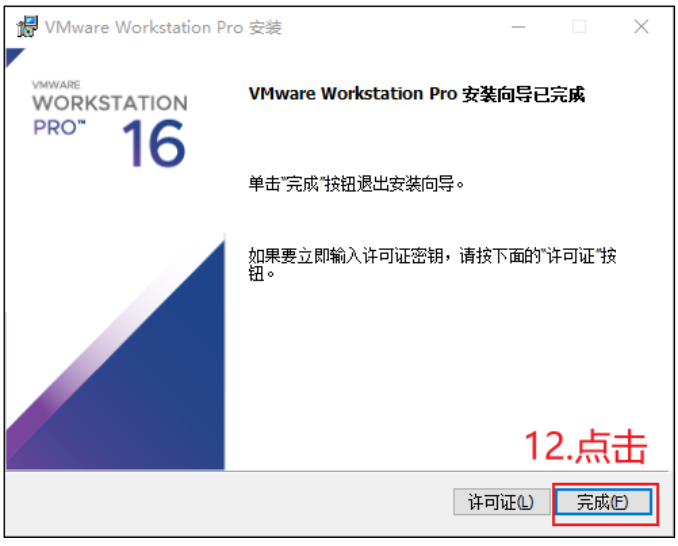
第 7 步：点击“安装”开始安装



第 8 步：等待安装完成



第 9 步：完成安装



VMWare 安装完成后，有两个软件，它们都可以使用，建议使用第 2 个：
① Vmware Workstation Pro：这是收费的，可以试用 30 天。
② Vmware Workstation 16 Player：这是免费的。
注意：本文是在 Windows 10 上安装 VMware。

1.2 使用 VMware 打开 Ubuntu

1.2.1 下载、解压 Ubuntu 映像文件

在百度网盘的“开发板配套资料”中，有 Ubuntu 映像文件，如图 1.1 所示：

图 1.1 Ubuntu 映像文件

在某个磁盘分区里解压文件，这个分区最好有 200G 的空闲空间。解压后，可以得到如图 1.2 所示文件：

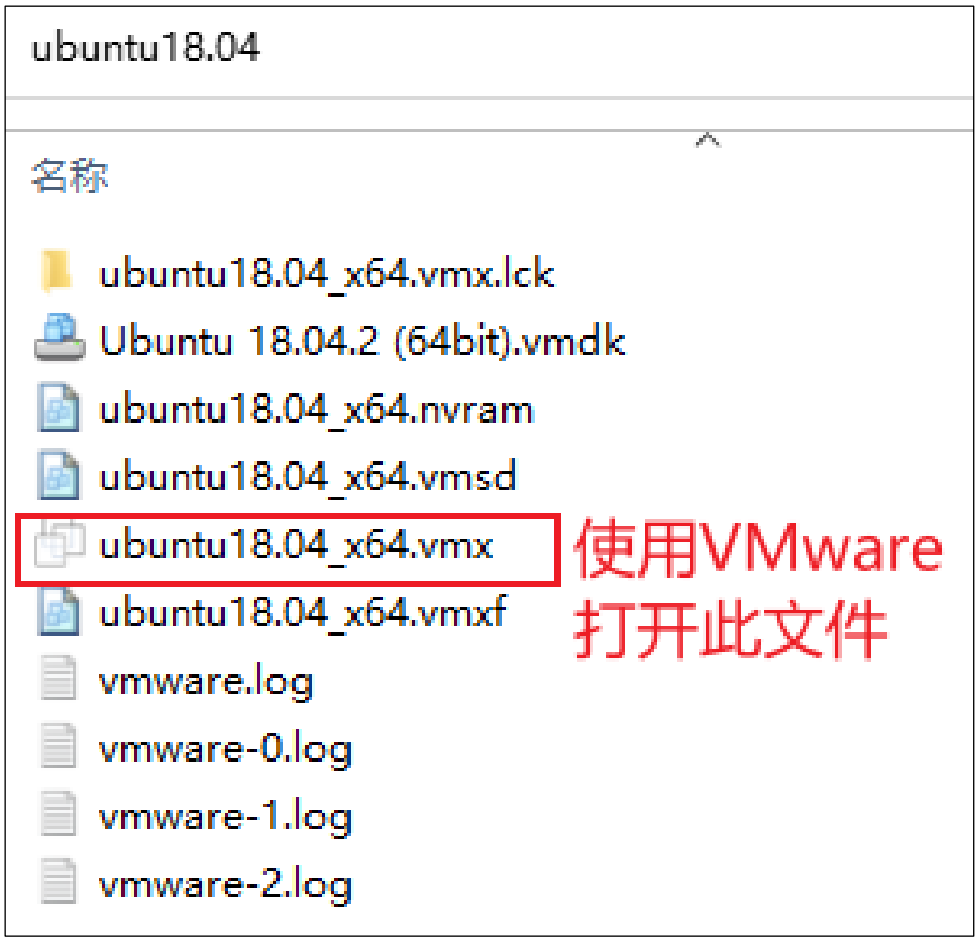


图 1.2 百问网提供的 Ubuntu 文件

1.2.2 在 BIOS 上启动虚拟化(virtualization)

大部分电脑的 BIOS 已经启动了虚拟化，可以打开设备管理器确认这点，如图 1.3 和图 1.4 所示：

图 1.3 Win10 下打开任务管理器

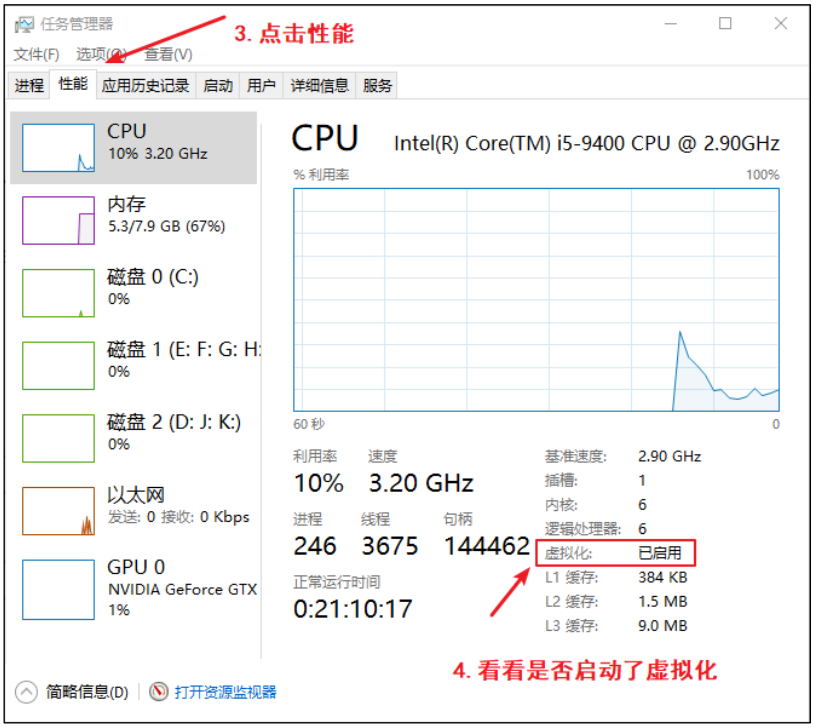


图 1.4 查看虚拟化是否开启

如果上图中虚拟化没有显示为“已启动”，需要重启电脑进入 BIOS 启动虚拟化。各个电脑的 BIOS 设置界面可能不一样，下面的步骤只是示例：

第 1 步：进入 BIOS

开机或重启电脑过程中，在自检画面处反复按 F2 键（注：部分机型使用 Fn+F2）进入 BIOS Setup 设置界面。

第 2 步：找到虚拟化菜单

用键盘的右方向键选中 “Configuration”菜单，然后使用下方向键选中“Intel Virtual Technology”选项并回车，如图 1.5 所示：

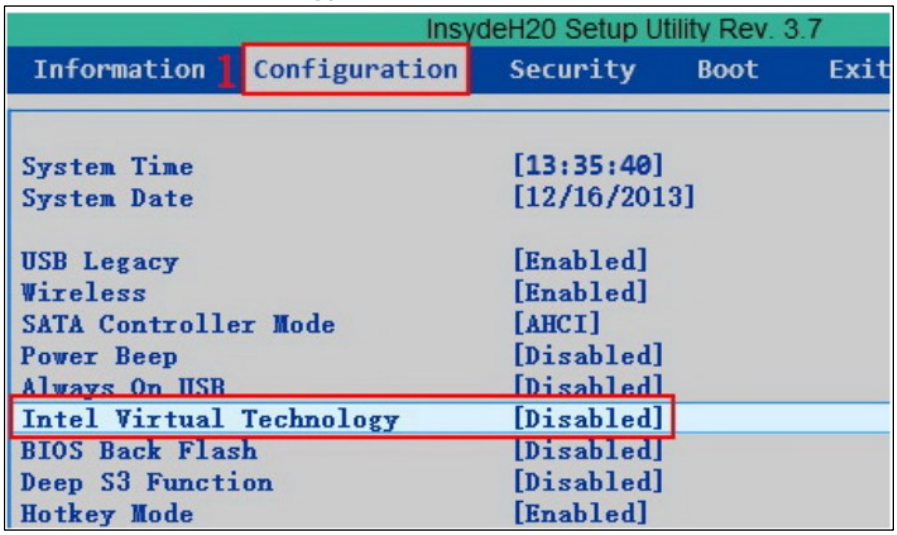


图 1.5 BIOS 找到虚拟化

第 3 步：使能虚拟化

在弹出的菜单中，选择“Enable”并回车，如图 1.6 所示：

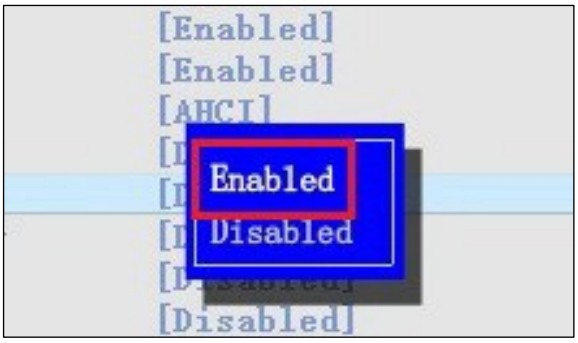


图 1.6 Enable 虚拟化

第 4 步：保存

最后按键盘的 F10 热键（注：部分机型需要配合 Fn+F10）调出保存对话框，选择“Yes”保存退出并自动重启电脑，如图 1.7 所示：

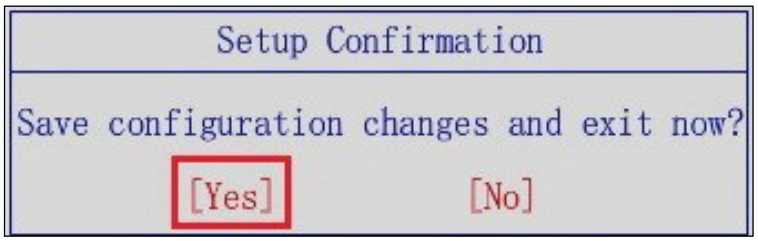


图 1.7 退出 BIOS 且保存

1.2.3 使用 VMware 运行 Ubuntu

第 1 步：以管理员身份打开 Vmware Workstation 16 player，如图 1.8 所示：

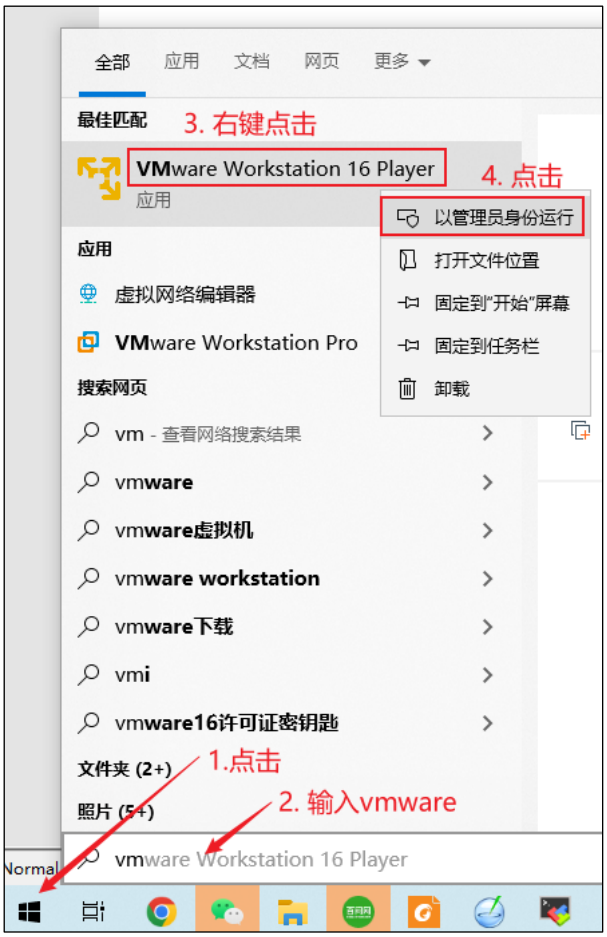


图 1.8 打开 Vmware

第 2 步：打开虚拟机。

使用 vmware 打开前面解压得到的“Ubuntu 18.04_x64.vmx”，如图 1.9 所示：

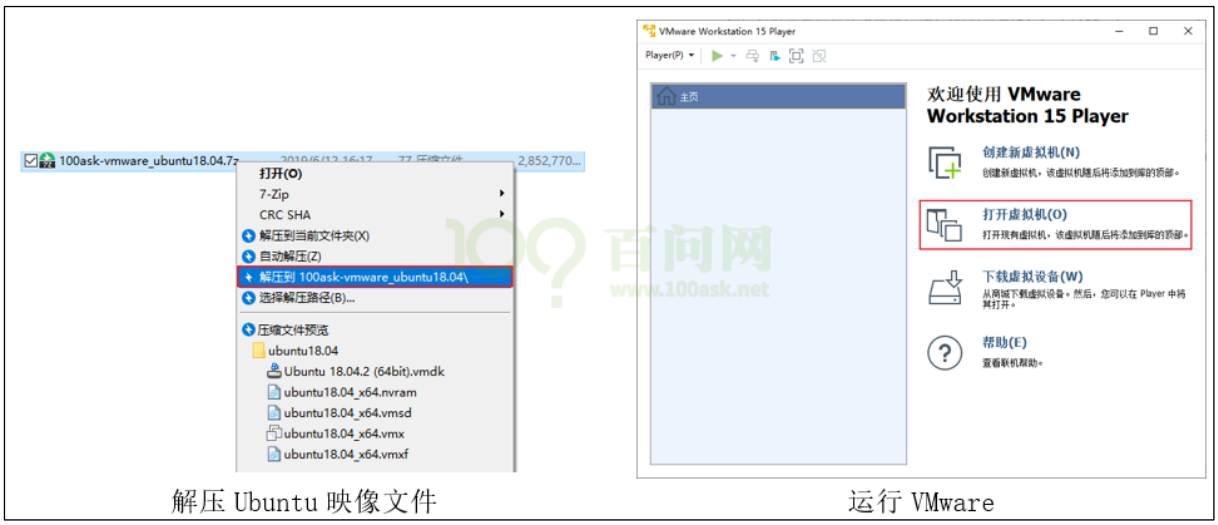


图 1.9 打开虚拟机

第 3 步：播放虚拟机，如图 1.10 所示：

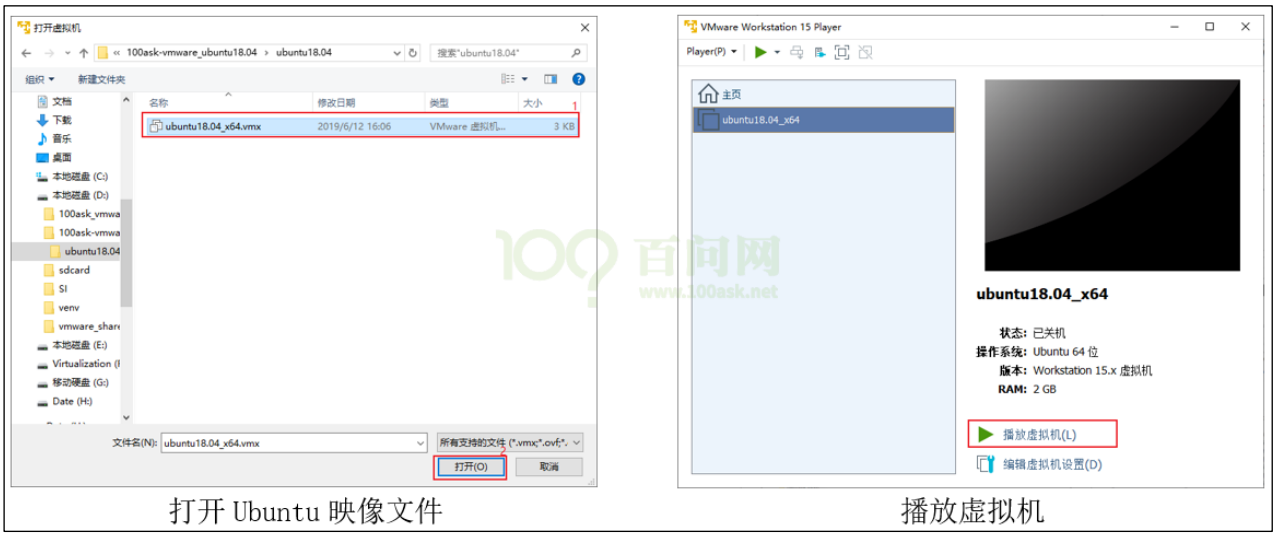


图 1.10 播放虚拟机

第 4 步：第一次启动 Ubuntu 时，选择默认的“我已复制该虚拟机”，启动后输入密码“123456”回车即可登录，如图 1.11 所示：

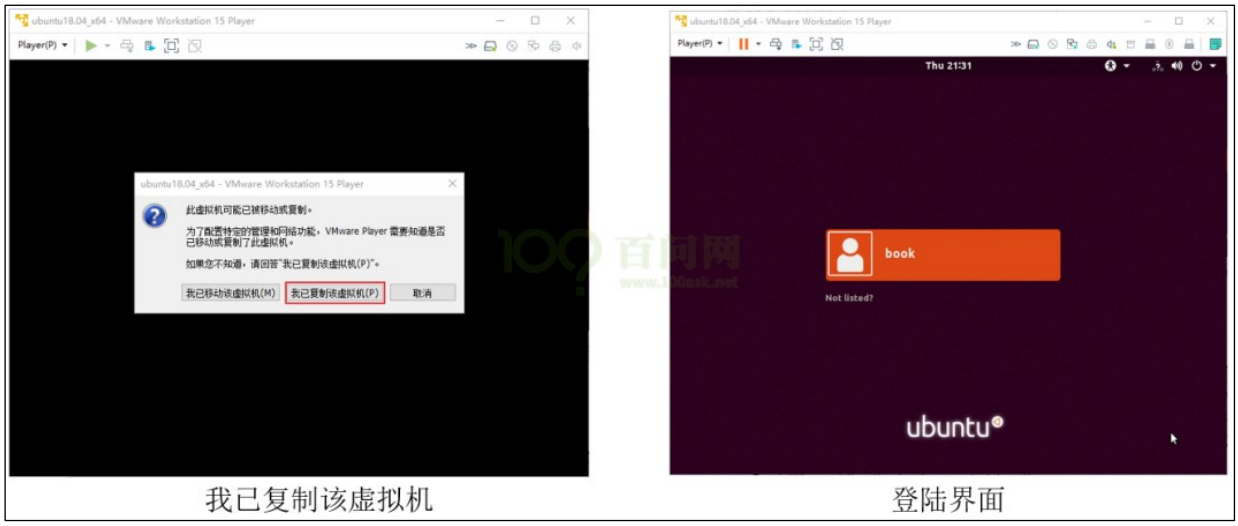


图 1.11 复刻并登录虚拟机

注意：虚拟机默认没有开启小键盘，如果使用小键盘输入，请先开启小键盘。如图 1.12 所示：

图 1.12 注意开启数字小键盘

第2章 Ubuntu 操作入门

2.1 Ubuntu 下打开终端

我们安装的 Ubuntu 是桌面版本，这样我们可以像在 windows 系统下操作一样，相对于平时所说的 Linux 命令行下操作来说，这种体验非常舒适。但是一般我们使用 Linux 都是在命令行下进行操作，所有的操作我们的都可以通过输入命令来完成，绝大多数情况下使用命令行来操作 Linux 系统比通过在 GUI 下操作的效率高很多，虽然说我们使用的 Ubuntu 是包含了 GUI 的 Linux 发行版，然而我们可以像在 windows 下那样唤出 Ubuntu 的终端，打开 Ubuntu 的终端非常简单，以我们使用的 Ubuntu18.04 为例，有种方法可以直接在 Ubuntu 的用户界面下

2.1.1 用搜索框打开终端

我们要输入各种命令，需要先打开终端。

点击 Ubuntu 桌面左上角图标进入搜索框，输入“term”可以弹出终端“Terminal”程序，运行它，如图 2.1 所示：

图 2.1 搜索打开终端

然后就可以在里面执行各种命令了。

2.1.2 使用右键打开终端

在桌面或者在文件浏览器的任何目录下右键鼠标后在弹出的菜单栏中选择“Open in Terminal”，如图 2.2 所示：

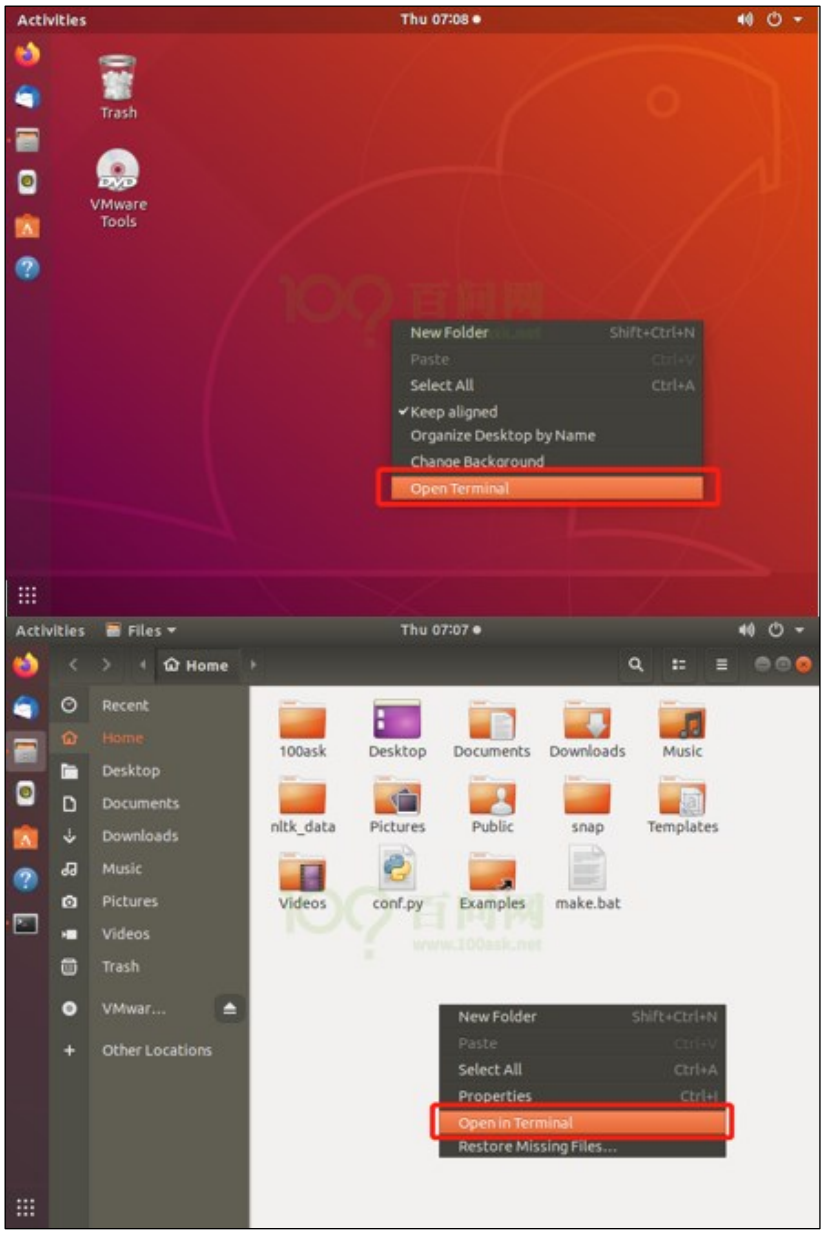


图 2.2 指定位置右键打开终端

2.1.3 快捷键打开终端

这是个比较快捷方便的方法：使用快捷方式打开终端，快捷方式为”Ctrl+Alt+T”，使用快捷方式可在绝大多情况下直接唤出 Ubuntu 的终端（无论你是在浏览器、文件管理器、查看邮件、甚至在一个已经打开的终端下工作，等等都可以直接唤出 Ubuntu 的终端）。

2.2 Ubuntu 系统初体验

2.2.1 Ubuntu 和 Windows 的最大差别：目录

Windows 中每一个分区都对应一个盘符，盘符下可以存放目录与文件，如图 2.3 所示：

图 2.3 Windows 的分区和盘符

注意：目录就是文件夹。

Windows 下某个文件的绝对路径以盘符开始，比如：

```
C:\abc\def\hello.txt
```

这是在 C 盘的 abc 目录下，有 def 子目录；而 def 中有 hello.txt 文件。

Ubuntu 中，以树状结构表示文件夹与文件，没有盘符的概念。比如：

```
/abc/def/hello.txt
```

这表示在根目录下有 abc 子目录，而 abc 下又有 def 目录；def 中有 hello.txt 文件。从名字“/abc/def/hello.txt”中你无法知道 hello.txt 文件位于磁盘哪一个分区。

注意：要想查看某个分区挂载在哪一个目录下，可以执行命令：

```
df -h
```

对于普通用户，在 Ubuntu 下不再关心分区、盘符。需要关心的是哪个目录存什么，如图 2.4 所示：

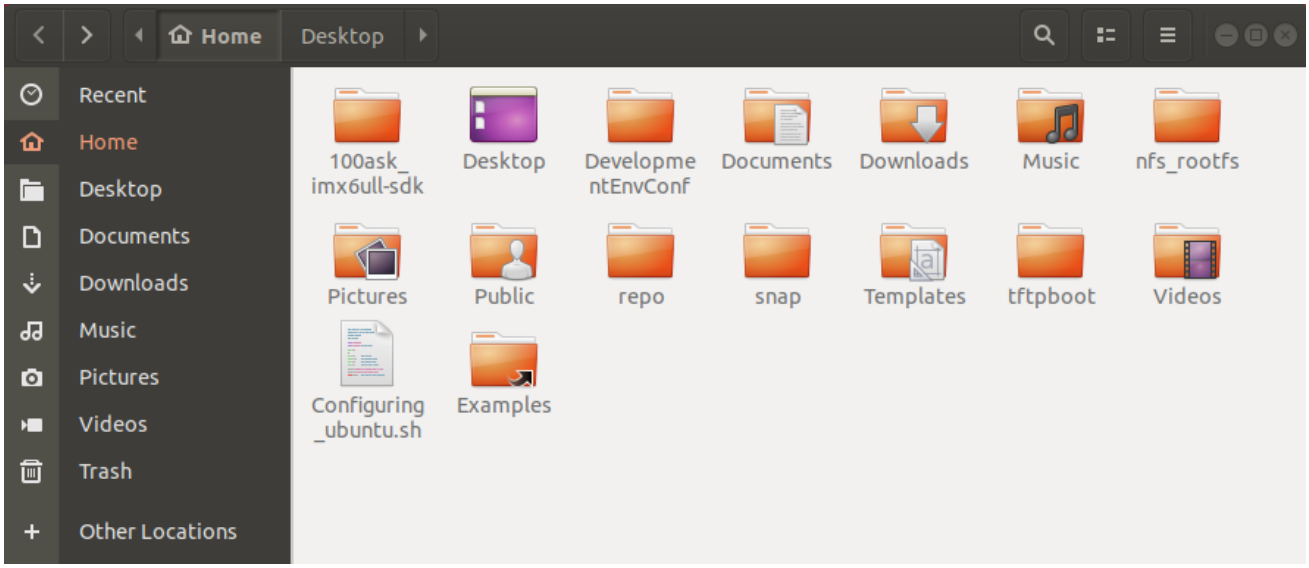


图 2.4 Ubuntu 下的文件目录

Ubuntu 中的目录遵循 FHS 标准(Filesystem Hierarchy Standard，文件系统层次标准)。它定义了文件系统中目录、文件分类存放的原则、定义了系统运行所需的最小文件、目录的集合，并列举了不遵循这些原则的例外情况及其原因。FHS 并不是一个强制的标准，但是大多的 Linux、Unix 发行版本遵循 FHS。

这些目录简单介绍如图 2.5 所示：

```
/
├── bin             所有用户都可以使用的、基本的命令
├── boot            启动文件，比如内核等
├── dev             设备文件，Linux特有的
├── etc             配置文件
├── home            家目录
│   ├── book        用户book的家目录
├── lib             库
├── media           插上U盘等外设时会挂载到该目录下
├── mnt             用来挂载其他文件系统
├── opt             Optional，可选的程序
├── proc            用来挂载虚拟的proc文件系统，可以查看各进程(process)的信息
├── root            root用户的家目录
├── sbin            基本的系统命令，系统管理员才能使用
├── sys             用来挂载虚拟的sys文件系统，可以查看系统信息：比如设备信息
├── tmp             临时目录，存放临时文件
├── usr             Unix Software Resource，存放可分享的与不可变动的的数据
│   ├── bin         绝大部分的用户可使用指令都放在这里(与开机无关)，/bin中的命令跟开机有关
│   ├── games       游戏
│   ├── include     头文件
│   ├── lib         库
│   ├── local       系统管理员在本机自行安装、下载的软件
│   ├── sbin        非系统正常运作所需要的系统命令
│   ├── share       放置共享文件的地方，比如/usr/share/man里存放帮助文件
│   └── src         源码
└── var             主要针对常态性变动的文件，包括缓存(cache)、log文件等
```

图 2.5 FSH 标准

2.2.2 Linux 文件属性

在终端执行“ls -al”命令显示当前目录下的所有文件及文件夹的详细信息。文件属性示意图如图 2.6 所示：

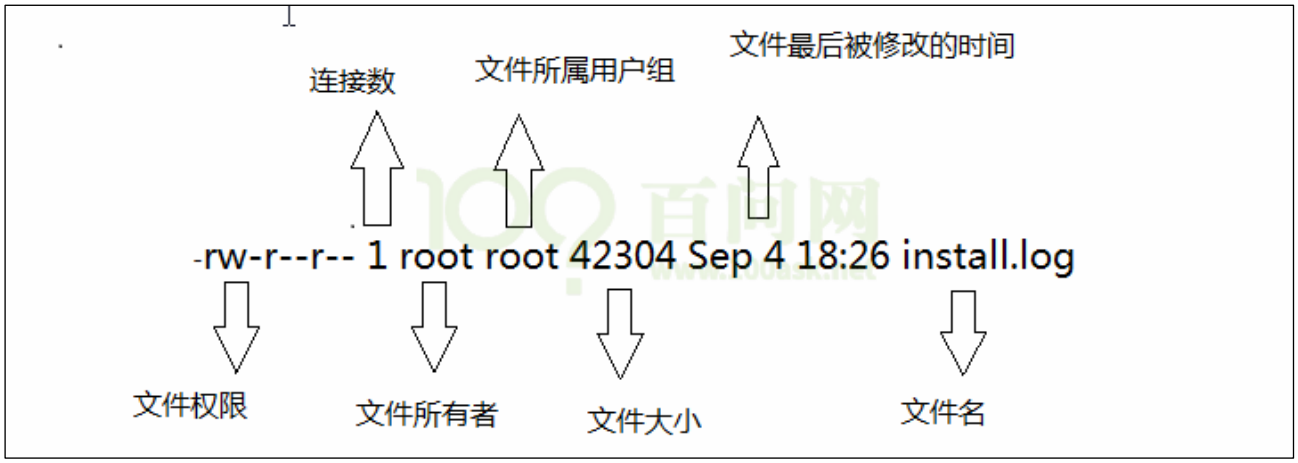


图 2.6 文件属性

第一个字符表示“文件类型”，它是目录、文件或链接文件等。

表 2-1 文件类型

d	目录
-	文件
l	链接文件
b	设备文件里的可供存储的接口设备
c	设备文件里的串行端口设备，如鼠标、键盘等

文件类型后面的 9 个字符以 3 个为一组，第一组表示“文件所有者的权限”；第二组表示“用户组的权限”；第三组表示“其他非本用户组的权限”。每组都是 rwx 的组合，其中 r 代表可读，w 代表可写，x 代表可执行；如果没有对应的权限，就会出现减号（-）。比如“rw-r--r--”表示：文件的所有者对该文件有读权限、写权限，但是没有执行权限；同一个用户组的其他用户对该文件只有读权限；其他用户对该文件也只有读权限。

- 连接数：表示有多少文件名连接到此节点。
- 文件所有者：表示这个文件的“所有者的账号”。
- 文件所属用户组。
- 文件大小：表示这个文件的大小，默认单位是 B(字节)。
- 文件最后被修改的时间：这个文件的创建文件日期或者是最近的修改日期
- 文件名：对应文件的文件名。
- 如果文件名之前多了一个“.”,则说明这个文件为“隐藏文件”，执行“ls -a”命令可以列出隐藏文件。

2.2.3 设置屏幕

在我们的后续学习过程中，很少使用 Ubuntu 的桌面系统，都是远程登录上去的。但是如果 Ubunut 的桌面显示太小、太大，总是让人不舒服。这时可以修改屏幕分辨率，方法如图 2.8 和图 2.9 所示：

图 2.7 点击“Show Applications”找到“Settings”

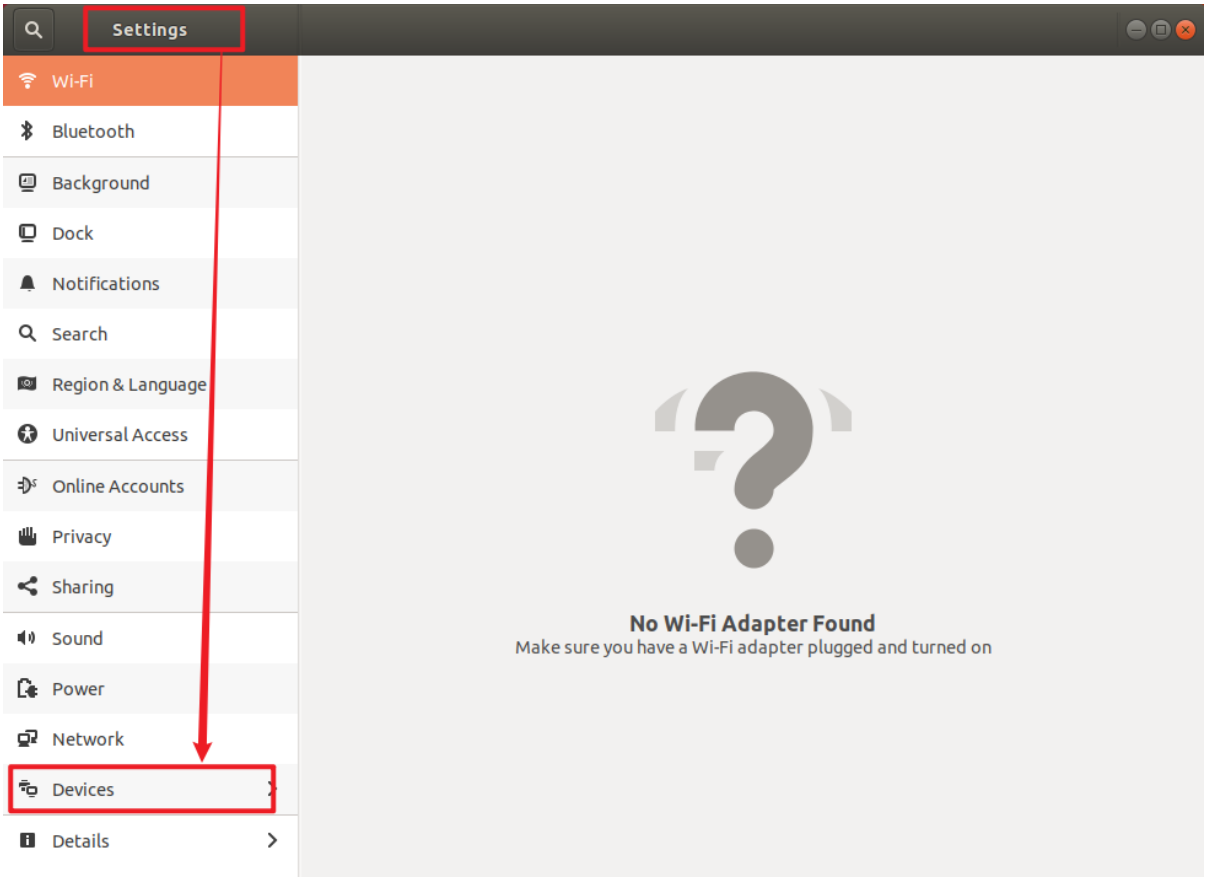


图 2.8 “Settings”中找到“Devices”并点击进入

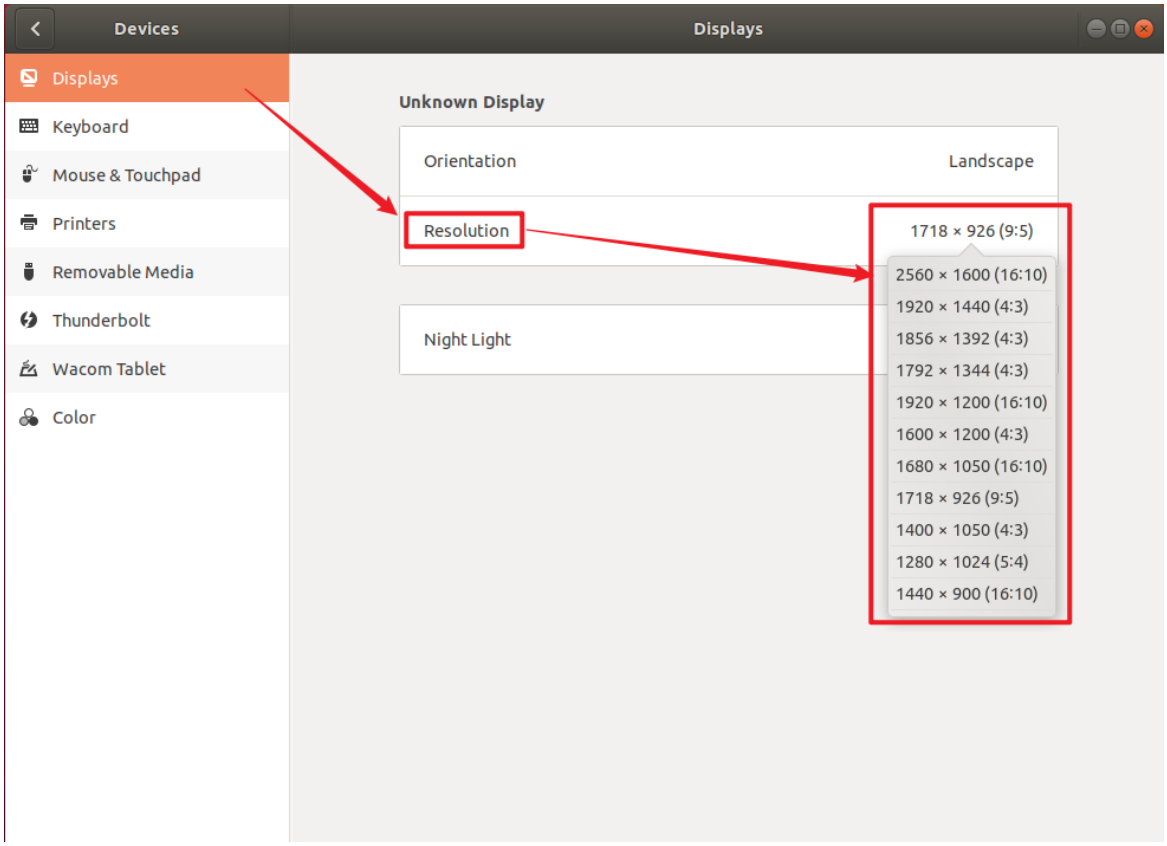


图 2.9 选择合适的分辨率

2.2.4 系统关机与重启

和我们使用 Windows 系统一样，当我们不使用 Ubuntu 系统以后就需要将其关机或者睡眠，千万不要通过直接退出 VMware 软件来关机!!一般的步骤是：

第1步 在虚拟机系统中关闭系统或在 VMware 软件上挂起虚拟机

第2步 关闭 VMware 软件

第3步 windows 系统

Ubuntu 的关机与重启很简单，在主界面，点击右上角的图标，然后选项对应的选项即可，如图 2.10 所示：

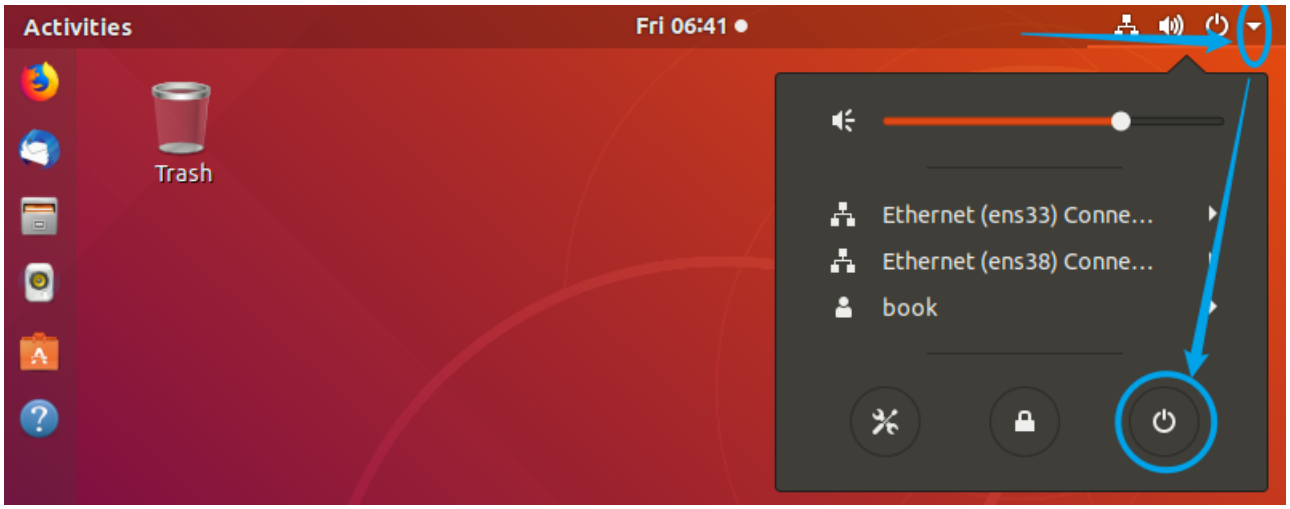


图 2.10 Ubuntu 关机

在弹出的对话框中我们可以进行重启或者关机操作，点击取消按钮可退出此对话框，如图 2.11 所示：

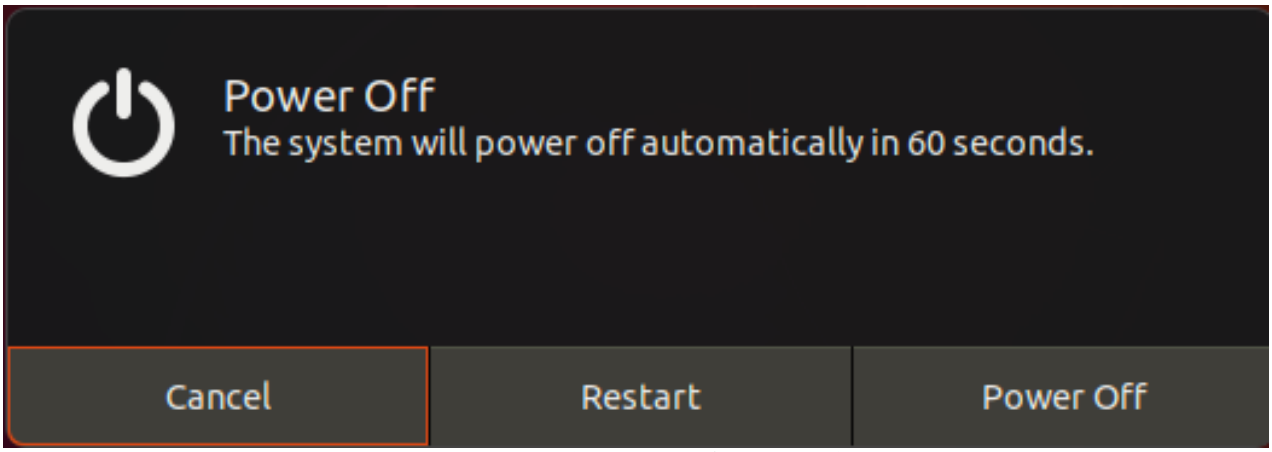


图 2.11 选择重启、关机或者取消

到这里，细心的读者可能会发现，在 windows 系统下我们还有一个选项就是“睡眠”，在 Ubuntu 中没有睡眠选项。其实我们可以通过 VMware 软件来实现虚拟机系统的睡眠操作，那就是挂起操作，将虚拟机系统挂起后，我们下次可以直接将虚拟机恢复到挂起时的状态。将虚拟机挂起非常简单，VMware 导航栏上的电源操作图标，或者在虚拟机的选项卡上右键唤出的菜单的电源选项中也有挂起

操作，如下所示：

在 VMware 导航栏上的电源操作图标进行挂起，如图 2.12 所示：

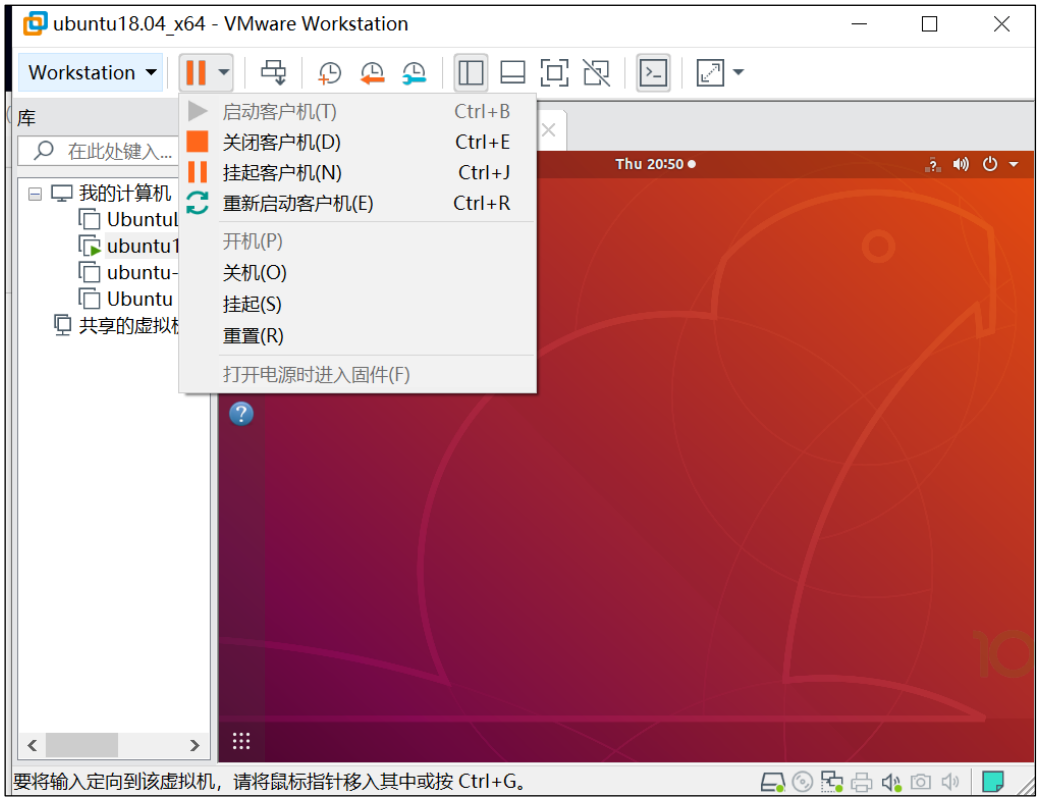


图 2.12 挂起虚拟机

在虚拟机的选项卡上右键唤出的菜单的电源选项中进行挂起，如图 2.13 所示：

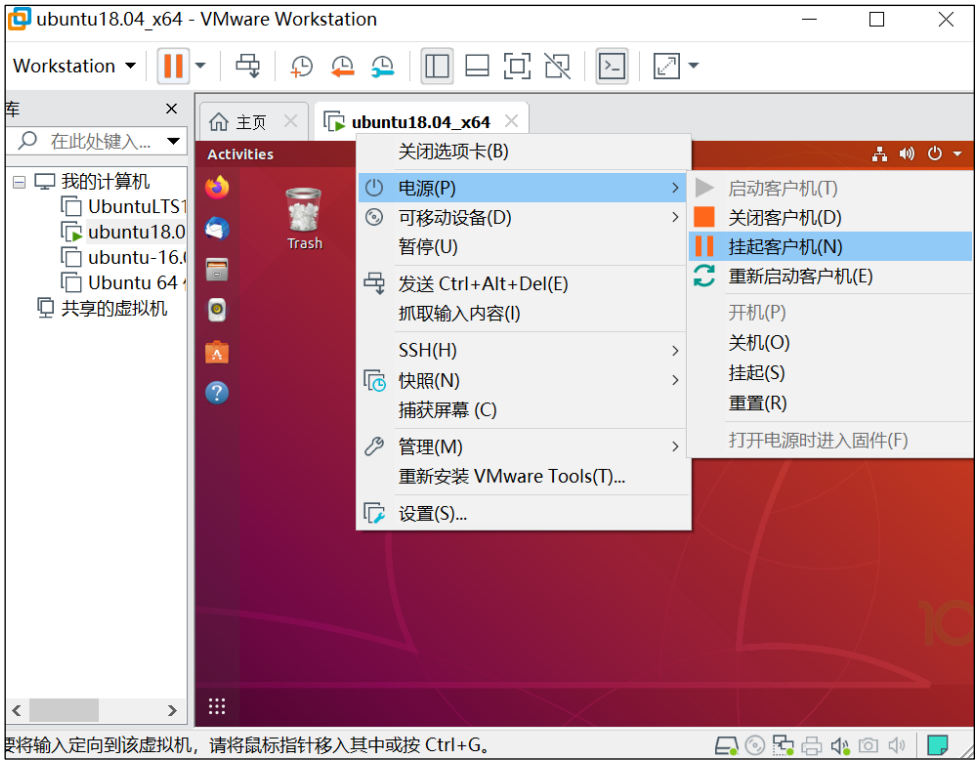


图 2.13 右键虚拟机选择挂起

2.2.5 文件浏览器

每个带有 GUI 的系统都应该有文件浏览器，我们使用的桌面版本的 Ubuntu 也不例外，那么 Ubuntu 的文件浏览器怎么打开呢？

其实要打开 Ubuntu 的文件浏览器非常简单，文件浏览器在 Ubuntu 默认的左侧导航栏中可以直接打开，如图 2.14 所示：

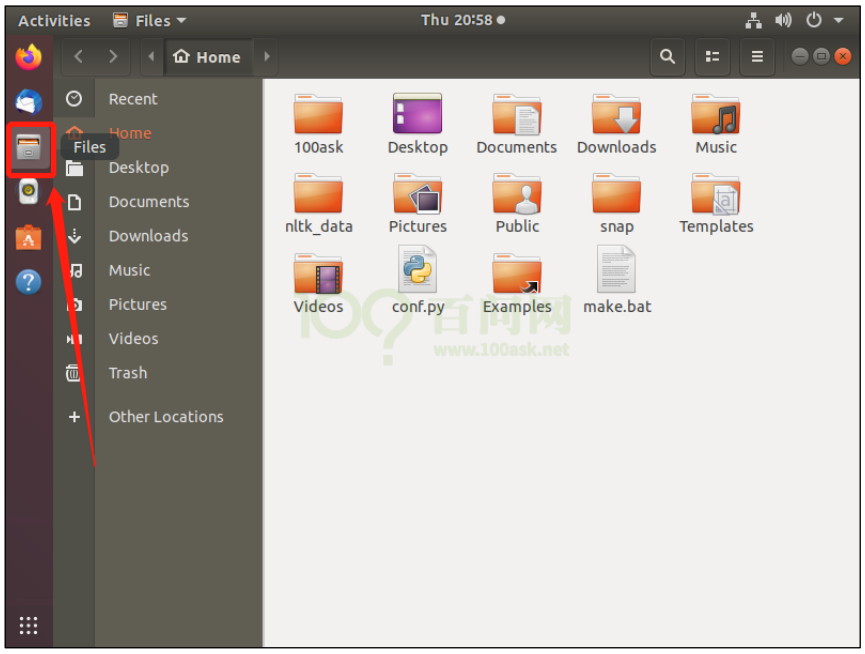


图 2.14 桌面打开文件浏览器

或者我们可以在所有的应用中找到文件浏览器打开，如图 2.15 所示：

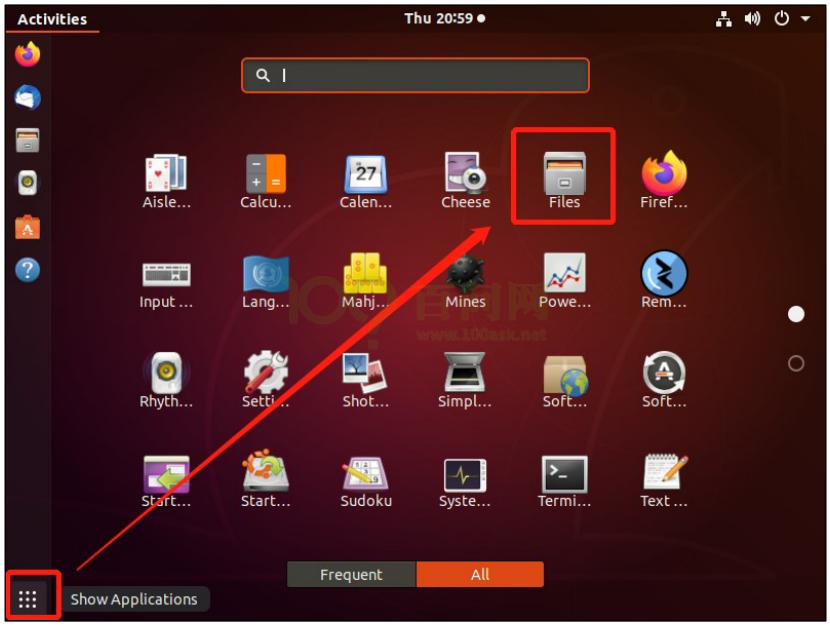


图 2.15 搜索打开文件浏览器

打开文件浏览器之后，我们就可以像在 windows 系统下利用文件资源管理器那样浏览磁盘中所有的文件。

2.3 Linux 常用命令

2.3.1 Linux 命令行介绍

1. Linux Shell 简介

Shell 的意思是“外壳”，在 Linux 中它是一个程序，比如/bin/sh、/bin/bash 等。它负责接收用户的输入，根据用户的输入找到其他程序并运行。比如我们输入“ls”并回车时，shell 程序找到“ls”程序并运行，把结果打印出来。

Shell 有很多种实现，我们常用 bash。

2. Linux 命令的提示符

在 Ubuntu 中打开终端后，即可看到类似下图的提示符：

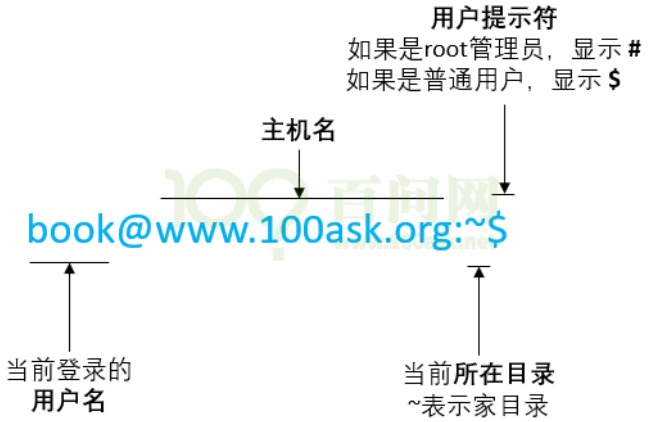


图 2.16 z 终端提示符

3. Linux 命令的格式

Linux 命令一般由三部分组成：

① command 命令；
② options 选项；
③ parameter 参数；

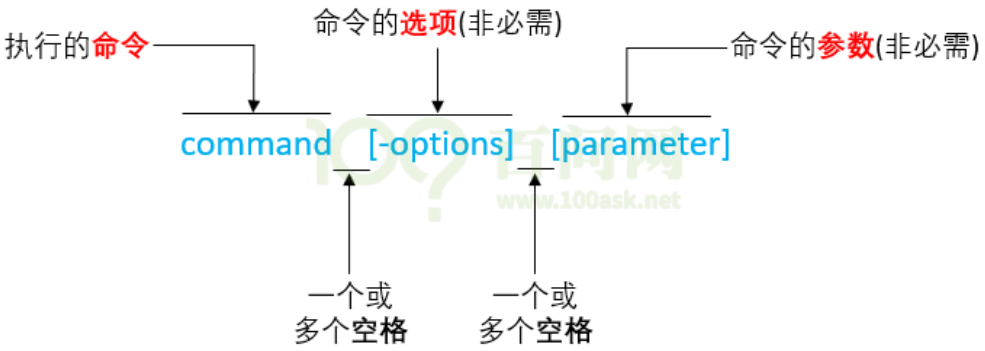


图 2.17 Linux 命令格式

说明：
① [ ]中括号表示 该部分可选，可有可无，需要根据命令的实际需要而添加;
② 命令、选项、参数都以空格分隔，不管几个空格都算一个空格;
③ 命令输入完毕后，按回车“Enter”键启动;

图 2.18 Linux 命令示例

4. 记住命令并不难，先背几个单词

序号	英语单词及含义	
1	directory	目录
2	change	改变
3	list	列出
4	print	打印
5	remove	删除
6	copy	复制
7	move	移动
8	clear	清除

图 2.19 常见命令含义

5. 绝对路径和相对路径

Linux 下的根目录为“/”，从根目录下出发可以找到任意目录、任意文件。从根目录开始表示目录或文件的方法称为“绝对路径”。比如：

```
/home/book
/home/book/1.txt
/bin/pwd
```

有时候使用绝对路径太过麻烦，可以使用相对路径。假设当前正位于/home/book目录下，那么：

```
./1.txt          表示当前目录下的 1.txt，即 /home/book/1.txt；“.”表示当前目录
../book/1.txt    表示当前目录的上一级目录里，book 子目录下的 1.txt
“/home/book/..”就是”/home”目录，”..”表示上一级目录
```

- 使用 . 表示当前路径
- 使用 .. 表示上一级路径
- 使用 ../..表示上上级路径，依此类推。

2.3.2 目录/文件操作命令

1. pwd

- **命令：** pwd
- **英文来源：** print working directory
- **功能：** 打印当前所在路径
- **命令格式：**

命令	选项	参数
pwd	\	\

注：pwd没有其他选项

◆ **示例：** pwd

```
book@www.100ask.org:~$ pwd
/home/book
```

示例功能：显示当前所在路径

图 2.20 pwd

2. cd

- **命令：** cd
- **英文来源：** change directory
- **功能：** 改变路径、切换路径
- **命令格式：**

命令	选项	参数
cd	\	[目录]

◆ **示例：** cd /home/

```
book@www.100ask.org:~$ pwd
/home/book
book@www.100ask.org:~$ cd /home/
book@www.100ask.org:/home$ pwd
/home
```

示例功能：切换到/home路径

图 2.21 cd

cd 命令有些缩略用法：

```
$ cd  -   // 进入上次目录，比如先进入 a 目录再进入 b 目录，执行此命令后即回到 a 目录
$ cd  ~   // 进入家目录
```

3. mkdir

- **命令：** mkdir
- **英文来源：** make directory
- **功能：** 创建目录
- **命令格式：**

命令	选项	参数
mkdir	-p	[目录]

parents

◆ **示例：** mkdir dir0

```
book@www.100ask.org:/work/001_linux_basic$ mkdir dir0
book@www.100ask.org:/work/001_linux_basic$ ls -l
total 4
drwxrwxr-x 2 book book 4096  7月 27 14:57 dir0
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：创建一个目录

◆ **示例：** mkdir -p dir1/dir2

```
book@www.100ask.org:/work/001_linux_basic$ mkdir -p dir1/dir2
book@www.100ask.org:/work/001_linux_basic$ ls -l
total 4
drwxrwxr-x 3 book book 4096  7月 27 14:58 dir1
book@www.100ask.org:/work/001_linux_basic$ cd dir1/
book@www.100ask.org:/work/001_linux_basic/dir1$ ls -l
total 4
drwxrwxr-x 2 book book 4096  7月 27 14:58 dir2
book@www.100ask.org:/work/001_linux_basic/dir1$
```

示例功能：创建目录及子目录

图 2.22 mkdir

4. rmdir

- **命令**：rmdir
- **英文来源**：remove directory
- **功能**：删除目录
- **命令格式**：

命令	选项	参数
mkdir	\	[目录]

◆ **示例**：rmdir dir1

```
book@www.100ask.org:/work/001_linux_basic$ ls
dir1  dir2
book@www.100ask.org:/work/001_linux_basic$ rmdir dir1
book@www.100ask.org:/work/001_linux_basic$ ls
dir2
```

示例功能：删除一个空目录

◆ **示例**：rmdir dir1

```
book@www.100ask.org:/work/001_linux_basic/dir1$ ls
1.txt
book@www.100ask.org:/work/001_linux_basic/dir1$ cd ..
book@www.100ask.org:/work/001_linux_basic$ rmdir dir1
rmdir: failed to remove 'dir1': Directory not empty
```

示例功能：不能删除一个非目录

图 2.23 rmdir

5. ls

- **命令**：ls
- **英文来源**：list
- **功能**：列出目录内容
- **命令格式**：

命令	选项			参数
ls	-l	-a	-h	[目录]

图 2.24 ls

使用示例：

◆ **示例**：ls

```
book@www.100ask.org:/work/001_linux_basic$ ls
dir1  dir2  file1  file2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：显示当前目录下文件

◆ **示例**：ls -l　（long）

```
book@www.100ask.org:/work/001_linux_basic$ ls -l
total 8
drwxrwxr-x 2 hceng hceng 4096  7月 27 11:31 dir1
drwxrwxr-x 2 hceng hceng 4096  7月 27 11:31 dir2
-rw-rw-r-- 1 hceng hceng    0  7月 27 11:31 file1
-rw-rw-r-- 1 hceng hceng    0  7月 27 11:31 file2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：显示文件更完整信息
use a long listing format.

◆ **示例**：ls -a　（all）

```
book@www.100ask.org:/work/001_linux_basic$ ls -a
.  ..  .dir0  dir1  dir2  .file0  file1  file2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：显示当前目录下文件及隐藏文件
do not ignore entries starting with.

◆ **示例**：ls -la

```
book@www.100ask.org:/work/001_linux_basic$ ls -la
total 20
drwxrwxrwx 5 hceng hceng 4096  7月 27 14:54 .
drwxrwxrwx 6 root  root  4096  7月 27 11:30 ..
drwxrwxr-x 2 book  book  4096  7月 27 11:51 .dir0
drwxrwxr-x 2 book  book  4096  7月 27 14:54 dir1
drwxrwxr-x 2 book  book  4096  7月 27 14:54 dir2
-rw-rw-r-- 1 book  book     0  7月 27 11:51 .file0
-rw-rw-r-- 1 book  book     0  7月 27 14:54 file1
-rw-rw-r-- 1 book  book     0  7月 27 14:54 file2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：“-l”和“-a”组合选项，
显示所有文件及完整信息。

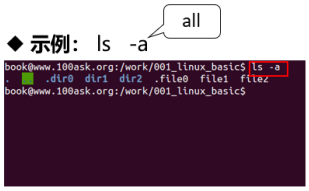
◆ ls –lh　// h表示--human-readable，大小以K/M/G等可读方式列出来

```
book@www.100ask.org:/work/001_linux_basic$ ls -lh
total 8.0K
drwxrwxr-x 2 book book 4.0K 7月 27 14:54 dir1
drwxrwxr-x 2 book book 4.0K 7月 27 14:54 dir2
-rw-rw-r-- 1 book book    0 7月 27 14:54 file1
-rw-rw-r-- 1 book book    0 7月 27 14:54 file2
```

① 文件属性　② 链接数　③ 文件所有者　④ 文件所属用户组　⑤ 文件大小　⑥ 最后修改时间　⑦ 文件名

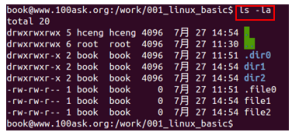
6. cp

◆ 示例：ls -a



示例功能：显示当前目录下文件及隐藏文件
do not ignore entries starting with.

◆ 示例：ls -la



示例功能："-l" 和 "-a" 组合选项，
显示所有文件及完整信息。

◆ ls –lh // h表示--human-readable，大小以K/M/G等可读方式列出来

```
book@www.100ask.org:/work/001_linux_basic$ ls -lh
total 8.0K
drwxrwxr-x 2 book book 4.0K 7月 27 14:54 dir1
drwxrwxr-x 2 book book 4.0K 7月 27 14:54 dir2
-rw-rw-r-- 1 book book    0 7月 27 14:54 file1
-rw-rw-r-- 1 book book    0 7月 27 14:54 file2
```

① 文件属性 ② 链接数 ③ 文件所有者 ④ 文件所属用户组 ⑤ 文件大小 ⑥ 最后修改时间 ⑦ 文件名

◆ 示例：cp -r dir1/ dir2/

```
book@www.100ask.org:/work/001_linux_basic$ ls
dir1
book@www.100ask.org:/work/001_linux_basic$ cp -r dir1 dir2
book@www.100ask.org:/work/001_linux_basic$ ls
dir1  dir2
```

示例功能：复制dir1文件夹

复制目录时，常用如下命令：

```
$ cp -rfd dir_a dir_b
```

- r：recursive，递归地，即复制所有文件
- f：force，强制覆盖
- d：如果源文件为链接文件，也只是把它作为链接文件复制过去，而不是复制实际文件

7. rm

- **命令**：rm
- **英文来源**：remove
- **功能**：删除文件或目录
- **命令格式**：

命令	选项	参数
rm	-r -p	文件 或 文件夹

◆ 示例：rm file1

```
book@www.100ask.org:/work/001_linux_basic$ ls
file1  file2
book@www.100ask.org:/work/001_linux_basic$ rm file1
book@www.100ask.org:/work/001_linux_basic$ ls
file2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：删除文件

◆ 示例：rm -r dir1/

```
book@www.100ask.org:/work/001_linux_basic$ ls
dir1  dir2
book@www.100ask.org:/work/001_linux_basic$ rm -r dir1/
book@www.100ask.org:/work/001_linux_basic$ ls
dir2
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：删除文件夹

删除目录时，常用如下命令：

```
$ rm -rf dir_a
```

- r：recursive，递归地，即删除所有文件
- f：force，强制删除

8. cat

- **命令：** cat
- **英文来源：** \
- **功能：** 串联文件的内容并打印出来
- **命令格式：**

命令	选项	参数
cat	\	文件

◆ **示例：** cat file1.txt file2.txt

```
book@www.100ask.org:/work/001_linux_basic$ ls
file1.txt  file2.txt
book@www.100ask.org:/work/001_linux_basic$ cat file1.txt file2.txt
hello world.
www.100ask.org
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：串联文件并依次全部打印在标准输出中
Concatenate files and print on the standard output

9. touch

- **命令：** touch
- **英文来源：** \
- **功能：** 修改文件的时间，如果文件不存在则创建空文件
- **命令格式：**

命令	选项	参数
touch	\	文件名

◆ **示例：** touch file

```
book@www.100ask.org:/work/001_linux_basic$ touch file
book@www.100ask.org:/work/001_linux_basic$ ls -l
total 0
-rw-rw-r-- 1 book book 0  7月 27 15:00 file
book@www.100ask.org:/work/001_linux_basic$
```

示例功能：创建一个文件file

2.3.3 改变文件的权限和属性

- chgrp：改变文件所属用户组
- chown：改变文件所有者
- chmod：改变文件的权限

1. chgrp 改变文件所属用户组

```
chgrp 【-R】 dirname/filename  ...
```

- -R:进行递归的持续更改，也连同子目录下的所有文件、目录都更新成为这个用户组之意。常常用在更改某一目录内所有文件的情况。

范例：

```
chgrp  hy  install.log
```

将 install.log 文件的用户组改为 hy 用户组。注意 hy 用户组必须要在

/etc/group 文件内存在才可以。

2. chowm 改变文件的所有者

```
chown [-R]  账号名  文件或目录
chown [-R] 账号名:组名  文件或目录
```

- -R：也是递归子目录。

范例：

```
chown bin install.log
chown  book:book  install.log
```

改变文件所有者和用户组的这两个命令的应用场景：复制文件，由于复制行为会复制执行者的属性和权限，因此复制后需要改变文件所属用户、用户组等。

3. chmod 改变文件的权限

文件权限有两种设置方法：数字类型改变权限和符号改变权限。

首先说明各个权限对应的数字：

- r: 4 或 0
- w: 2 或 0
- x: 1 或 0

这 3 种权限的取值相加后，就是权限的数字表示。例如：文件 a 的权限为“-rwxrwx---”，它的数值表示为：

- owner = rwx = 4+2+1 = 7
- group = rwx = 4+2+1 = 7
- others = --- = 0+0 +0 = 0

所以在设置权限时，该文件的权限数字就是 770。

④ 数字类型改变权限

使用数值改变文件权限的命令如下：

```
chmod [-R]  xyz  文件或目录
```

- xyz：代表权限的数值，如 770。
- -R：以递归方式进行修改，比如修改某个目录下所有文件的属性。

范例：

```
chmod 777 .bashrc
```

将文件.bashrc 这个文件的所有权限设置都启用。

⑤ 符号类型改变文件权限方式

使用 u、g、o 三个字母代表 user、group、others 3 中身份。此外 a 代表 all，即所有身份。

范例：

```
chmod u=rwx,go=rx  .bashrc
```

也可以增加或去除某种权限，“+”表示添加权限，“-”表示去除权限：

```
chmod a+w  .bashrc
chmod a-x  .bashrc
```

2.3.4 查找/搜索命令

1. find

在 Windows 中搜索文件，一般查找文件需要传入两个条件：

① 在哪些目录中查找；

② 查找的内容；

在 Linux 中，查找文件的也需要这两个条件，不同于 Windows 使用搜索框查找，Linux 中使用 find 命令查找文件。

find 命令格式为：

```
find 目录名 选项 查找条件
```

举例 1：

```
$ find  /home/book/dira/  -name  " test1.txt "
```

说明：

- /home/book/dira/指明了查找的路径。
- “-name”表明以名字来查找文件 。
- “test1.txt”，就指明查找名为“test1.txt”的文件。

举例 2：

```
$ find  /home/book/dira/ -name  " *.txt "
```

说明:查找指定目录下面所有以“.txt”结尾的文件，其中“*”是通配符。

举例 3：

```
find  /home/book/dira/  -name "dira"
```

说明： 查找指定目录下面是否存在“dira”这个目录或文件，“dira”是名称。

注意：

① 如果没有指定查找目录，则为当前目录。

```
$ find . -name " *.txt "    //其中.代表当前路径。
$ find -name " *.txt "      //没加路径，默认是当前路径下查找。
```

② find 还有一些高级的用法，如查找最近几天(几个小时)之内(之前)有变动的文件

```
$ find  /home/book  -mtime -2       //查找/home 目录下两天内有变动的文件。
```

2. grep

grep 命令的作用是查找文件中符合条件的字符串，其格式如下：

```
grep [选项] [查找模式] [文件名]。
```

假设 dira 目录的 test1.txt 和 dirb 目录的 test1.txt 都含有如下内容：`aaa AAAAAA abc abcabcabc cbacbacba match_pattern nand->erase。`

通过查找字符串，希望显示如下内容：

① 所在的文件名----grep 查找时默认已经显示目标文件名

② 所在的行号------使用-n 选项。

`grep -rn "字符串" 文件名 r(recursive)：递归查找 n(number)：`显示目标位置的行号

- 字符串:要查找的字符串

- 文件名:要查找的目标文件，如果是*则表示查找当前目录下的所有文件和目录。

举例：

```
//在 test1.txt 中查找字符串 abc grep -rn "abc" * 在当前目录递归查找字符串 abc
$ grep -n "abc" test1.txt
```

注意：可以加入-w 全字匹配。

可以在 grep 的结果中再次执行 grep 搜索，比如搜索包含有 ABC 的头文件，可执行如下命令：

```
$ grep “ABC” * -nR | grep “\.h”
```

上述命令把第 1 个命令“grep “ABC” * -nR”通过管道传给第 2 个命令。即第 2 个命令在第 1 个命令的结果中搜索。

2.3.5 压缩/解压缩命令

1. gzip

gzip 的常用选项：

- -l(list) 列出压缩文件的内容。
- -k(keep) 在压缩或解压时，保留输入文件。
- -d(decompress) 将压缩文件进行解压缩。

举例：

① 查看压缩文件

```
$ gzip -l pwd.1.gz
```

② 解压文件

```
$ gzip -kd pwd.1.gz //该压缩文件是以.gz 结尾的单个文件
```

③ 压缩文件

```
$ gzip -k mypwd.1 /得到了一个.gz 结尾的压缩文件
```

注意：

a) 如果 gzip 不加任何选项，此时为压缩。压缩完该文件会生成后缀为.gz 的压缩文件，并删除原来的文件。所以，推荐使用 gzip -k 来压缩源文件，这样会保留原来的文件。

b) 相同的文件内容，如果文件名不同，压缩后的大小也不同。

c) gzip 只能压缩单个文件，不能压缩目录。

2. bzip2

bzip2 的常用选项：

- -k(keep)在压缩或解压时，保留输入文件；
- -d(decompress) 将压缩文件进行解压缩；

① 压缩文件

```
$ bzip2 -k mypwd.1 得到一个.bz2 后缀的压缩文。
```

② 解压文件

```
$ bzip2 -kd mypwd.1.bz2
```

注意：

- 如果 bzip2 不加任何选项，此时为压缩
- 压缩完该文件会生成后缀为.bz2 的压缩文件， 并删除原来的文件。所以说，推荐使用 bzip2 -k 来压缩文件，这样可以保留原来的文件。
- bzip2 只能压缩单个文件，不能压缩目录。
- 单个文件的压缩使用 gzip 或 bzip2，压缩有两个参数：
  a) 压缩时间
  b) 压缩比。

一般情况下，小文件使用 gzip 来压缩，大文件使用 bzip2 来压缩。bzip2 的的压缩率更高。

3. tar

tar 常用选项：

- -c(create)：表示创建用来生成文件包 。
- -x：表示提取，从文件包中提取文件。
- -t：可以查看压缩的文件。
- -z：使用 gzip 方式进行处理，它与”c“结合就表示压缩，与”x“结合就表示解压缩。
- -j：使用 bzip2 方式进行处理，它与”c“结合就表示压缩，与”x“结合就表示解压缩。
- -v(verbose)：详细报告 tar 处理的信息。
- -f(file)：表示文件，后面接着一个文件名。 -C <指定目录> 解压到指定目录。

例 1：tar 打包、gzip 压缩

① 把目录 dira 压缩、打包为 dira.tar.gz 文件：

```
$ tar czvf dira.tar.gz dira。
```

注意：“tar －czvf”与“tar czvf”是一样的效果，所以说，后面统一取消“-”。

② 查看压缩文件：

```
$ tar tvf  dira.tar.gz
```

③ 解压文件，可以用-C 指定解压到哪个目录：

```
$ tar xzvf dira.tar.gz                  //解压到当前目录
$ tar xzvf dira.tar.gz -C /home/book    //解压到/home/book。
```

例 2：tar 打包、bzip2 压缩

④ 把目录 dira 压缩、打包为 dira.tar.bz2 文件

```
$ tar cjvf dira.tar.bz2 dira
```

⑤ 查看压缩文件

```
$ tar tvf dira.tar.bz2
```

⑥ 解压文件，可以用-C 指定解压到哪个目录

```
$ tar xjvf dira.tar.bz2                  //解压到当前目录:
$ tar xjvf dira.tar.bz2 -C /home/book     //解压到/home/book
```

2.3.6 网络命令

注意：只要你的 Ubunut 能上网，比如执行"ping www.baidu.com"成功，那么就不要按本章来设置网络。

注意：如果你的 Ubunut 不能上网，请按《>>>>>>>>第三篇第 1 章配置 VMware 使用双网卡》，也不需要阅读本章。

注意：本章只是介绍性的文档，不要执行它的命令。

1. ifconfig

查看网络、设置 IP。ifconfig 常用选项：

- -a ：显示所有网卡接口
- up：激活网卡接口
- down：关闭网卡接口
- address：xxx.xxx.xxx.xxx，IP 地址

示例：

① fconfig：查看当前正在使用的网卡

```
$ ifconfig
```

```
ubuntu@ubuntu1804:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.246.130  netmask 255.255.255.0  broadcast 192.168.246.255
        inet6 fe80::f629:2a76:8be5:f671  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:9a:02:ef  txqueuelen 1000  (Ethernet)
        RX packets 11016  bytes 1384737 (1.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8758  bytes 1044826 (1.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 12981  bytes 3244821 (3.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12981  bytes 3244821 (3.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 2.25

② ifconfig -a：查看所有网卡

```
$ ifconfig -a
```

```
ubuntu@ubuntu1804:~$ ifconfig -a
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.246.130  netmask 255.255.255.0  broadcast 192.168.246.255
        inet6 fe80::f629:2a76:8be5:f671  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:9a:02:ef  txqueuelen 1000  (Ethernet)
        RX packets 11016  bytes 1384737 (1.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8758  bytes 1044826 (1.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 12981  bytes 3244821 (3.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12981  bytes 3244821 (3.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 2.26

③ 设置网 IP：

```
$ sudo ifconfig ens160 192.168.1.137
```

2. route 和 DNS

确保 Windows 和 Ubuntu 的网络能互相 ping 通之后，如果 Ubuntu 无法上网，原因通常有 2 个：路由没设置好，DNS 没设置好。

如果执行以下命令不成功，表示路由没设置好：

```
$ ping  8.8.8.8
connect: Network is unreachable
```

如果“ping 8.8.8.8”成功，但是“ping www.baidu.com”不成功，则是 DNS 没设置好：

```
$ ping www.baidu.com
ping: unknown host www.baidu.com
```

DNS 的设置比较简单，8.8.8.8 是好记好用的 DNS 服务器，修改 Ubuntu 中的 /etc/resolv.conf 文件，内容如下：

```
nameserver  8.8.8.8
```

路由信息使用 route 命令查看，其输出信息可以参考链接：

https://akaedu.github.io/book/ch36s05.html

摘录如下：

假设某主机上的网络接口配置和路由表如下：

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:C2:8D:7E
          inet addr:192.168.10.223  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:420 (420.0 b)
          Interrupt:10 Base address:0x10a0

eth1      Link encap:Ethernet  HWaddr 00:0C:29:C2:8D:88
          inet addr:192.168.56.136  Bcast:192.168.56.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:603 errors:0 dropped:0 overruns:0 frame:0
          TX packets:110 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:55551 (54.2 Kb)  TX bytes:7601 (7.4 Kb)
          Interrupt:9 Base address:0x10c0

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:37 errors:0 dropped:0 overruns:0 frame:0
          TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3020 (2.9 Kb)  TX bytes:3020 (2.9 Kb)
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.10.0    *               255.255.255.0   U     0      0        0 eth0
```

```
192.168.56.0    *               255.255.255.0   U     0      0        0 eth1
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
default         192.168.10.1    0.0.0.0         UG    0      0        0 eth0
```

上述 route 命令输出信息中各项的含义请看下表：

Destination	目标网段或者主机
Gateway	网关地址，”*” 表示目标是本主机所属的网络，不需要路由
Genmask	网络掩码
Flags	标记。一些可能的标记如下：
	U - 路由是活动的
	H - 目标是一个主机
	G - 路由指向网关
	R - 恢复动态路由产生的表项
	D - 由路由的后台程序动态地安装
	M - 由路由的后台程序修改
	! - 拒绝路由
Metric	路由距离，到达指定网络所需的中转数
Ref	路由项引用次数
Use	此路由项被路由软件查找的次数
Iface	该路由表项对应的输出接口

在上面的例子中，这台主机有两个网络接口：

① 一个网络接口连到 192.168.10.0/24 网络

② 另一个网络接口连到 192.168.56.0/24 网络。

如果要发送的数据包的目的地址是 192.168.56.3，跟第一行的子网掩码做与运算得到 192.168.56.0，与第一行的目的网络地址不符，再跟第二行的子网掩码做与运算得到 192.168.56.0，正是第二行的目的网络地址，因此从 eth1 接口发送出去，由于 192.168.56.0/24 正是与 eth1 接口直接相连的网络，因此可以直接发到目的主机，不需要经路由器转发。

如果要发送的数据包的目的地址是 202.10.1.2，跟前三行路由表条目都不匹配，那么就要按缺省路由条目，从 eth0 接口发出去，首先发往 192.168.10.1 路由器，再让路由器根据它的路由表决定下一跳地址。

可以使用 route 命令管理路由。

示例：

a) 添加路由：首先得确定网关 IP，假设为 192.168.1.1

```
$ sudo  route  add  default  gw  192.168.1.1
$ ping 8.8.8.8   // 验证
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=53 time=19.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=53 time=19.8 ms
```

b) 删除路由：

```
$ sudo  route  del  default  gw  192.168.1.1
```

2.3.7 vi 编辑器

vi 是一个命令，也是一个命令行下的编辑器，它有如下功能：

- 打开文件、新建文件、保存文件
- 光标移动
- 文本编辑
- (多行间|多列间)复制、粘贴、删除
- 查找和替换

很多人不习惯在命令行下编辑文件，实际开发中也不会经常在命令行下编辑文件。但是在 Linux 系统中对文件做些简单修改时，使用 vi 命令的效率非常高。并且在很多时候，比如现场调试时，并没有 GUI 形式的编辑工具，vi 是唯一选择。

1. 模式

vi 编辑器有三种模式，各个模式侧重点不一样：

a)一般模式（光标移动、复制、粘贴、删除）

b)编辑模式（编辑文本）

c)命令行模式（查找和替换）

vi 编辑器的三种模式间切换如下图所示：

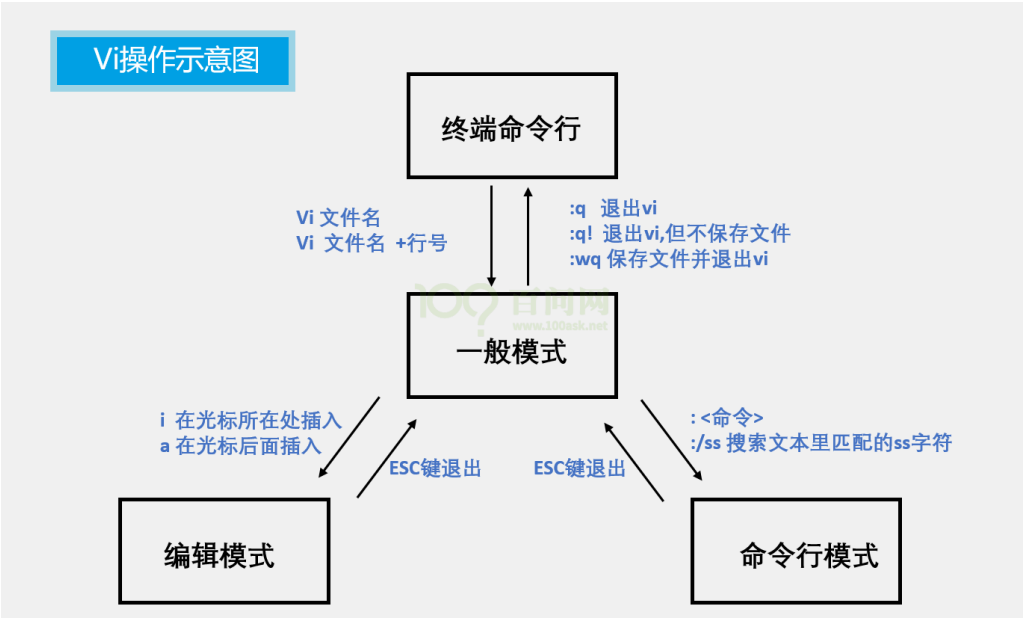


图 2.27

注意：

- 当不知道处于何种模式时，按 ESC 键返回到一般模式。
- wq(write quit)
- i(insert)

2. 文件的打开/新建/保存

打开文件、新建文件，命令如下(如果文件存在则打开文件，否则新建文件并打开)：

```
$ vi   文件名
```

修改结束之后，输入“:” 进入命令行模式，再输入“wq”保存退出：

```
:wq      保存并退出文件
```

注意：如果文件不存在，也需要输入“:wq”才可以保存新文件，否则不会新建文件。

在编辑完成时，返回一般模式，方法如下：

第1步 输入“:w”则保存文件，如果已经保存文件，输入“:q”则退出文件

第2步 直接输入“:wq”保存并退出

第3步 如果不想保存被修改的内容，则输入“:q!”强制退出。

这些命令列表如下：

命令	描述
x	保存当前文档并且退出。
q	退出。
w	保存文档 。
q!	退出 vi/vim，不保存文档。

3. 编辑文件

打开文件后，默认处于“一般模式”，这时可以输入以下字母：

指令	描述
i	在当前光标所在字符的前面，转为编辑模式。
I	在当前光标所在行的行首转换为编辑模式。
a	在当前光标所在字符的后面，转为编辑模式。
A	在光标所在行的行尾，转换为编辑模式。
o	在当前光标所在行的下方，新建一行，并转为编辑模式。
O	在当前光标所在行的上方，新建一行，并转为编辑模式。

4. 快速移动光标

在一般模式下，可以使用下面快捷键移动光标或是翻页：

移动光标	
h （或左方向键）	光标左移一个字符。
l （或右方向键）	光标右移一个字符。
j （或下方向键）	光标下移一行。
k （或上方向键）	光标上移一行。
nG 或 ngg	光标移动到第 n 行首。
n+	光标下移 n 行。

n-	光标上移 n 行。
屏幕翻滚	
Ctrl + f	屏幕向下翻一页，相当于下一页。
Ctrl + b	屏幕向上翻一页，相当于上一页。

详细介绍如下：

① 快速的定位到某一行：文件头、文件尾、指定某一行

- ngg：光标移至第 n 行的行首（n 为数字,想要跳转的行），
- 1gg：就跳到第一行的行首，就是文件头
- 2gg：就跳到第二行的行首
- G：转至文件结尾

② 在某一行如何快速定位到某一列：

- 0:（数字零）光标移至当前行行首
- $：光标移至当前行行末
- fx：搜索当前行中下一个出现字母 x 的地方

注意：当你不知道 vi 当前处于何种模式时，使用 esc 键返回到一般模式。

5．文本复制/粘贴/删除/撤销

在一般模式下，可以执行以下命令：

复制、删除和粘贴	
cc	删除整行，并且修改整行内容。
dd	删除该行，不提供修改功能。
ndd	删除当前行向下 n 行。
x	删除光标所在的字符。
X	删除光标前面的一个字符。
nyy	复制当前行及其下面 n 行。
p	粘贴最近复制的内容。
s	删除光标所在字符。
r	替换光标处字符。

a) 复制

```
yy   //复制当前行(y:yank(复制))
nyy  //复制当前行及其后的 n-1 行(n 是数字)
```

b) 粘贴

```
p  //粘贴(p:paste)
```

c) 删除

```
dd   //删除光标所在行(d:delete)
ndd  //删除当前行及其后的 n-1 行(n 是数字)
x    //删除光标所在位置的字符
```

d) 撤销

```
u  //撤销上一步操作
```

6. 文本查找和替换

在一般模式下，可以执行以下命令。

a) 查找

```
/pattern  //从光标开始处向文件尾搜索 pattern，后按下 n 或 N
```

注意：

- n：在同一个方向重复上一次搜索命令
- N：在反方向重复上一次搜索命令
- 在/pattern 之前先跳到第一行则进行全文件搜索。

b) 替换

- :%s/p1/p2/g //将文件中所有的 p1 均用 p2 替换
- :%s/p1/p2/gc //替换时需要确认
- “s“ 全称：substitute 替换；
- “g“ 全称：global 全局；
- “c“ 全称：confirm，确认

参考

https://www.runoob.com/linux/linux-command-manual.html

7. vi 编辑器使用示例

本例创建一个名为 hello.txt 的文件，添加如下 2 行文件字，再回退去删除第 2 行的字母“X”：

```
www.100ask.net
wiki.100askX.net
```

首先执行以下命令，如果当前目录下没有 hello.txt 它就会新建并打开该文件，否则直接打开该文件：

```
vi  hello.txt
```

界面如下：

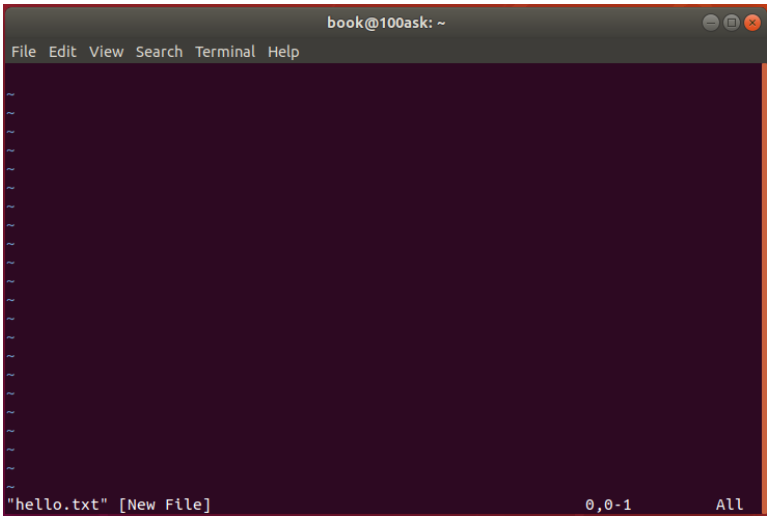


图 2.28

输入字符“i”，它表示“insert”，即在光标前输入字符，界面如下：

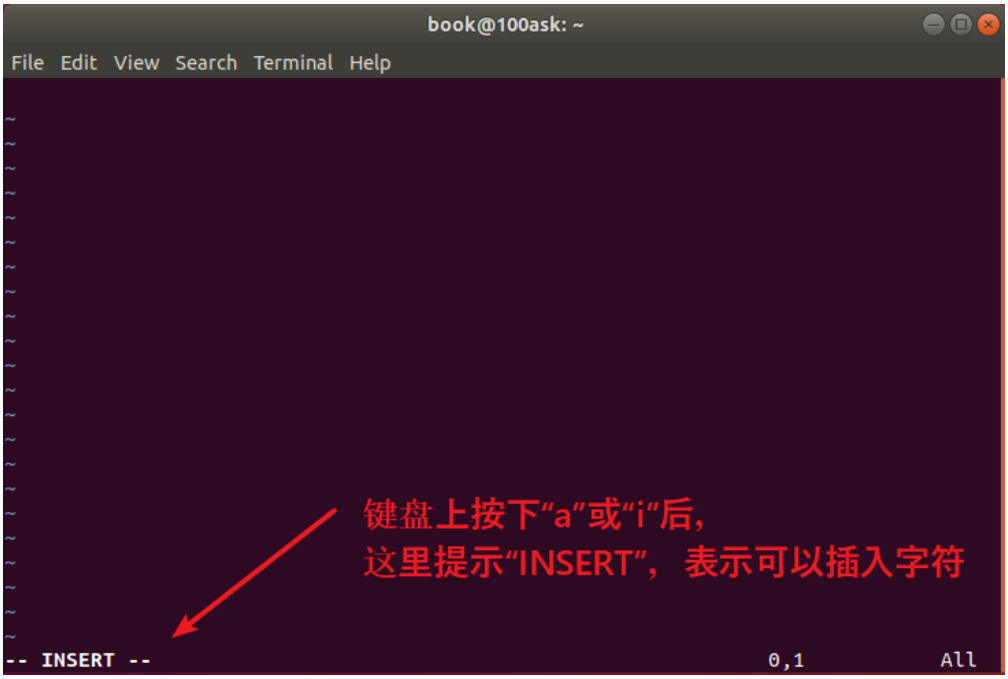


图 2.29

在上述界面中，可以输入字符了，可以使用回车键、删除键、箭头键等等，跟一般得文本工具没什么差别，如下：

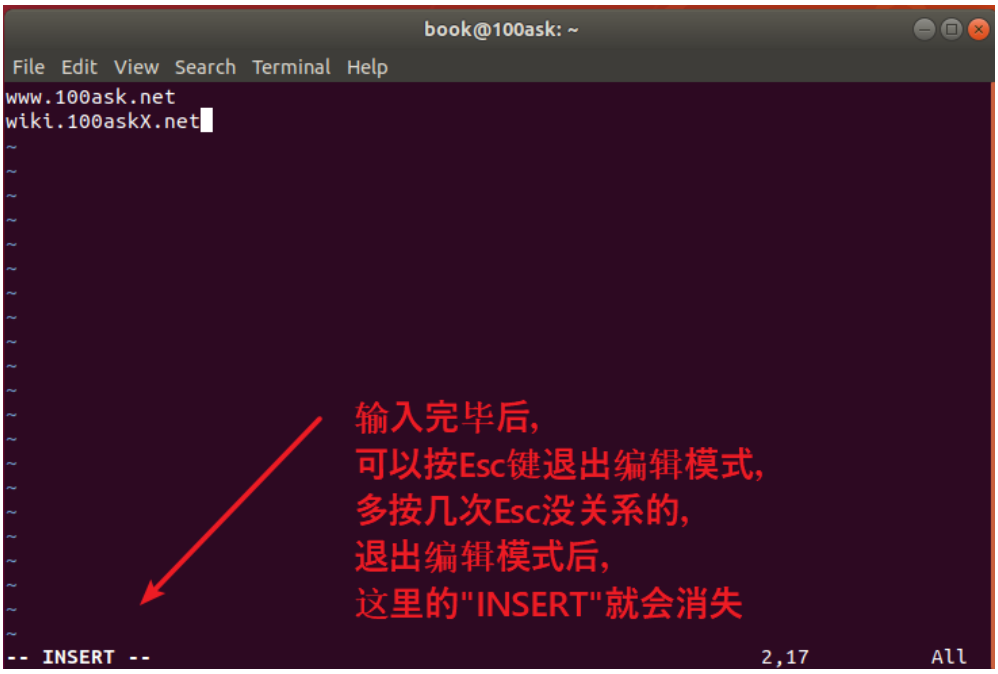


图 2.30

无论当前是否处于编辑模式，都可以使用箭头键移动光标。

- 在编辑模式下，使用删除键(Backspace)删除字符；
- 不在编辑模式下时，使用“x”删除光标所在的字符；

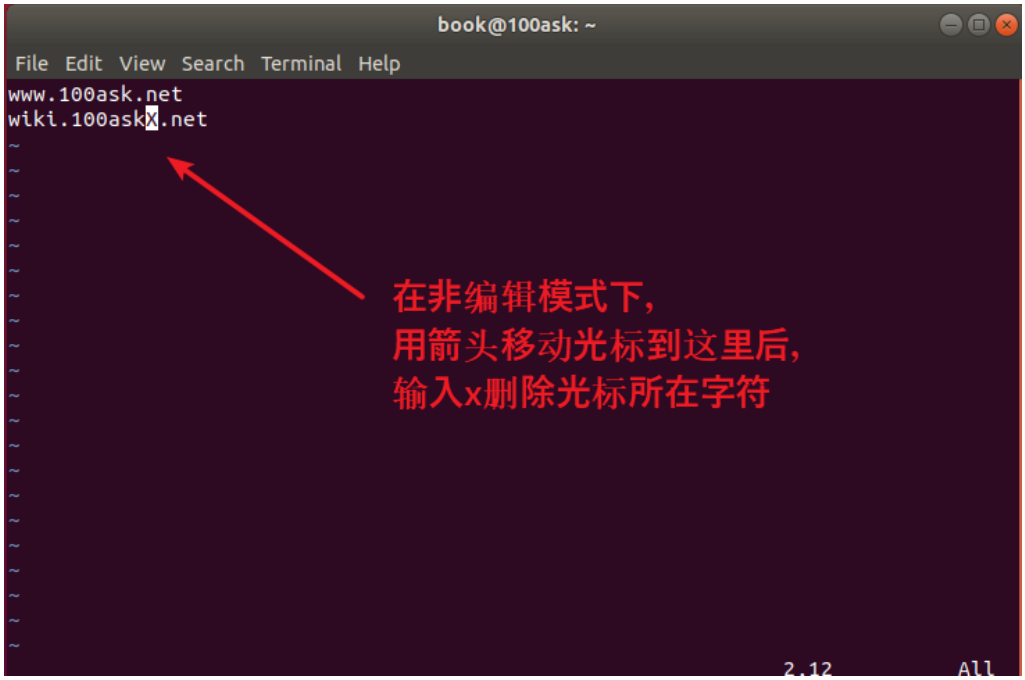


图 2.31

最后输入“:wq”保存并退出，“:”表示进入命令行模式，“wq”表示“write and quit”，即写入并退出。如果不想保存则可以输入“:q!”，它表示“退出、不保存”：

图 2.32

最后执行“cat hello.txt”命令验证一下：

```
book@100ask:~$ cat hello.txt
www.100ask.net
wiki.100ask.net
book@100ask:~$
```

图 2.33

注意：本教程没有教你怎么在 Ubuntu 中安装中文输入法，因为我们是在 Windows 编辑源码，或是提倡多英文写注释。如果你需要在 Ubuntu 下使用中文，请自行百度。

2.3.8 其他命令

1. file

查看文件类型。其格式如下：

```
file 文件名
```

使用 gcc 编译得到的程序是运行于 PC 的，但是很多初学者经常把这些程序放到 ARM 板子上去运行，这时一般都会提示：

```
xxx  not found
```

它并非“找不到”，而是格式不正确。这时你可以执行“`file xxx`”查看它的类型，确定它是给 PC 还是给 ARM 编译的。

2. which 和 whereis

which 命令和 whereis 命令作用是查找命令或应用程序的所在位置，其格式如下：

```
which   命令名/应用程序名
whereis  命令名/应用程序名。
```

示例：

```
$ which pwd     //定位到/bin/pwd
$ which gcc     //定位到/usr/bin/gcc
$ whereis pwd   //可得到可执行程序的位置和手册页的位置
```

2.4 Ubuntu 下包管理

2.4.1 软件包管理系统

像我们日常使用的 windows 提供的应用商店或者手机提供的应用市场那样，大多数现代的类 Unix 操作系统也都提供了一种中心化的机制用来搜索和安装软件。软件通常存放在存储库中，并通过包的形式对外进行分发。处理包的工作称为包管理。包提供了操作系统的基本组件，以及共享的库、应用程序、服务和文档。这个我们称为软件包管理系统，其除了安装软件外，它还提供了工具来更新已经安装的包。

大多数软件包系统都是围绕软件包文件的集合构建的。软件包文件通常是一个存档文件，它包含已编译的二进制文件和软件的其他资源，以及安装脚本。软件包文件同时也包含有价值的元数据，包括它们的依赖项，以及安装和运行它们所需的其他软件包的列表。

虽然这些包管理系统的功能和优点大致相同，但打包格式和工具会因平台（不同的 Linux 发行版）而异，如下表所示：

操作系统	格式	工具
Debian	.deb	apt, apt-cache, apt-get, dpkg
Ubuntu	.deb	apt, apt-cache, apt-get, dpkg
CentOS	.rpm	yum
Fedora	.rpm	dnf
FreeBSD	Ports, .txz	make, pkg

由上表可知，Debian 及其衍生版，如我现在使用的 Ubuntu，它们的包格式为 .deb。APT 这款先进的软件包管理工具提供了大多数常见的命令如：搜索存储库、安装软件包及其依赖项，并管理升级，等等。在系统中，我们还可以使用 dpkg 程序来安装单个的 deb 文件，APT 命令作为底层 dpkg 的前端，有

时也会直接调用它。

目前发布的 debian 衍生版大多数都包含了 apt 命令，它提供了一个简洁统一的接口，可用于通常由 apt-get 和 apt-cache 命令处理的常见操作。这个命令是可选的，但使用它可以简化一些任务。

2.4.2 apt 命令使用

上一小节介绍了软件包管理系统，本小节秉承上小节的内容，介绍软件包管理系统中 apt 命令。

我们习惯于使用 windows 系统以及智能手机，这两个下载和安装软件都是非常容易简单的。但是 Ubuntu 下该如何安装软件呢？Ubuntu 安装软件不像 Windows 下那样，直接双击.exe 文件就开始安装了。一般来说 Ubuntu 下很多软件是需要先自行提供源码，使用源码自行编译，编译完成以后使用命令”install” 来安装到系统中。当然 Ubuntu 下也有其它的软件安装方法，使用得最多的方法就是自行编译源码后进行安装，尤其是嵌入式 Linux 开发。命令 ”install” 格式如下所示：

```
install [选项 ]... [-T] 源文件 目标文件
install [选项 ]... 源文件 ... 目录
install [选项 ]... -t 目录 源文件 ... 或： install [选项 ]... -d 目录 ...
```

“install” 命令是将文件 (通常是编译后的文件)复制到目的位置，在上面得三种形式中，将源文件复制到目标文件或将多个源文件复制到一个已存在的目录中同时设置其所有权和权限模式。在第四种形式会创建指定的目录。命令“ install”通常和命令 apt-get”组合在一起使用的。

其实 Ubuntu 中我们可以直接安装很多的软件&游戏，但是对于刚刚接触 Linux 系统的我们往往会不知如何安装安装软件，其实我们利用软件包管理系统可以直接下载并安装所有通过认证的软件，其中 Ubuntu 下我们用的最多的下载工具： APT 下载工具， APT 下载工具可以实现软件自动下载、配置、安装二进制或者源码的功能。

APT 下载工具和上面讲解到的 ”install” 命令结合构成了 Ubuntu 下最常用的下载和安装软件方法。并且其解决了 Linux 平台下一安装软件的一个缺陷：即软件之间相互依赖问题。

2.4.3 更换软件源

APT 采用 C/S 模式，也就是客户端/服务器模式，一般来说我们的 PC 机作为客户端，当需要下载

软件的时候就向服务器请求，因此我们需要知道服务器的地址，也叫做软件源或者更新源，这个一般默认使用的是国外的软件源（服务器）。因为我们是在中国，如果没有选择中国的服务器，可能会导致下载速度非常慢甚至是下载失败。所以我们需要修改软件源为国内的服务器，这里由两种方法可以进行修改：

1．在桌面环境中修改软件源：

打开系统设置，打开“软件和更新”设置，打开以后如下图所示：

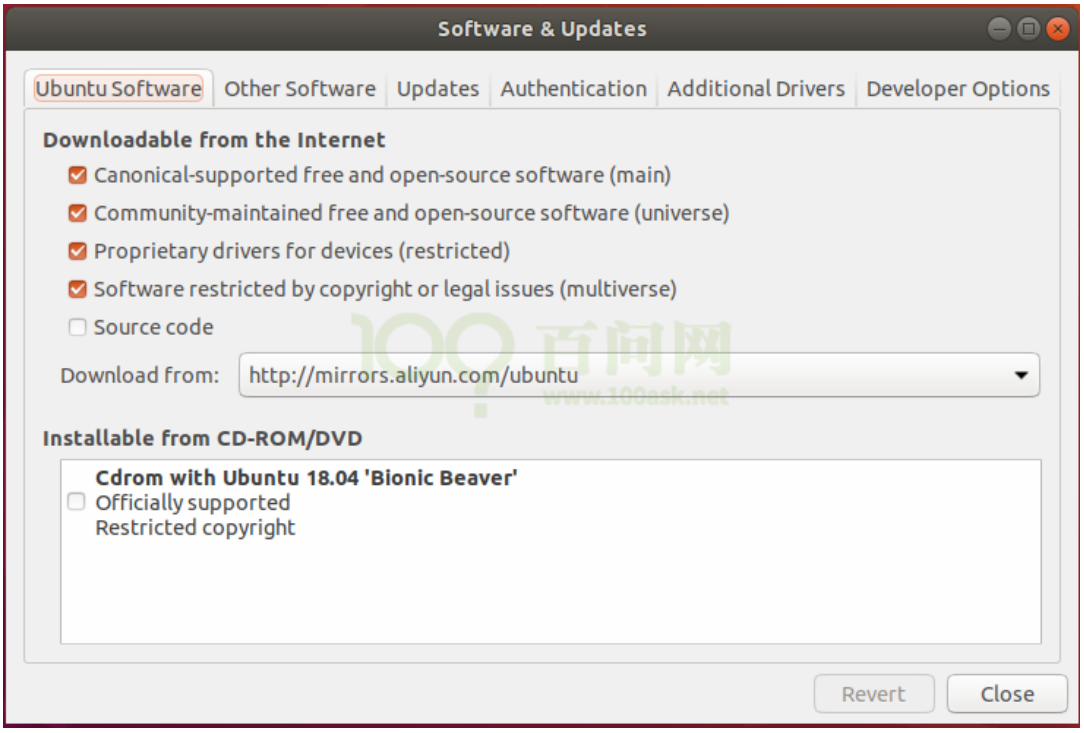


图 2.34

上图中的“Ubuntu 软件”选项卡下面的“下载自”就是 APT 工具的安装源。

2. 修改配置文件

Ubuntu 配置的默认源并不是国内的服务器，下载更新软件都比较慢。首先备份源列表文件 sources.list：

- 首先备份源列表

```
sudo cp /etc/apt/sources.list /etc/apt/sources.list_backup
```

打开 sources.list 文件修改选择合适的源，替换原文件的内容，保存编辑好的文件，以阿里云更新服务器为例：

- 打开 sources.list 文件

```
sudo vim /etc/apt/sources.list
```

编辑 /etc/apt/sources.list 文件，在文件最前面添加阿里云镜像源：

```
#  阿里源
deb http://mirrors.aliyun.com/Ubuntu/ bionic main restricted universe multiverse
deb http://mirrors.aliyun.com/Ubuntu/ bionic-security main restricted universe multi
verse
deb http://mirrors.aliyun.com/Ubuntu/ bionic-updates main restricted universe multiv
erse
deb http://mirrors.aliyun.com/Ubuntu/ bionic-proposed main restricted universe multi
verse
deb http://mirrors.aliyun.com/Ubuntu/ bionic-backports main restricted universe mult
iverse
deb-src http://mirrors.aliyun.com/Ubuntu/ bionic main restricted universe multiverse
deb-src http://mirrors.aliyun.com/Ubuntu/ bionic-security main restricted universe m
ultiverse
deb-src http://mirrors.aliyun.com/Ubuntu/ bionic-updates main restricted universe mu
ltiverse
```

```
deb-src http://mirrors.aliyun.com/Ubuntu/ bionic-proposed main restricted universe m
ultiverse
deb-src http://mirrors.aliyun.com/Ubuntu/ bionic-backports main restricted universe
multiverse
```

在我们使用 APT 工具下载安装或者更新软件的时候，首先会在下载列表中与本机软件进行对比，看一下需要下载哪些软件，或者升级哪些软件，默认情况下 APT 会下载并安装最新的软件包，被安装的软件包所依赖的其它软件也会被下载安装或者更新，非常智能省心。

2.5 Ubuntu 的 gedit 编辑器

进行文本编辑是非常常用的操作，在 windows 系统为我们默认提供了记事本让我们可以进行文本编辑，Ubuntu 系统下也为我们提供一些文本编辑工具来让我们使用，如 gedit、vi/vim。

gedit 是一个窗口式的编辑器，只能在 Ubuntu 桌面环境下使用。

2.5.1 在 GUI 环境下打开 gedit

我们一般在使用 Ubuntu 的文件管理器时，直接双击文本文件默认使用的就是“Gedit”编辑器。

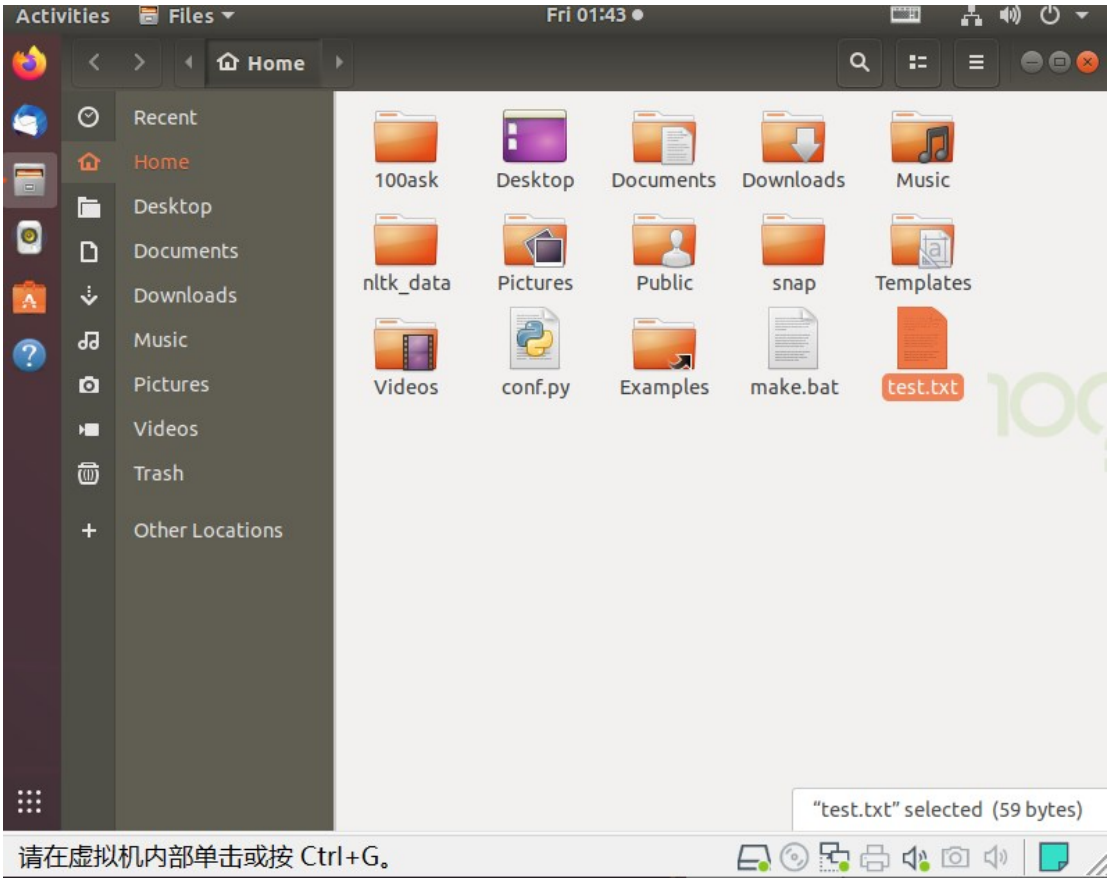


图 2.35

或者单击右键需要编辑的文本文件，选择应用打开：

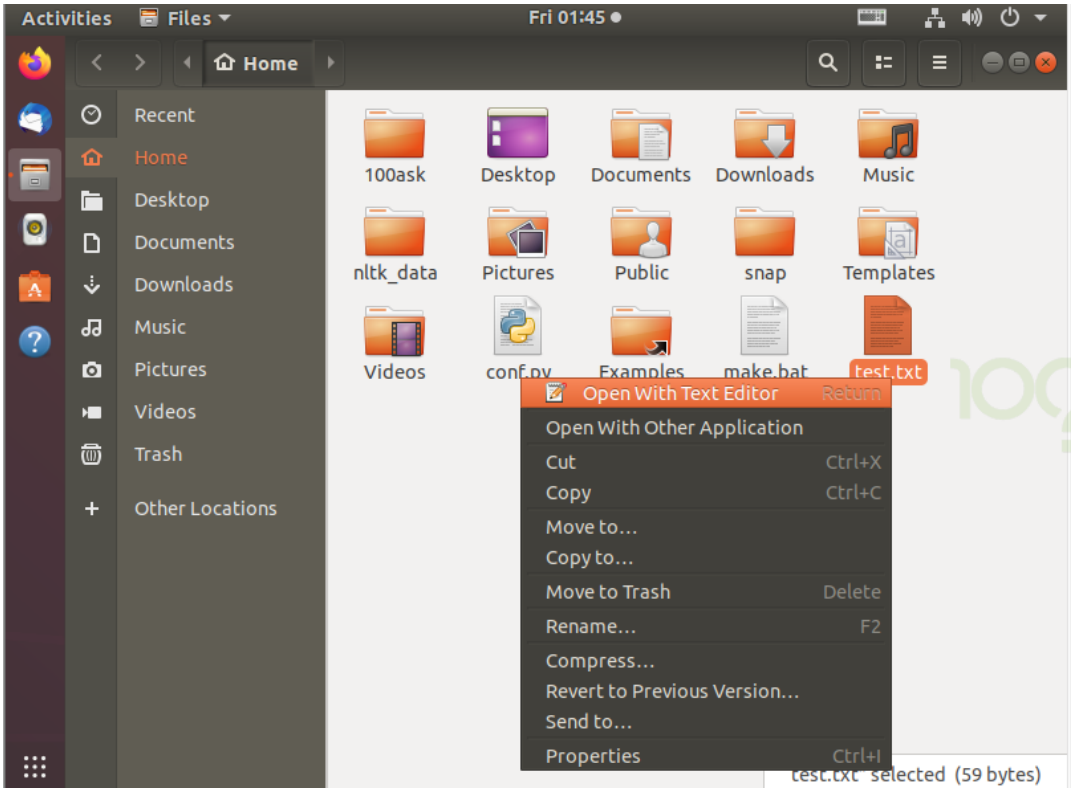


图 2.36

2.5.2 在终端里打开 gedit

在终端里，可以直接运行 gedit 命令打开编辑器，也可以运行“gedit 文件名”打开指定文件，比如：

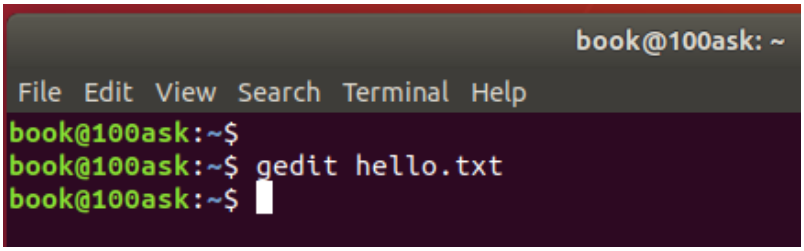


图 2.37

如果要修改其他用户的文件，比如/etc/fstab，如下：

图 2.38

2.5.3 gedit 的使用

gedit 跟 Windows 下记事本的用法没什么差别。

在编辑器中我们可以点击 ”Open” 按钮浏览最近打开过的文件列表并打开文件；点击 ”Save” 按钮可以保存当前正在编辑的文件；点击右侧的菜单栏进

行更多的操作等等。

快捷键也跟 windows 下一样：

- 保存文件：Ctrl + s
- 另存为：Ctrl + Shift + s
- 搜索文本内容： Ctrl + f

2.6 强烈建议使用纯英文的 ubuntu

编译程序时，在纯英文环境下错误信息是以英文来描述的，比如：

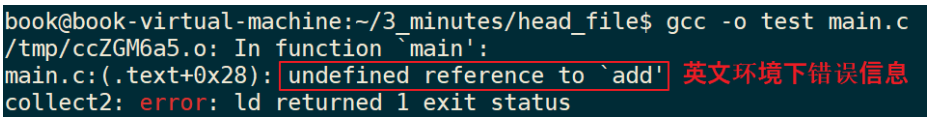


图 2.39

这些错误信息在网上一贴，就有很多答案，百度找不到就用 bing，再不行就用 google，国内找不到可以找国外。

如果使用中文环境的话，错误信息是以中文来描述的，你只能在百度上搜答案。

2.7 默认不能使用 root 用户登录

在开发过程中很少用 root 用户，要使用 root 权限时可以在命令前加上“sudo”，比如“sudo ps -a”。

如果你就是喜欢用 root 用户，可以按下图操作，先给 root 用户设置密码，以后就可以用 root 用户登录了：

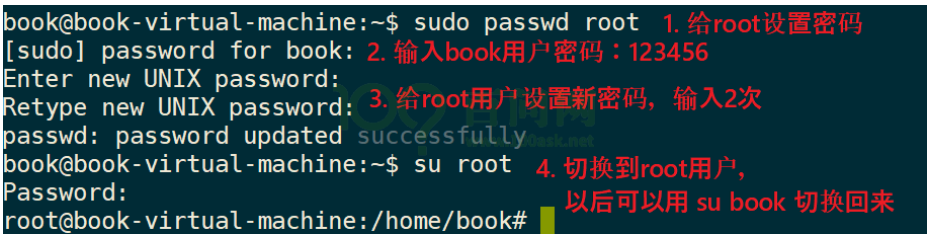


图 2.40

>>>>>>>>第三篇 环境搭建与开发板操作<<<<<<<<

操作视频链接：https://www.bilibili.com/video/BV1zV411U7H9

第1章 配置 VMware 使用双网卡

对于初学者，VMWare 设置为使用双网卡是最方便的：

- NAT 网卡：Ubuntu 通过它上网，只要 Windows 能上网，Ubuntu 就能上网
- 桥接网卡：Ubuntu 通过它跟开发板联通

NAT，Network Address Translation，指网络地址转换。使用 NAT 网卡时，Ubuntu 要访问外网，是委托 Windows 发出数据包，Windows 接收到回应后再转发给 Ubuntu。外界看到的都是 Windows，看不到 Ubuntu。使用 NAT 时，只要 Windows 能上网，Ubuntu 就必定能上网，无需设置 Ubuntu 的网络。

使用桥接网卡时，Ubuntu 就是使用一个真实的网卡：开发板的网线也连接到这个真实的网卡上，这样 Windows、Ubuntu、开发板就都可以用过这个网卡互通了。

使用双网卡时，VMWare 打开的 Ubuntu 虚拟机界面如图 1.1 所示：

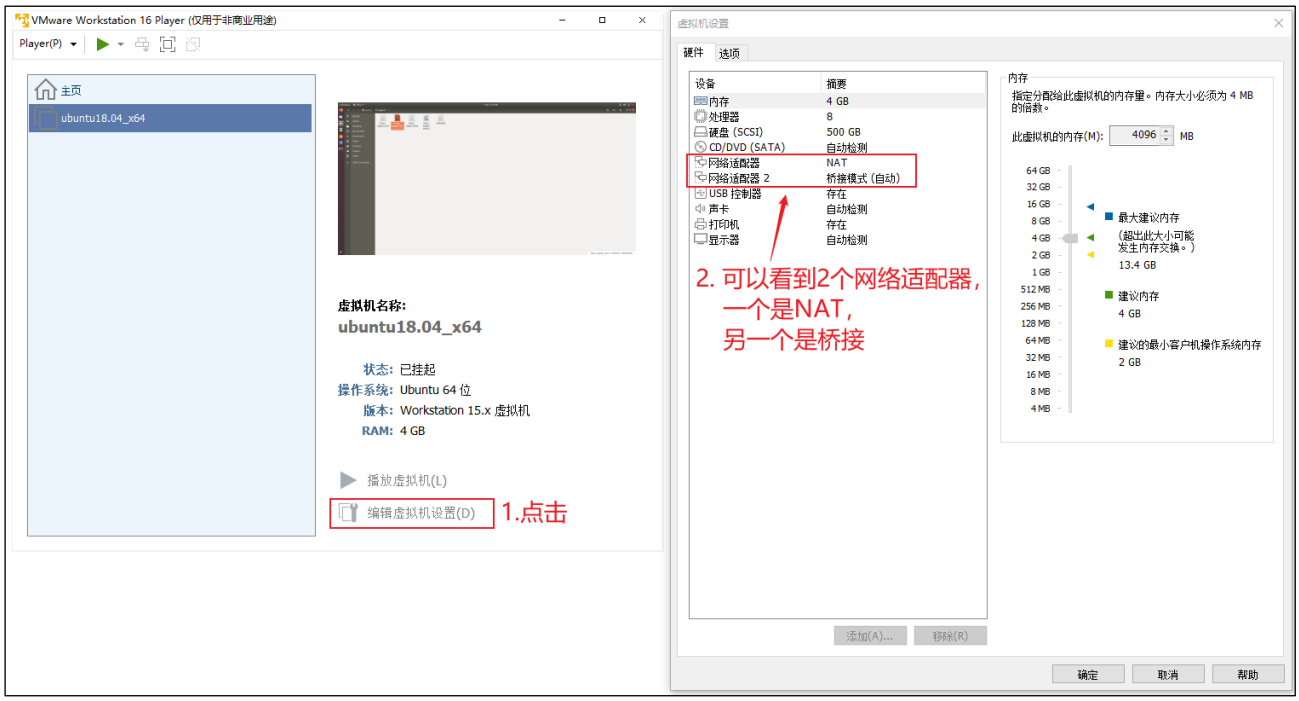


图 1.1 设置双网卡

① 点击进入“编辑虚拟机设置”；
② 查看是否有双网卡，即两个“网络适配器”；

1.1 添加 NAT 网卡

具体操作视频链接：

https://www.bilibili.com/video/BV1zV411U7H9?p=2

使用我们制作的 Ubuntu 映像时，它已经添加了 NAT 网卡，无需再添加 NAT

网卡。如果你的 Ubuntu 虚拟机中没有 NAT 网卡，则可以如图 1.2 所示添加 NAT 网卡：

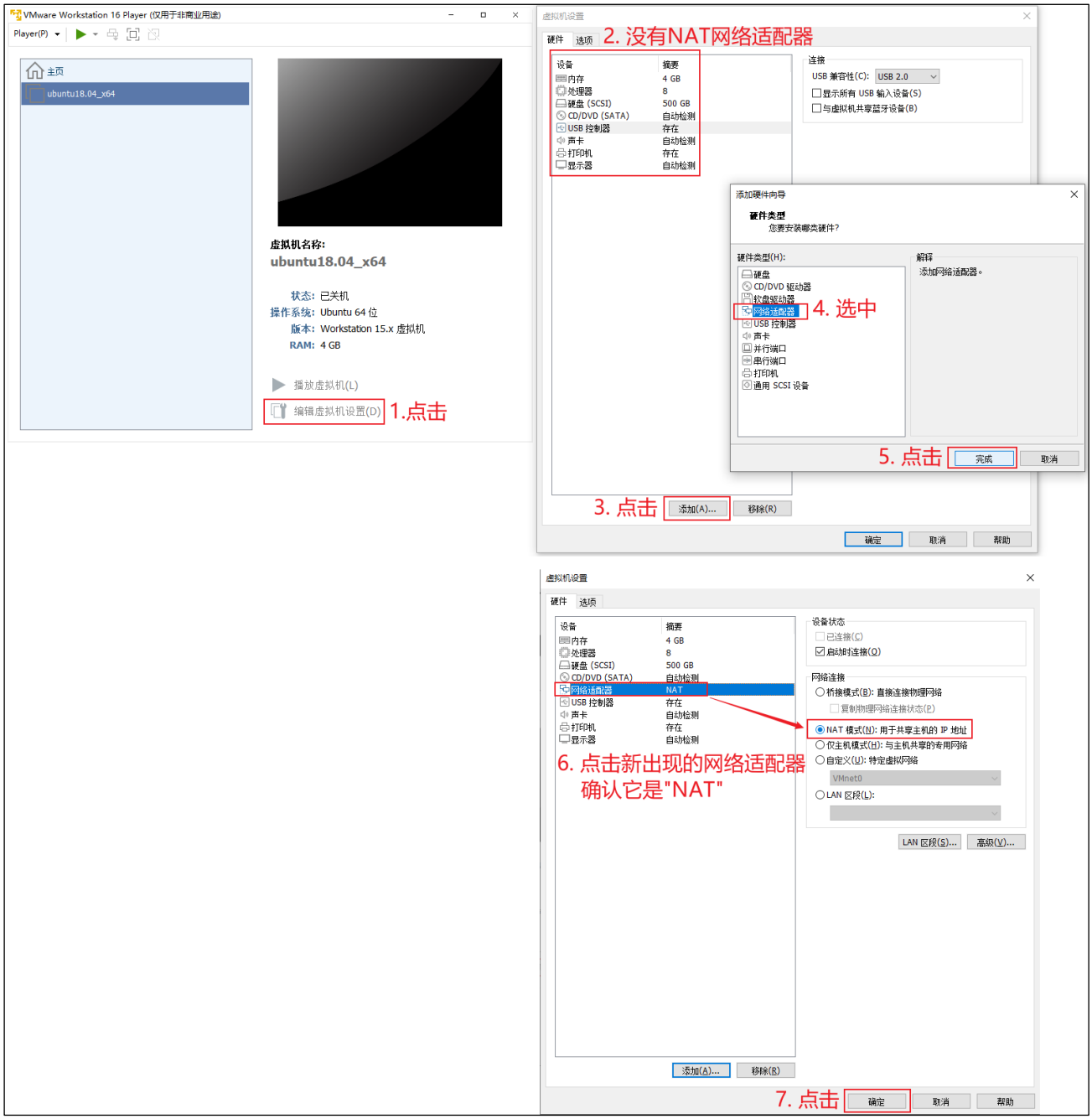


图 1.2 添加 NAT 网卡

① 点击进入“编辑虚拟机设置”；
② 如果没有 NAT 模式的网卡，则继续下一步；
③ 点击“添加”；
④ 选择“网络适配器”；
⑤ 点击“完成”；
⑥ 设置新添加的“网络适配器”的“网络连接”为“NAT 模式”；
⑦ 点击确定完成 NAT 网卡的添加；

添加 NAT 网卡后，可以启动 Ubuntu，使用 ifconfig 命令查看 IP，再使用 ping 命令确认可以连接外网，如图 1.3 所示：

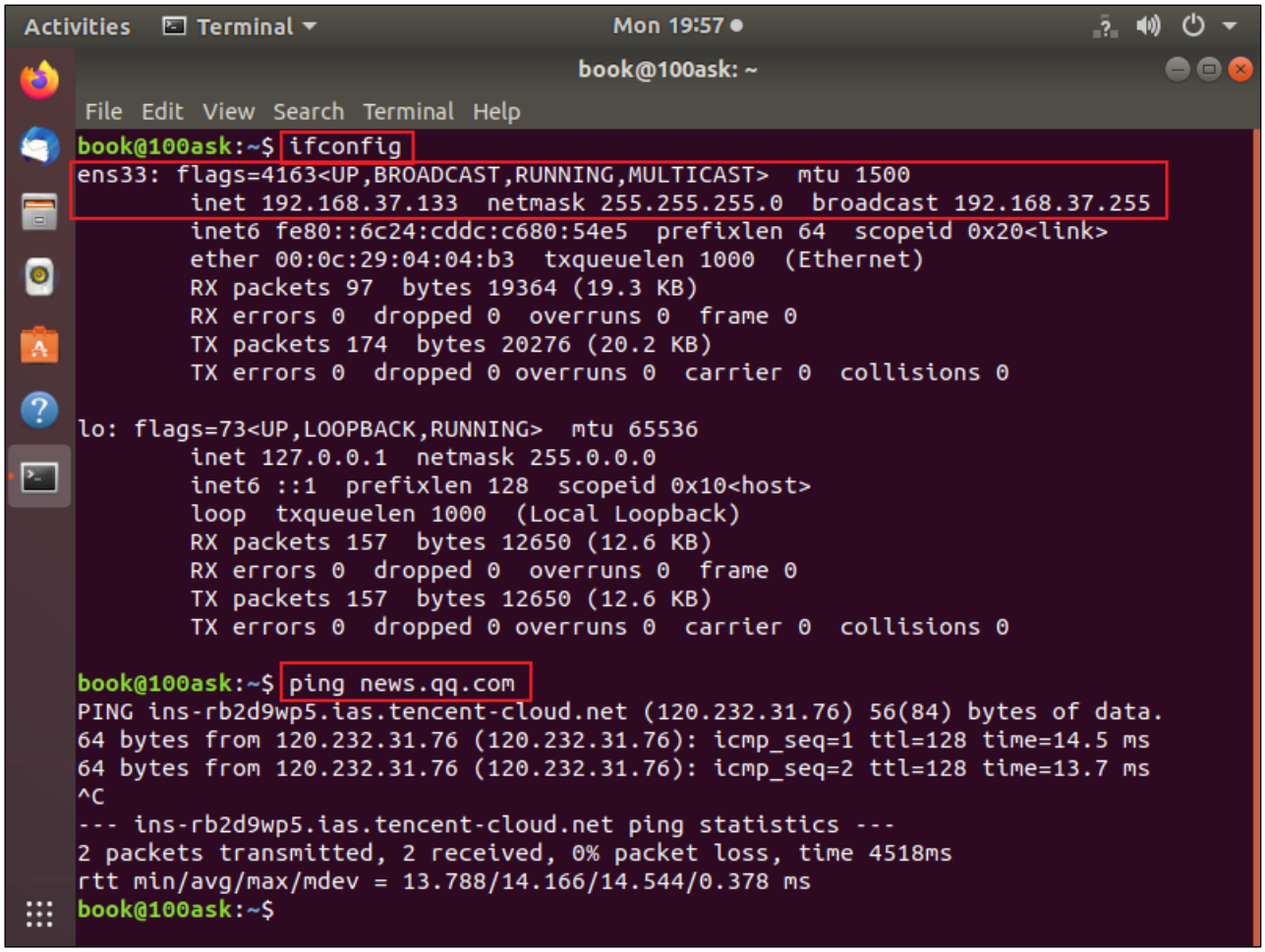


图 1.3 ifconfig

1.2 添加桥接网卡

具体操作视频链接：

https://www.bilibili.com/video/BV1zV411U7H9?p=3

注意：添加桥接网卡时，USB 网卡插上或不插上都可以。

如果你的 Ubuntu 虚拟机中没有桥接网卡，则可以如图 1.4 所示添加：

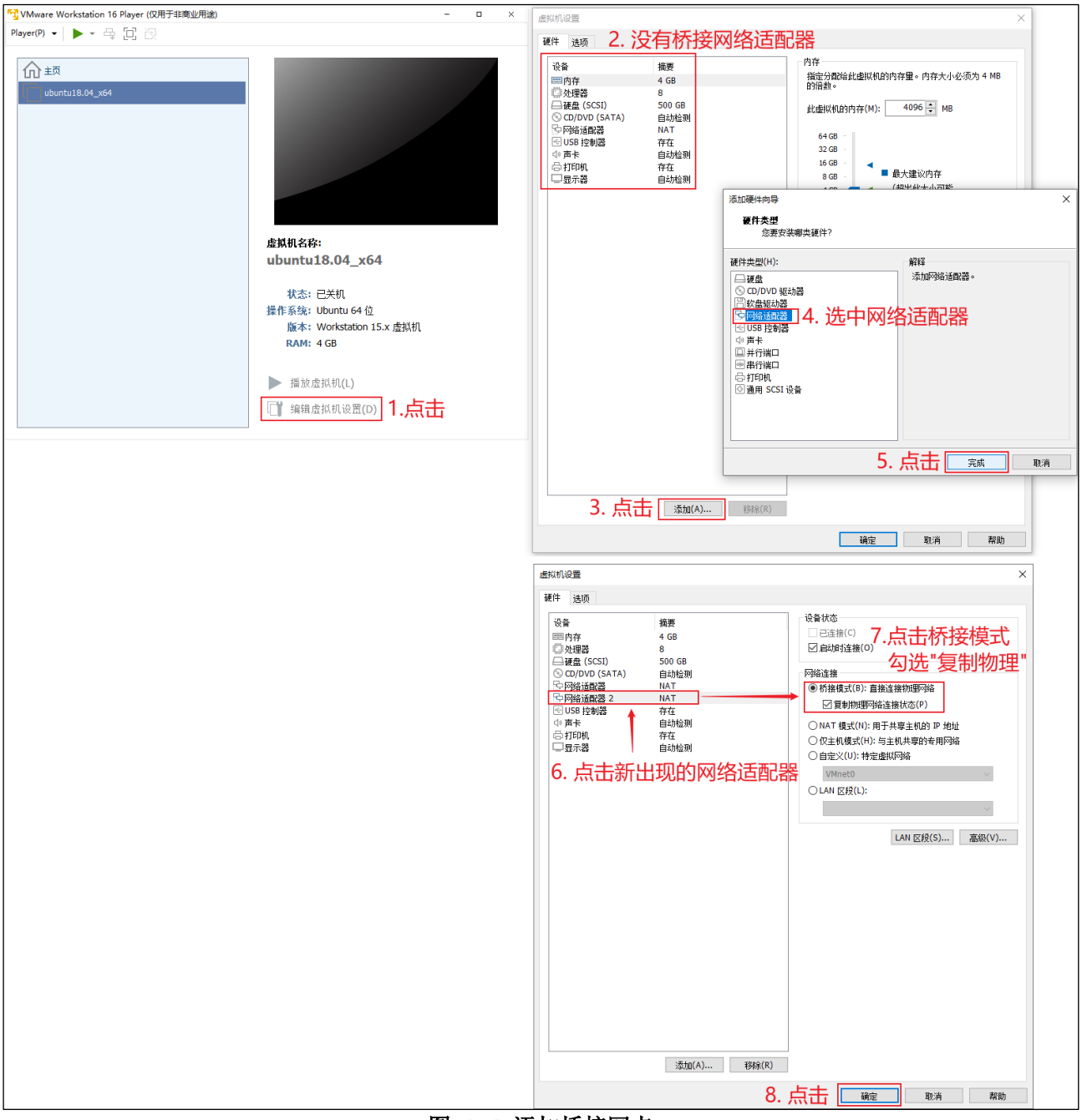


图 1.4 添加桥接网卡

① 点击进入“编辑虚拟机设置”；
② 如果没有桥接模式的网卡，则继续下一步；
③ 点击“添加”；
④ 选择“网络适配器”；

⑤ 点击“完成”；
⑥ 设置新添加的“网络适配器”的“网络连接”为“桥接模式”；
⑦ 点击确定完成桥接网卡的添加；
添加桥接网卡后，还需要进一步配置才能使用。

1.3 配置桥接网卡

VMWare 中使用桥接网卡，是为了跟开发板相连。使用桥接网卡时，必须有真实的网卡。

百问网的开发板，配有一个 USB 有线网卡。目前测试多款网卡，发现只有此款可以满足我们的日常使用需求。

1.3.1 连接网卡

把 USB 网卡连接至电脑 USB 接口，再使用网线连接 USB 网卡和开发板。

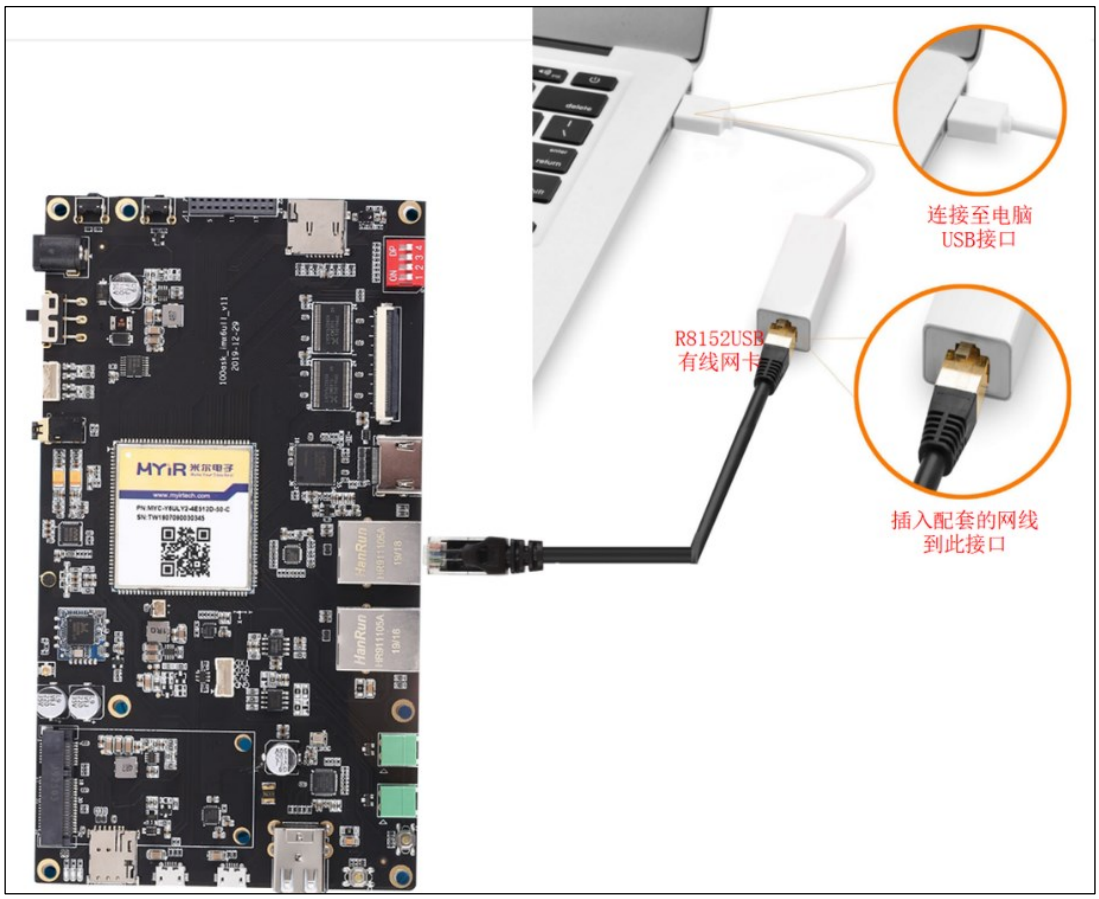


图 1.5 USB 网卡连接开发板和 PC

接好线后，开发板上电，接下来需要设置 IP：Windows 上 USB 网卡、Ubuntu 使用的桥接网卡、开发板的网卡，这 3 个网卡的 IP 要设置为同一个网段。

1.3.2 windows 配置

连接好网线后，在 Windows 设备管理器-->网络适配器下会新增一个“ASIX AX88772C USB 2.0 to Fast Ethernet Adapter”的网络设备，如下图所示：

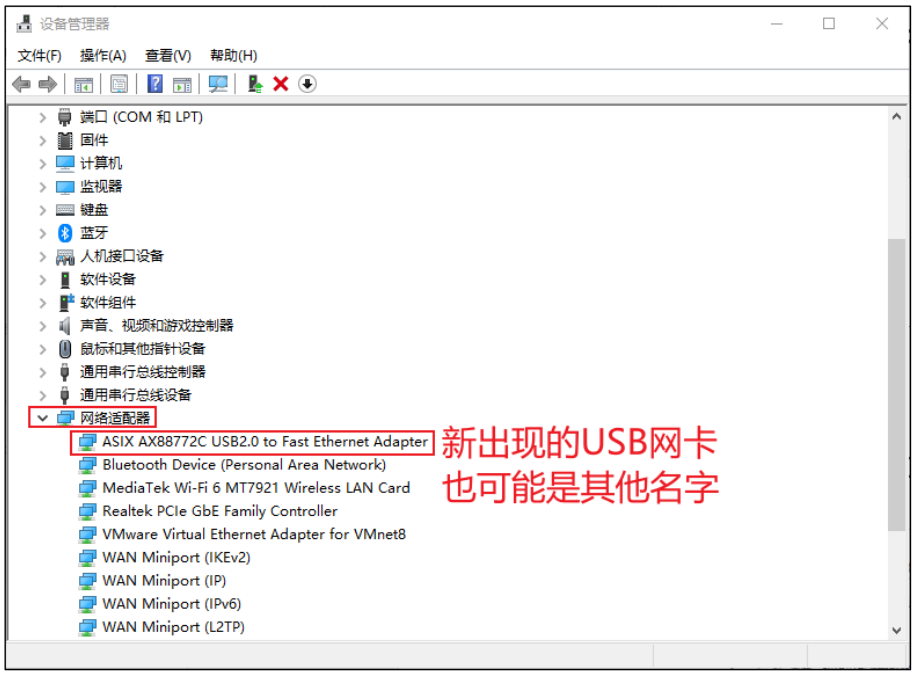


图 1.6 设备管理器查看 USB 网卡

确认 USB 网卡名称后，参考下图。打开“控制面板→网络和 Internet→网络和共享中心→更改适配器设置”，配置 USB 网卡的 IP。

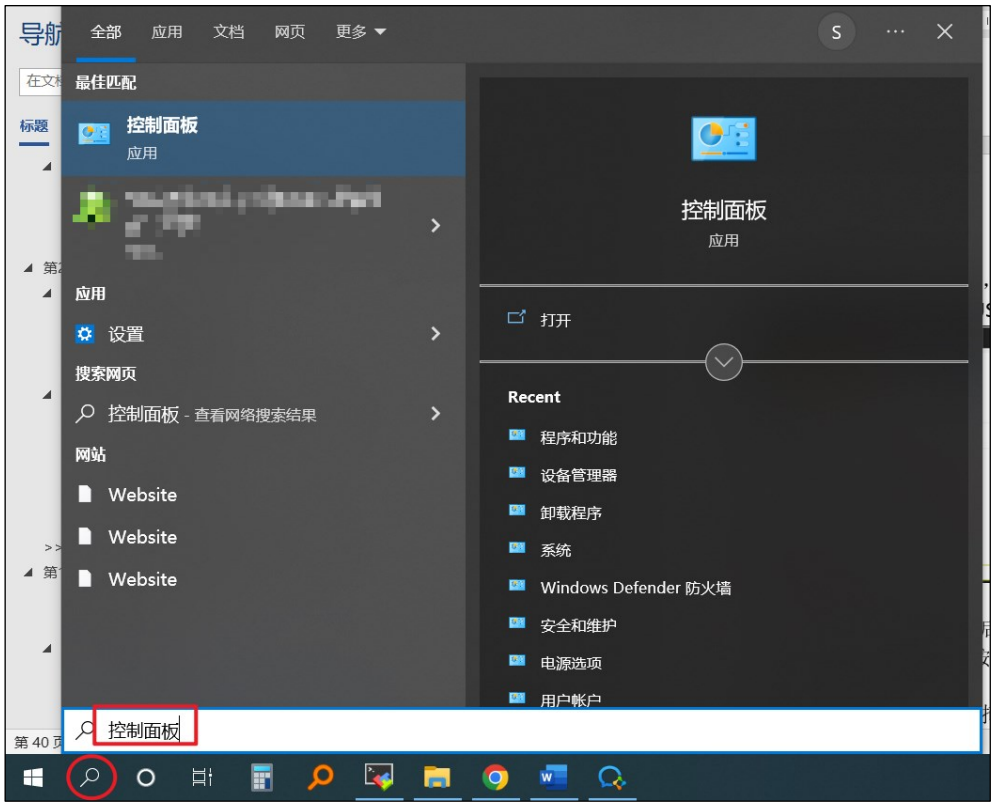


图 1.7 找到“控制面板”并打开

图 1.8 点击点入“网络和 Internet”

图 1.9 点击进入“网络和共享中心”

图 1.10 更改网络适配器设置

进入网络适配器页面后，参考下图，鼠标右键点击 USB 网卡设备，在弹出的选项中点击“属性”按钮：

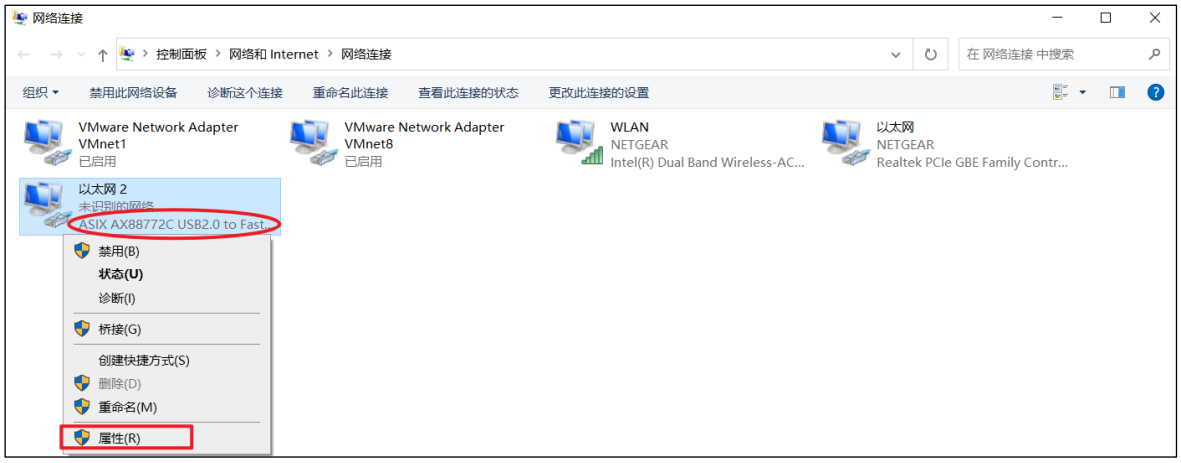


图 1.11 找到 USB 网卡以太网且打开其“属性”

之后在弹出新的属性对话框内点击“Internet 协议版本 4(TCP/IPV4)”：

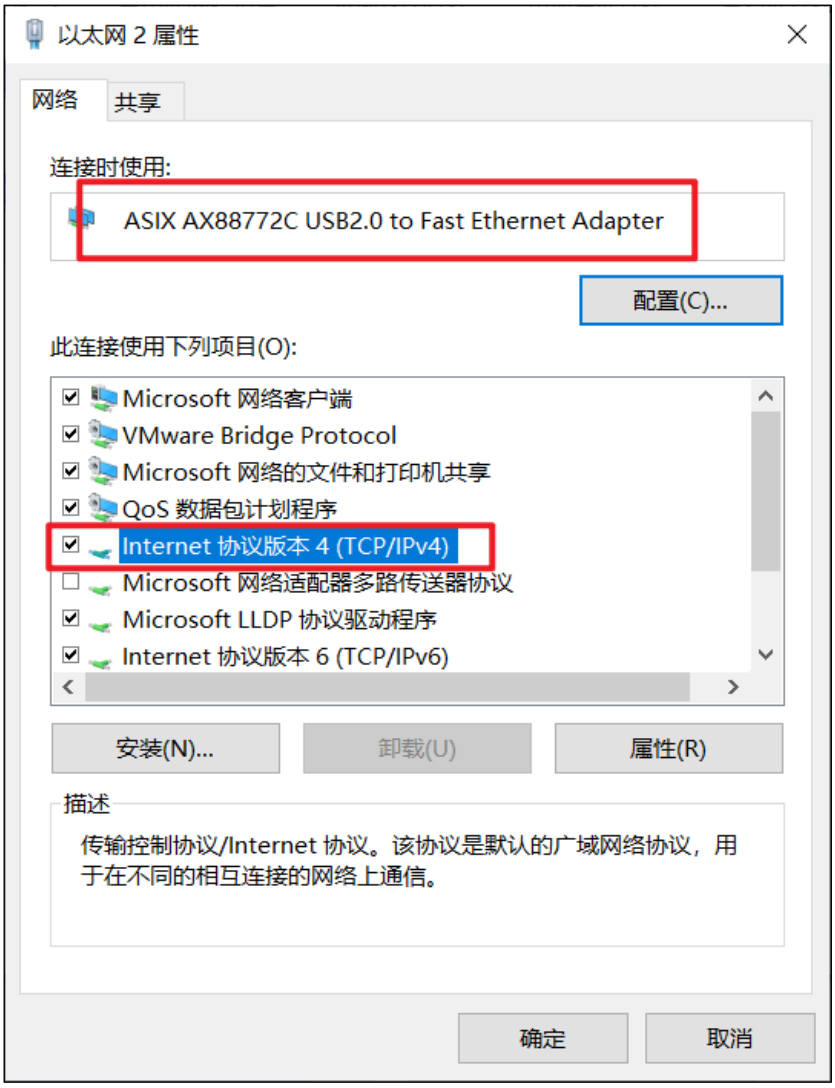


图 1.12 点击进入 USB 网卡的 IPv4 设置

继续在新弹出的对话框参考下图填入 IP 地址“192.168.5.10”、子网掩码

“255.255.255.0”、默认网关“192.168.5.1”，最后点击“确定”。

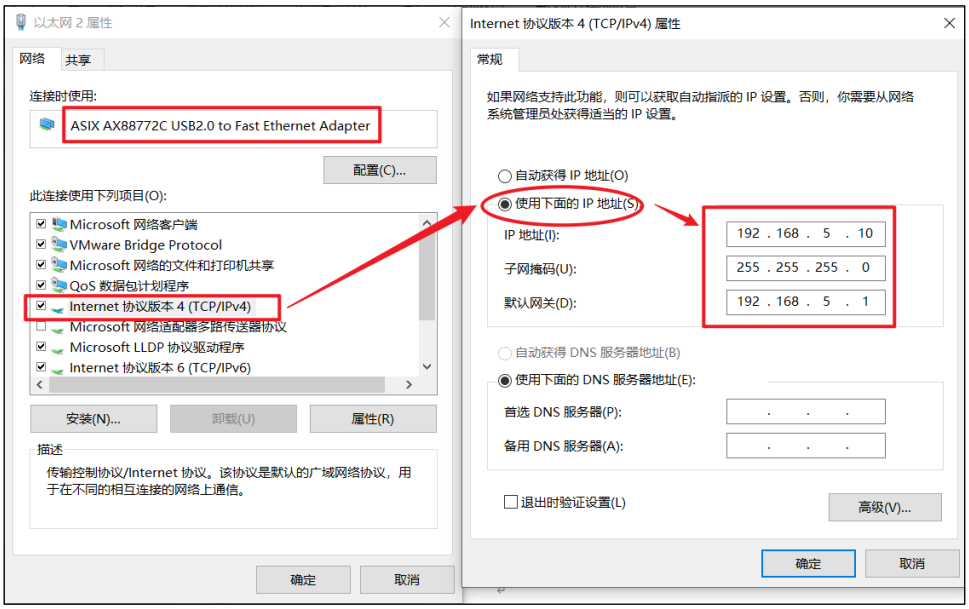


图 1.13 更改 USB 网卡 IP

1.3.3 常见问题

如果在设备管理器里没有看到“ASIX AX88772C USB 2.0 to Fast Ethernet Adapter”或其他名字的 USB 网络设备，有可能是 vmware 接管了这个 USB 网卡。

可以重新拔插 USB 网卡到电脑上，如果 vmware 中有如图 1.14 提示，按图图 1.14 选择“连接到主机”、“记住我的选择，以后不再询问”。

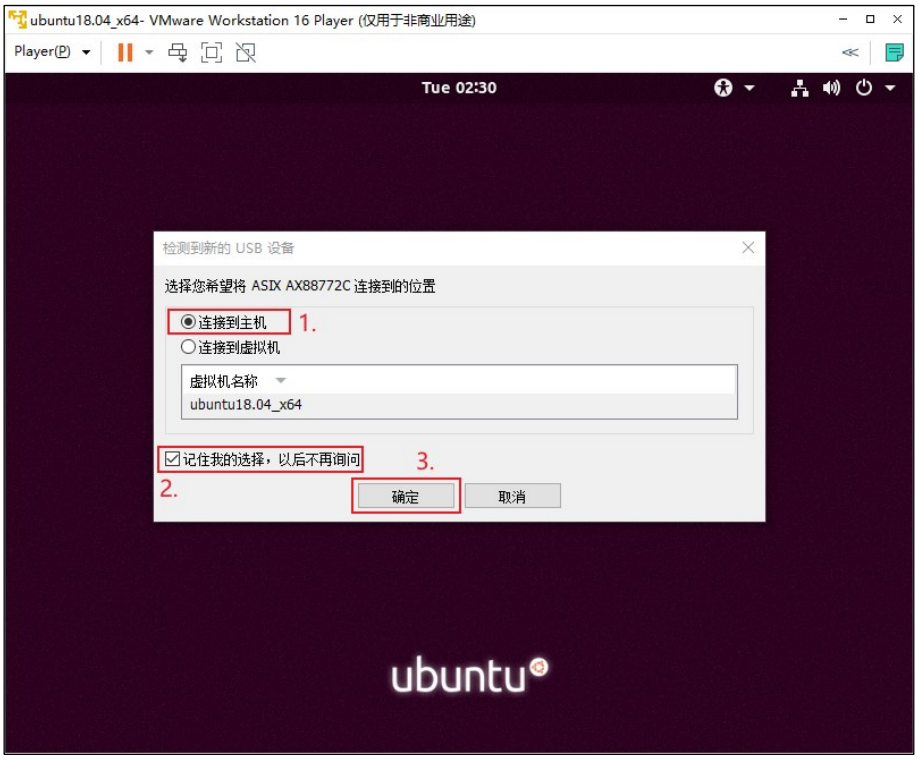


图 1.14 连接 USB 到主机

如果系统没有弹出上图所示窗口，请参考下图图 1.15 查看此 usb 网卡是否已经连接到了 ubuntu。如果它前面已经被打勾，就表示它被连接到 Ubuntu 了，这时点击“断开连接(连接主机)”。

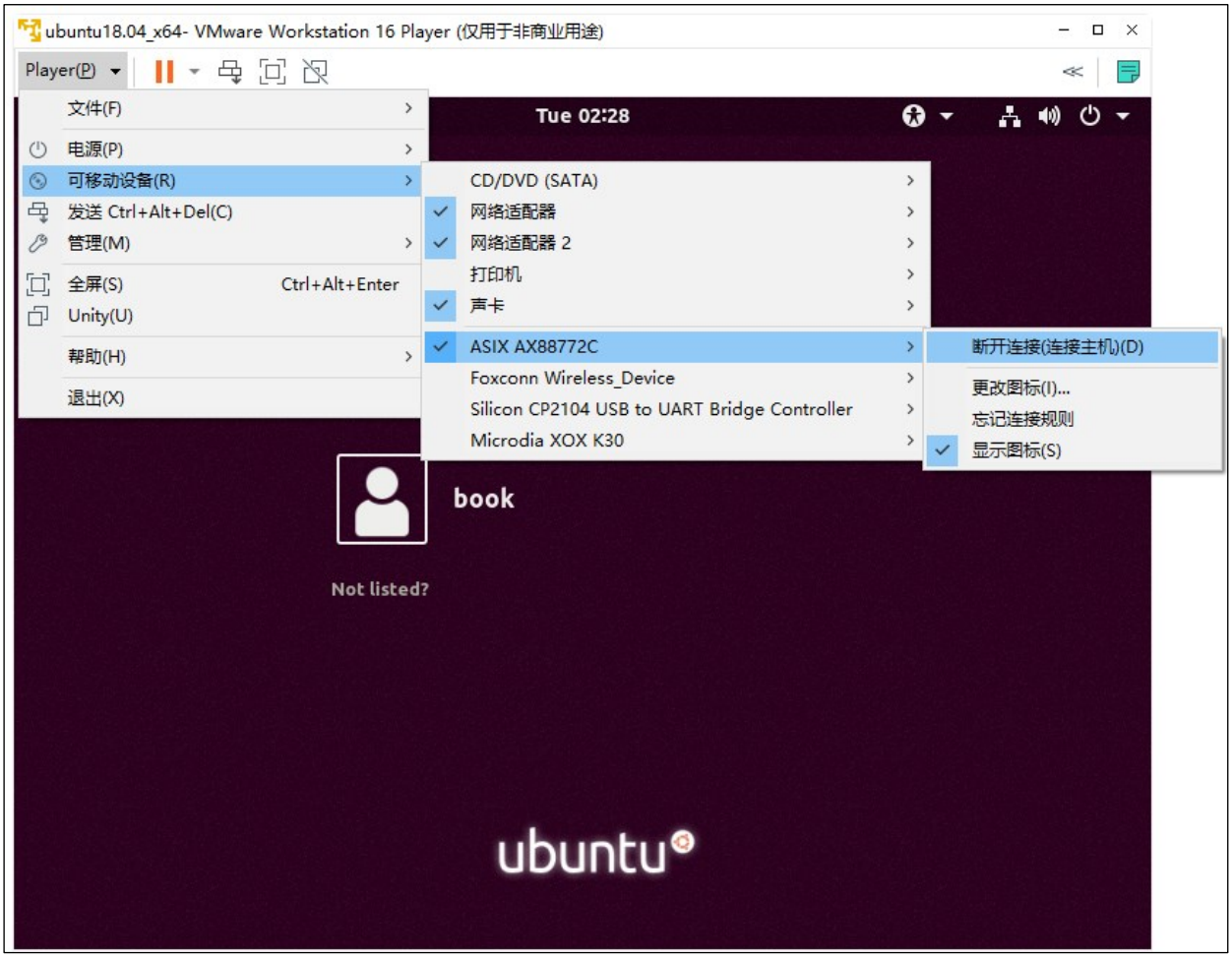


图 1.15 确认 USB 网卡和 Ubuntu 的连接状态

如果没有出现上述两种情况，就是说 vmware 并未接管 USB 网卡，但是在“windows 设备管理-->网络适配器”内依旧没有新增设备，可能是由于驱动问题，请安装相应的设备驱动。

由于此 usb 网卡设备驱动是免驱设备，正常情况系统会自动装载此设备驱动，如没有自动安装驱动，请使用驱动精灵/驱动人生等工具自动安装。

1.3.4 vmware 配置

插上 USB 网卡后，电脑中有多个网卡，使用哪个网卡作为桥接网卡呢？需要在 vmware 中配置，选择 USB 网卡用作桥接网卡；然后才能在 Ubuntu 中设置它的 IP。

在 windows 上设置 USB 网卡的 IP 后，请参考图 1.16 配置 vmware 虚拟网络编辑器：在开始菜单搜索“虚拟网络编辑器”，点击“以管理员身份运行”打开虚拟网络编辑器：

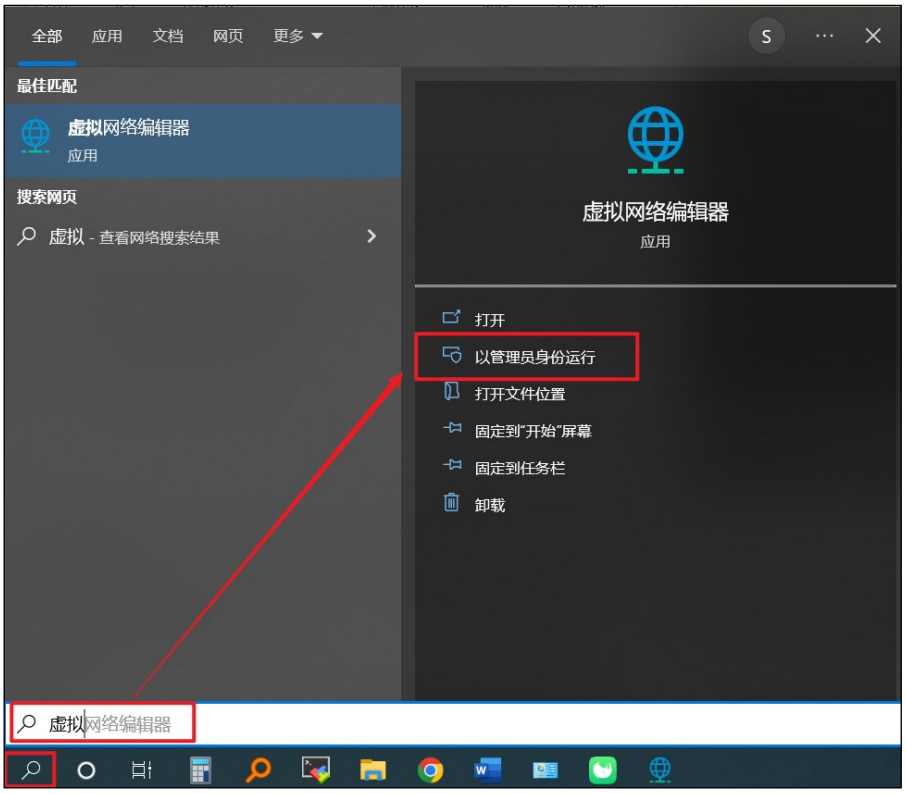


图 1.16 更改虚拟机设置

参考图 1.17，点击“VMnet0”，选择“桥接模式”，在桥接模式下的“已桥接至”下拉框中，选中 USB 网卡(它的名字可以在设备管理器中得到)，最后点击确定即可完成 vmware 配置。

注意：必须是“VMnet0”，如果没有“VMnet0”可以点击“添加网络”。

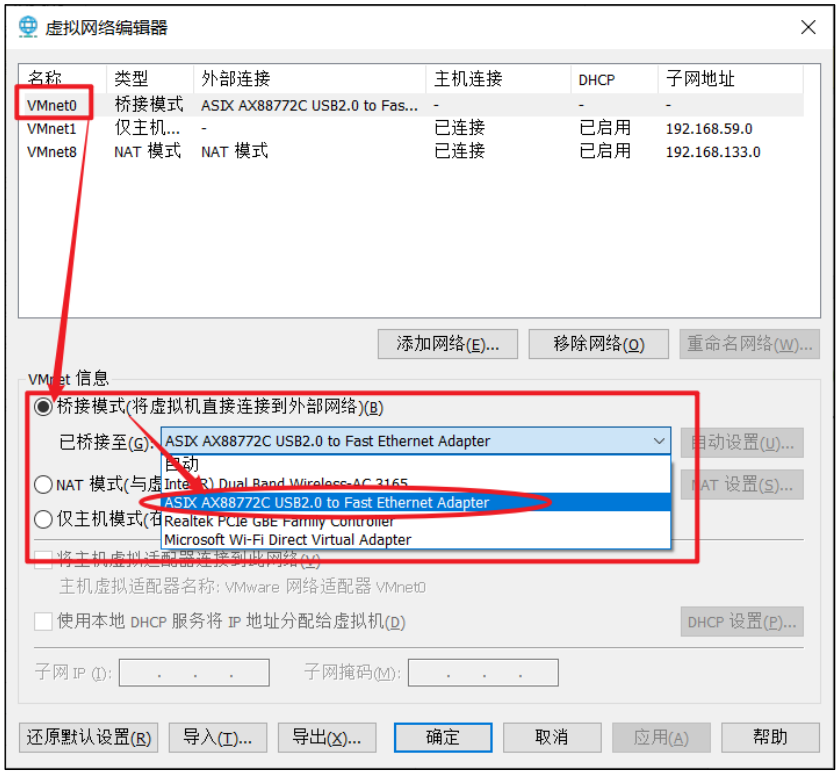


图 1.17 桥接 USB 网卡

1.3.5 ubuntu 配置

在 vmware 中选择 USB 网卡用作桥接网卡后，才能在 Ubuntu 中配置它的 IP。

使用 vmware 开打 ubuntu 虚拟机，在 Ubuntu 关机状态下，点击“编辑虚拟机设置”，在弹出的虚拟机设置对话框，确认有一个“网络适配器”是桥接的，如图 1.18 所示：

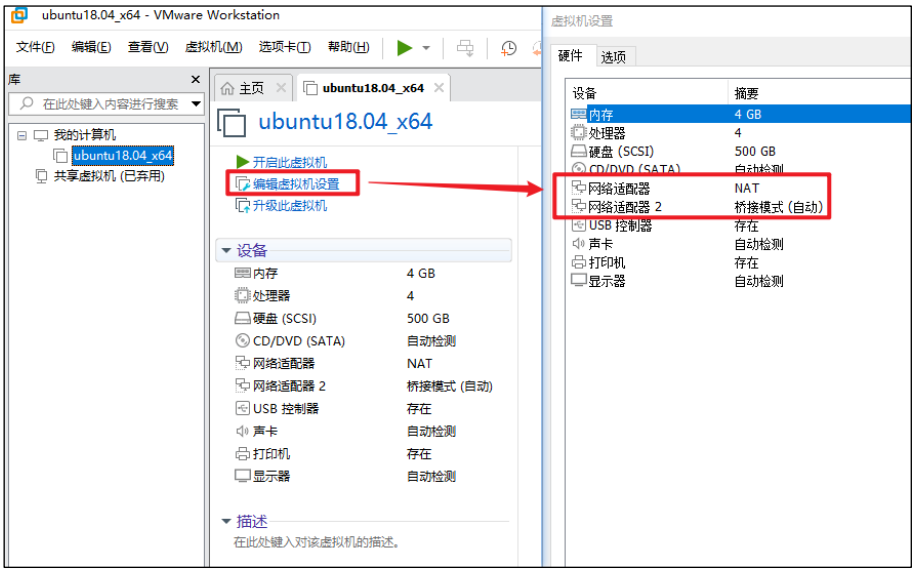


图 1.18 Ubuntu 网络适配器设置

接下来启动 Ubuntu，在 Ubuntu 中设置桥接网卡的 IP 地址为静态 IP，参考：

- 《>>>>>>>>第二篇 2.2.3 设置屏幕图 2.7》所示，打开 系统 **Setting**。

打开 Setting 后，在左侧找到 Network 选择栏，点击显示详细内容，可以看到有 2 个网卡：ens33、ens36。它们对应 NAT 网卡、桥接网卡。我们要设置的是桥接网卡，哪个是桥接网卡？可以点击图 1.19 中的红色箭头图标查看 IP，有 IP 的就是 NAT 网卡，没有 IP 的就是桥接网卡：

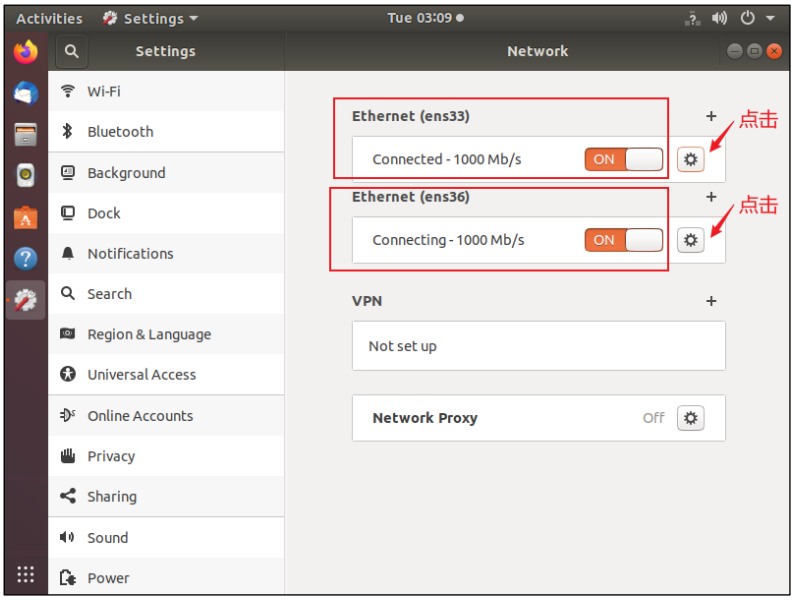


图 1.19 查看 Ubuntu 网络连接

确认 ens36 没有 IP 后，它就是要设置的桥接网卡。点击它右边的“设置”图标，在弹出的设置界面内，点击“IPV4”切换出设置页面，之后选择“Manual”表示手工设置 IP 地址，在“Address、Netmask、Gateway”输入框分别填入：192.168.5.11、255.225.225.0、192.168.5.1。填写完毕后，点击“Apply”，会弹出一个对话框提示输入 root 用户的密码，请参考图 1.20 所示。

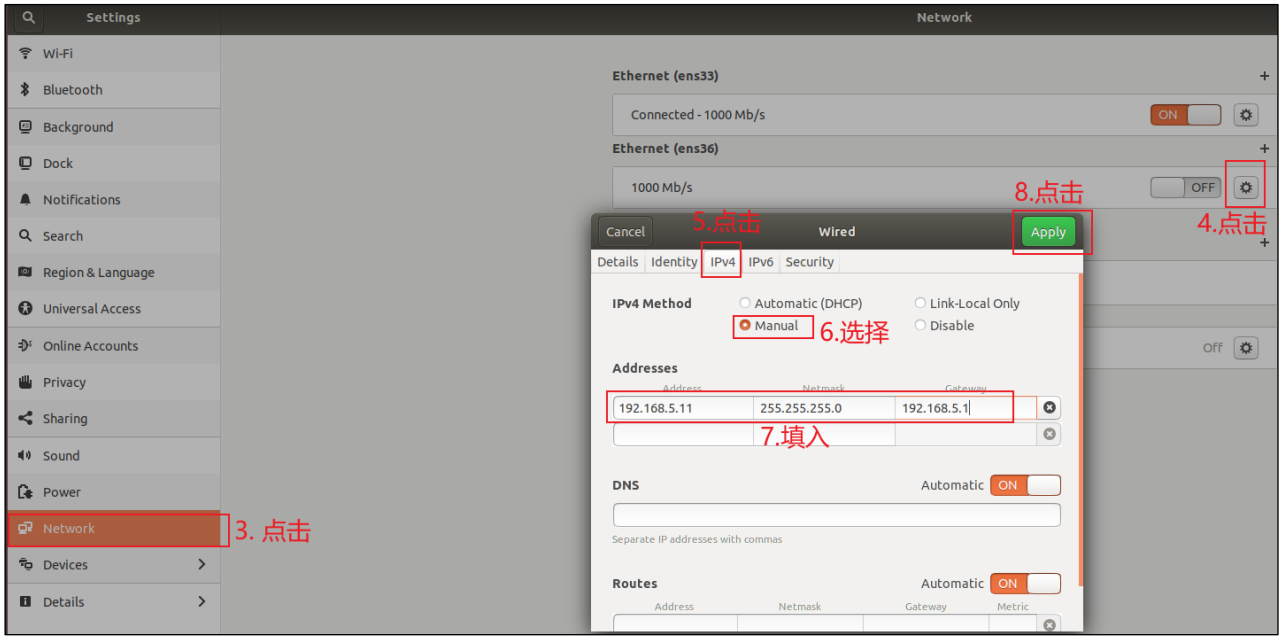


图 1.20 设置 Ubuntu 的静态 IP

在弹出的授权请求对话框里面输入 root 用户的密码，后点击“Authenticate”授权，设置完毕。

注意：如果未设置 root 用户密码，请在 ubuntu 终端下执行“sudo passwd root”命令来设置 root 用户密码，然后重新设置网络。

我们可以在 ubuntu 终端下输入 ifconfig 命令来查看 IP 地址是否设置正确，如图 1.21 所示。

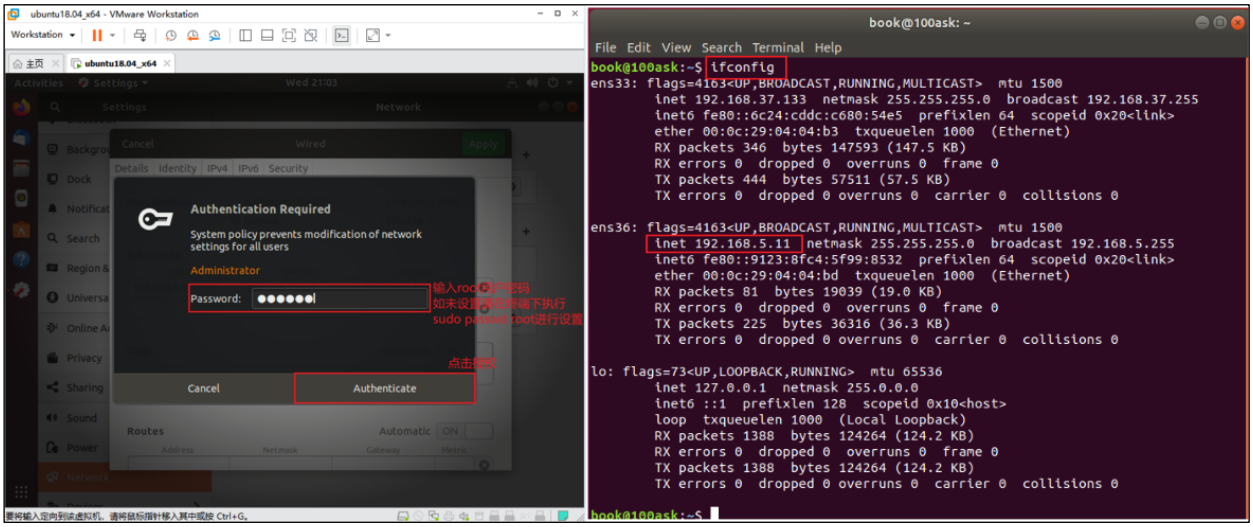


图 1.21 查看 Ubuntu 的 IP

1.3.6 开发板设置 IP 地址

配置开发板 IP 需要使用串口，如果你还不会连接串口，可以先跳过本节。

IMX6ULL 有两个网卡(如下图所示)，推荐使用 eth0，使用网线连接 eth0 和 USB 网卡：

图 1.22 开发板上的网口

设置开发板的 IP 有两种方法：手工设置 IP，修改配置文件设置 IP。

手工设置的方法很简单，但是每次启动开发板都要重新设置，在开发板串口中执行命令即可：

```
ifconfig eth0 192.168.5.9
```

设置成功后可以使用 ifconfig 命令来查看已设置的 IP 地址，参考如图 1.23 所示命令。

```
[root@100ask:~]# ifconfig eth0 192.168.5.9
[root@100ask:~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.9  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::201:1fff:fe2d:3e4d  prefixlen 64  scopeid 0x20<link>
        ether 00:01:1f:2d:3e:4d  txqueuelen 1000  (Ethernet)
        RX packets 290  bytes 18345 (17.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 58  bytes 4488 (4.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 51

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2  bytes 196 (196.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 196 (196.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

usb0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether a2:25:50:e7:47:95  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@100ask:~]#
```

图 1.23 手动设置开发板 IP

修改配置文件设置 IP，修改一次即可，无需重复配置。修改开发板**/etc/network/**目录下的 interfaces 文件：

```
[root@100ask:~]# vi /etc/network/interfaces
```

修改为如下内容：

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
    address 192.168.5.9
    netmask 255.255.255.0
    gateway 192.168.5.1
```

- 按下键盘 ESC，然后输入执行 `:wq` 保存并退出，
- 执行`/etc/init.d/S40network restart` 重启网络服务。

- 使用 **cat** 命令查验修改

```
[root@100ask:~]#  vi /etc/network/interfaces
[root@100ask:~]# cat /etc/network/interfaces
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
        address 192.168.5.9
        netmask 255.255.255.0
        gateway 192.168.5.1
[root@100ask:~]#
```

图 1.24 cat 指令验证 IP 修改

重启开发板后使用 **ifconfig** 查看 IP 地址是否已经自动配置：

```
[root@100ask:~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:01:3F:2D:3E:4D
          inet addr:192.168.5.9  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 00:01:1F:2D:3E:4D
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:161 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11828 (11.5 KiB)  TX bytes:10344 (10.1 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:384 errors:0 dropped:0 overruns:0 frame:0
          TX packets:384 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:28416 (27.7 KiB)  TX bytes:28416 (27.7 KiB)

wlan0     Link encap:Ethernet  HWaddr 38:01:46:DB:EA:6E
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:6 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

wlan1     Link encap:Ethernet  HWaddr 3A:01:46:DB:EA:6E
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:6 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

[root@100ask:~]#
```

图 1.25 ifconfig 查询 IP

1.4 三者互 ping 验证

设置完上述 IP 地址后，我们知道：

```
windows ip：192.168.5.10
ubuntu  ip：192.168.5.11
开发板  ip：192.168.5.9
```

接下来验证三者是否可以互相网络通信。

1.4.1 windows ping ubuntu

在 windows 下打开命令提示符，输入以下命令 ping Ubuntu：

```
ping 192.168.5.11
```

同样的，在 ubuntu 的终端里输入以下命令 ping windows：

```
ping 192.168.5.10
```

正常情况如图 1.26 所示，如果你在测试时发现只能单向 Ping 通，请检查 windows 防火墙是否全部关闭。

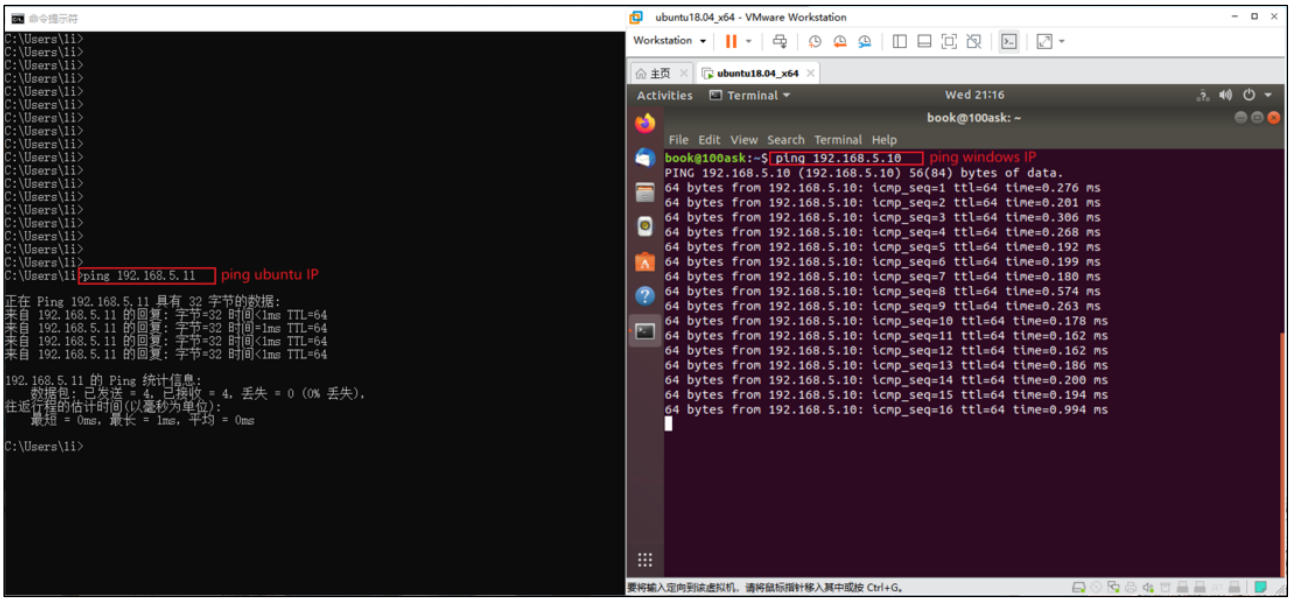


图 1.26 Windows ping Ubuntu

1.4.2 开发板 ping windows 和 ubuntu

如下所示在开发板手动设置 ip 地址为 192.168.5.9 之后，使用 ping 命令来验证是否可以 ping 通 ubuntu 和 windows 主机：

```
ping 192.168.5.10
ping 192.168.5.11
```

正确情况如下图所示：

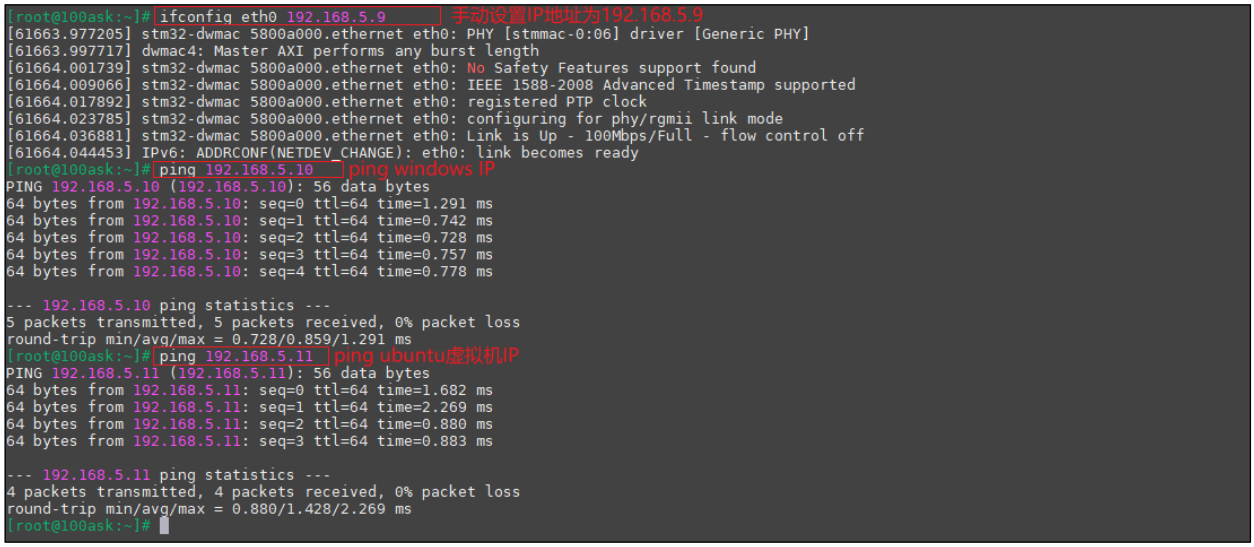


图 1.27 开发板 ping Windows

1.4.3 windows 和 ubuntu ping 开发板

如图 1.28 所示为 windwos 和 ubuntu 去 ping 开发板 IP，在 windows 下使用 命令提示符，执行 `ping 192.168.5.9` 去 ping 开发板，来确认是否可以和开发板网络通信，在 ubuntu 使用终端，执行 `ping 192.168.5.9` 去 ping 开发板来确认是否可以和开发板网络通信。

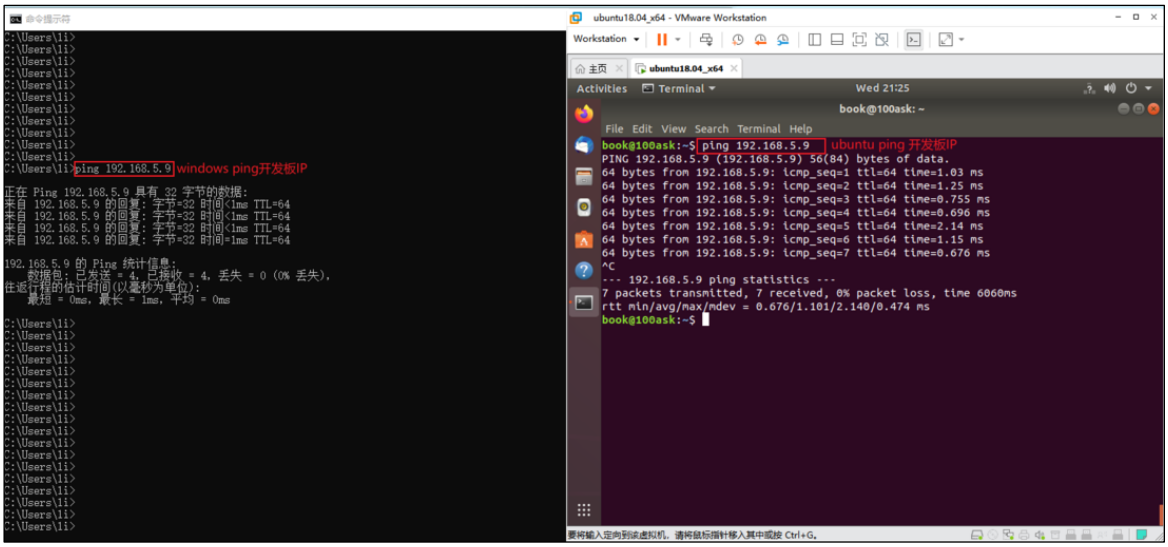


图 1.28 Windows Ubuntu ping 开发板

1.5 开发板上网

我们的开发板一般有两个网口，其中一个被我们用来和 usb 网卡连接，作为调试使用。

开发板上另一个网口可以用来上网：使用网线连接至可以上网的路由器，然后在开发板终端上执行“`udhcpc -i eth1`”，等待 eth1 网卡获得 IP 后，就可以访问外网了。

第2章 安装软件

2.1 安装 Windows 软件

“<开发板配套资料>\02_开发工具\”

在网盘资料包的上述目录中,可以得到一系列的安装软件，建议全部安装。有如下软件：

表 2-1 Windows 中需要安装的软件

软件名	说明
vmware	虚拟机软件，安装时需要用到管理员权限，前面已经安装过
Source insight	阅读、编写源码的工具，即装即用；推荐初学者使用
Visual Studio Code	阅读、编写源码的工具，需要进行很多配置；不推荐初学者使用
MobaXterm	串口工具、远程登录工具
Filezilla	文件传输工具，在 Windows 和 Ubuntu 之间传输文件
Notepad++	文本编辑工具，比记事本好用

Visual Studio Code 的配置比较麻烦，建议初学者使用 Source insight 来阅读、编写源码。

2.2 安装 Ubuntu 软件

确保 Ubuntu 能上网之后,使用下面命令一键配置/初始化开发环境(其实就是安装 tftp，nfs，vim 等软件，此脚本只支持 Ubuntu-16.04 /Ubuntu-18.04)。

```
book@100ask:~$ git clone https://e.coding.net/weidongshan/DevelopmentEnvConf.git
book@100ask:~$ cd DevelopmentEnvConf
book@100ask:~$ sudo ./Configuring_ubuntu.sh
```

上述命令是下载脚本，给它添加可执行权限，运行它。按提示输入 book 密码 123456 和选择对应的系统，如图 2.1 所示：

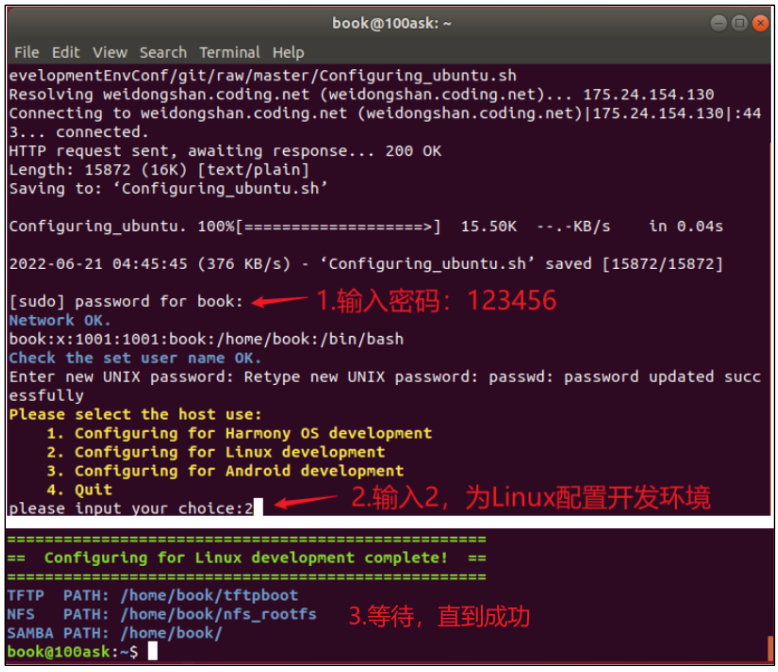


图 2.1 执行配置环境脚本命令

注意：如果 Ubuntu 无法上网，请参考前面《1.1 添加 NAT 网卡》进行设置。

如果执行该命令出现如图 2.2 所示的错误：

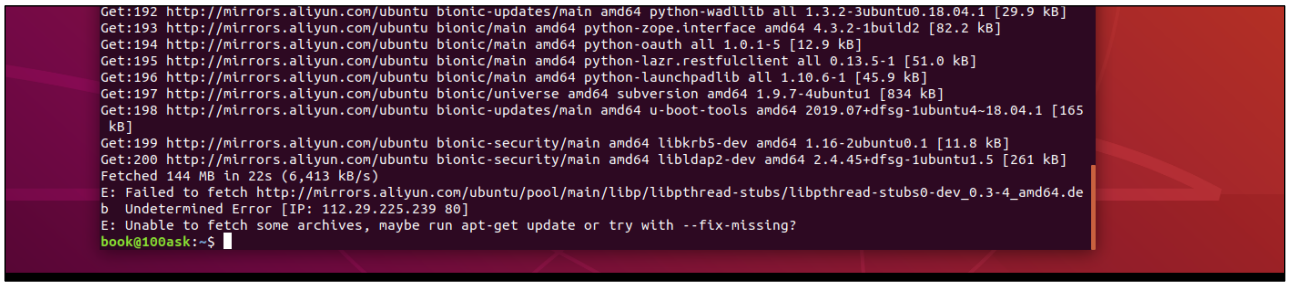


图 2.2 执行脚本错误

可以先执行下面的命令后再重新执行前面的命令：

```
sudo apt-get update
```

2.3 使用 MobaXterm 远程登录 Ubuntu

先确认 Ubuntu 的 IP，可以使用它的 NAT 网卡 IP，也可以使用它的桥接网卡 IP。建议使用 NAT 网卡 IP，因为使用桥接网卡的话必须启动开发板。

在 Ubuntu 终端执行 ifconfig 命令确定 NAT 网卡 IP(注意：这个 IP 过一段时间会发生变化，那就使用新 IP 重新连接)，如图 2.3 所示：

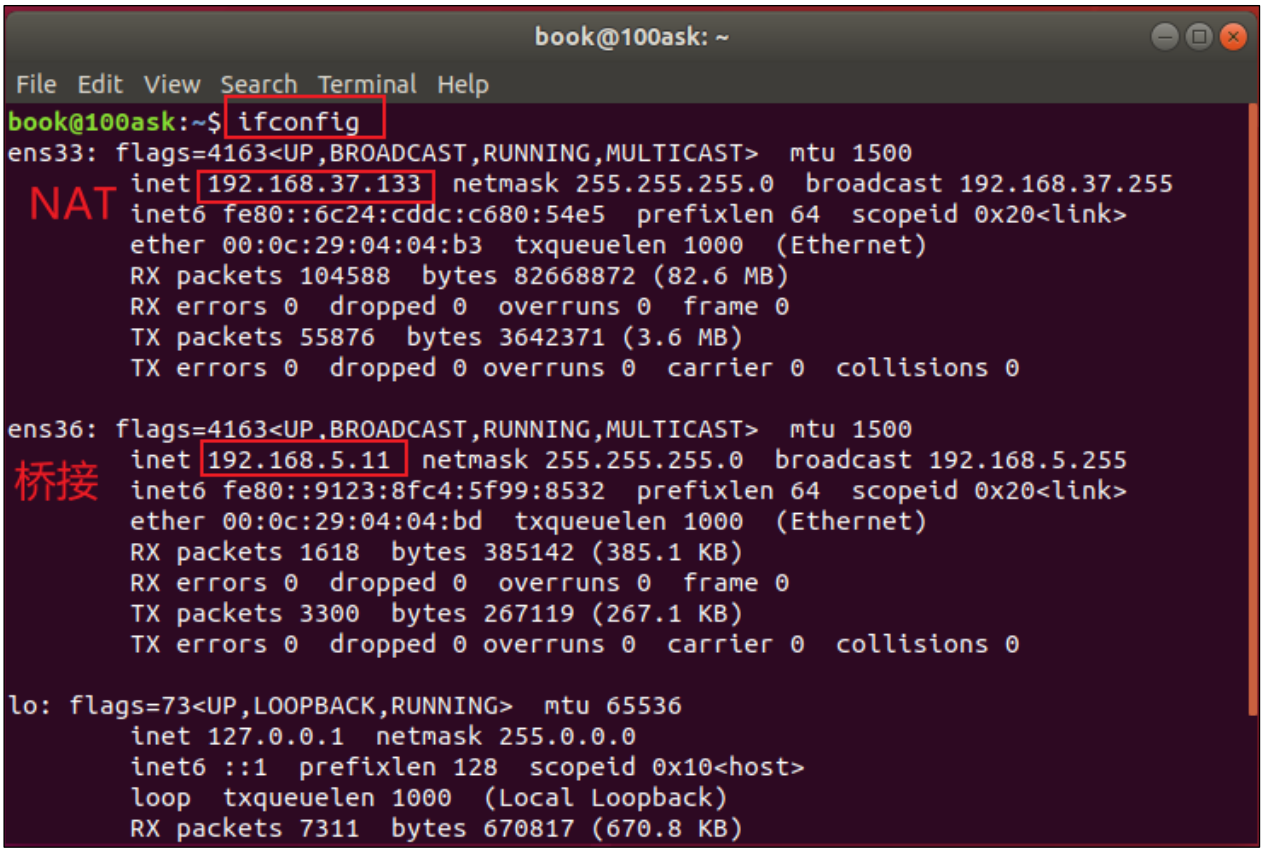


图 2.3 ifconfig 查看 NAT 网卡 IP

安装、运行 MobaXterm，如下建立 Session：

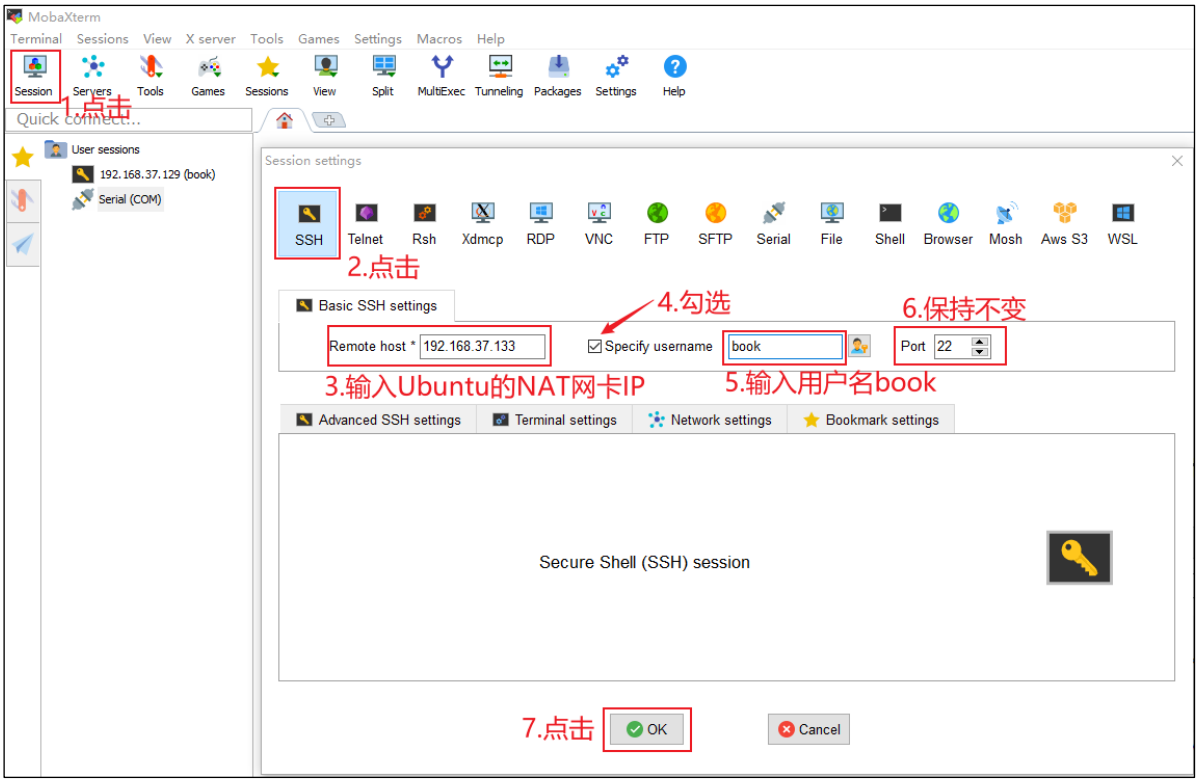


图 2.4 建立 ssh

按图 2.4 操作后，在 MobaXterm 左侧就可以看到图 2.5 这项，双击它就可以登录 Ubuntu(第 1 次登录时会提示你输入密码，密码是 123456)，然后就可以执行各种 Linux 命令了：

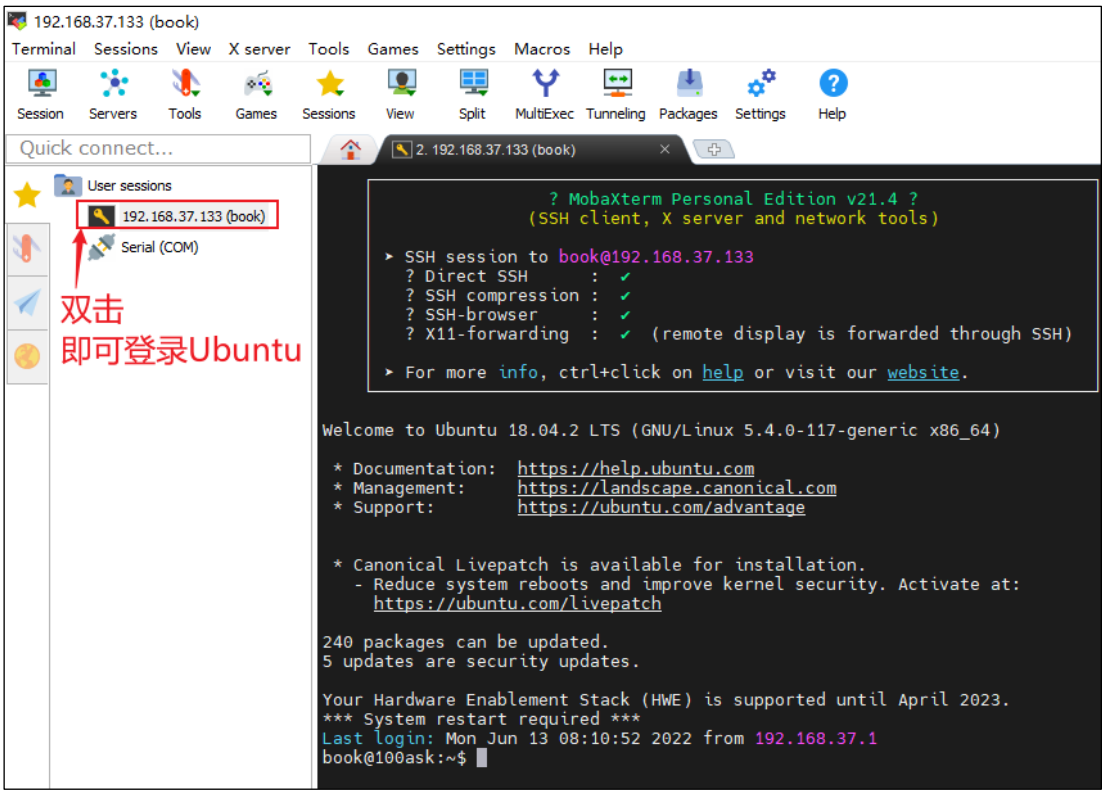


图 2.5 Mobaxterm 登陆上了 Ubuntu

2.4 使用 FileZilla 在 Windows 和 Ubuntu 之间传文件

使用 MobaXterm 既可以 ssh 登录又可以传输文件，不过 Mobaxterm 在传输文件时使用效率上没有 FileZilla 高，所以我们推荐 Windows 和 Ubuntu 互相传输文件时使用 FileZilla。

双击打开 FileZilla 后，按图 2.6 操作：

图 2.6 FileZilla 连接 Windows 和 Ubuntu

第 1 次连接时，会有如图 2.7 所示的提示，选择“总是信任”：

图 2.7 FileZilla 建立连接

在 Filezilla 中，左边是 Windows 文件，右边是 Ubuntu 的文件，如图 2.8：

图 2.8 FileZilla 下 Windows 和 Ubuntu 互传文件

2.5 编程示例：Ubuntu 上的 Hello 程序

本节演示如何在 Windows 编写程序、上传到 Ubuntu，在 Ubuntu 中编译、执行。只涉及一个简单的 Hello 程序，使用命令行编译，不涉及 Makefile 等知识，这些知识在后面的应用基础中讲解。

2.5.1 用 Source Insight 编写 hello.c

启动 Source Insight，点击“File”->“New”，新建文件，如图 2.9：

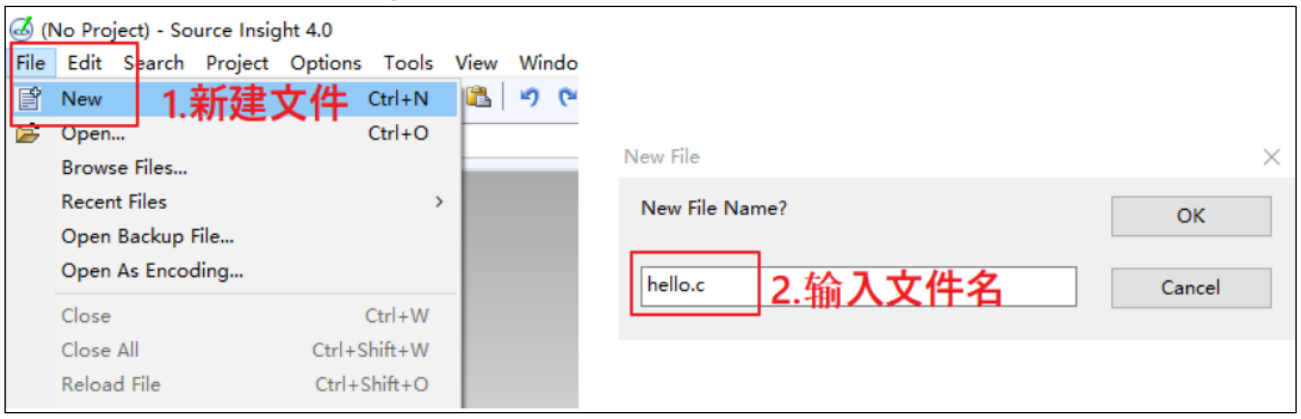


图 2.9 SI 新建文件

接下来编写代码，保存文件，如图 2.10 所示：

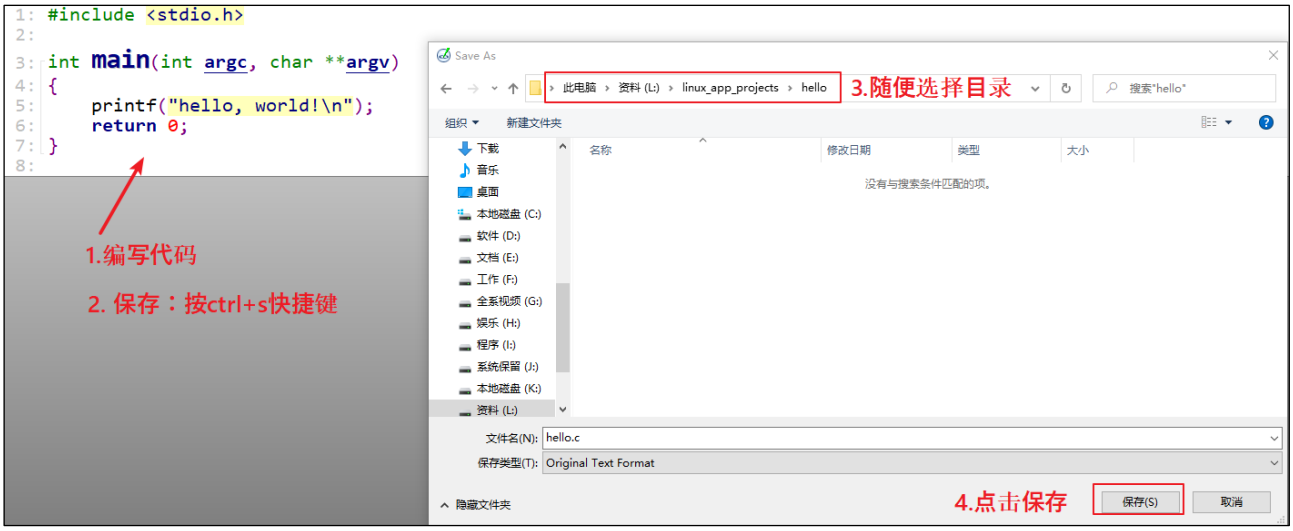


图 2.10 SI 编写代码并保存

hello.c 的源码如下：

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("hello, world!\n");
        return 0;
}
```

2.5.2 使用 FileZilla 上传源码

如图 2.11 操作：

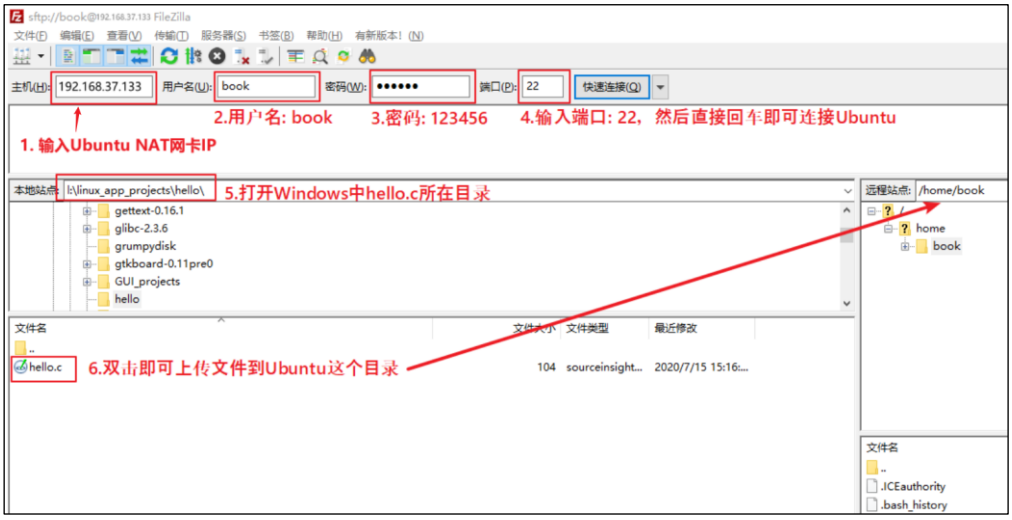


图 2.11 从 windows 传文件到 Ubuntu

2.5.3 编译、运行程序

如图 2.12 操作，对于 gcc 命令的用法在后面讲到应用开发基础时再细讲，这里只是体验一下：

图 2.12 体验 Hello World!

2.6 下载 BSP 及配置工具链

100ask_imx6ull 开发板的 BSP 都保存在 Git 站点上，通过 repo 命令进行统一管理。

2.6.1 配置 repo

下载 repo 工具前需要设置 git 的邮箱和用户名，git 邮箱和用户名请根据个人情况进行配置。

```
book@100ask:~$ git config --global user.email "user@100ask.com"
book@100ask:~$ git config --global user.name "100ask"
```

注意：请先配置 git 邮箱和用户名，否则会导致下载失败(图 2.13 为参考示例图)。

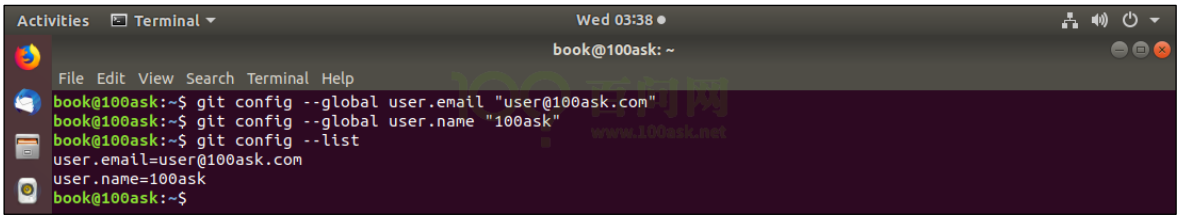


图 2.13 配置本地 git 信息

2.6.2 下载 BSP

通过 repo 管理多个 git 仓库中的源码，可以及时更新最新代码，以方便开发者学习使用。执行以下 4 条命令(**注意：repo init 那行命令不好复制，需要要认真比对命令，有的 pdf 阅读器复制出去可能会格式错误导致命令错误**)：

1

```
book@100ask:~$ git clone https://e.coding.net/codebug8/repo.git
```

2

```
book@100ask:~$ mkdir -p 100ask_imx6ull-sdk && cd 100ask_imx6ull-sdk
```

3

```
book@100ask:~/100ask_imx6ull-sdk$ ../repo/repo init -u https://gitee.com/weidongshan/manifests.git -b linux-sdk -m imx6ull/100ask_imx6ull_linux4.9.88_release.xml --no-repo-verify
```

4

```
book@100ask:~/100ask_imx6ull-sdk$  ../repo/repo sync -j4
```

图 2.14 是下载 IMX6ULL Pro 版源码的过程：

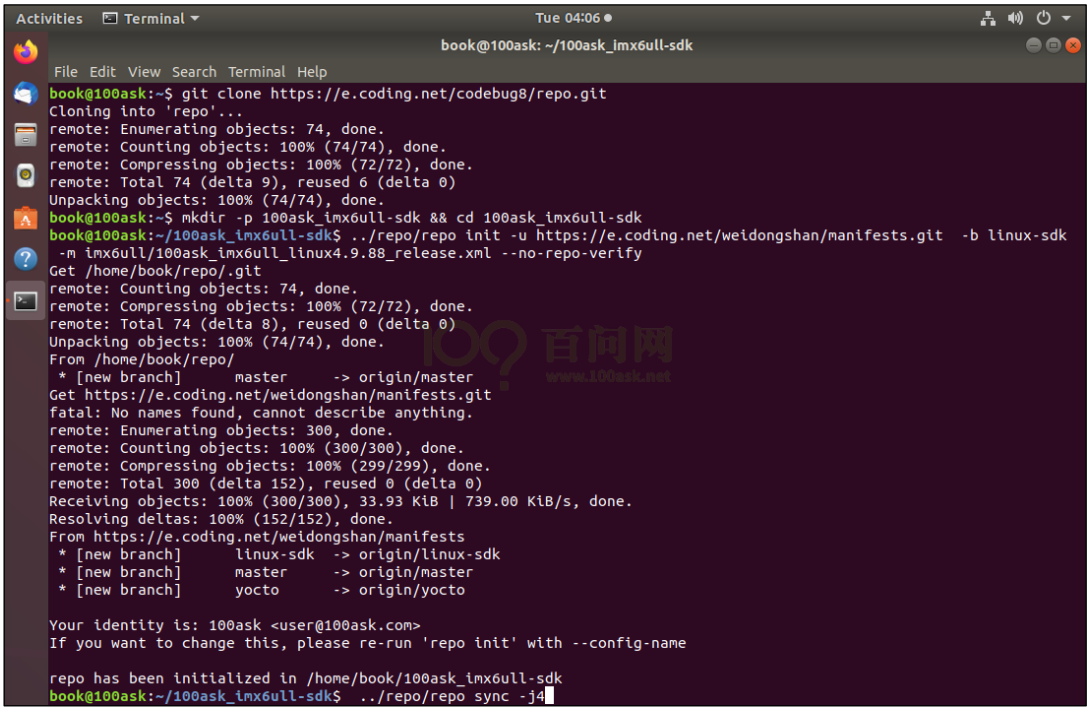


图 2.14 下载源码

注意：下载成功后，可以看到名为 Linux-4.9.88 的目录，可以在 Ubuntu 压缩

它，再传回 Windows。在 Windows 下解压后，用 source insight 建立工程，这样就可以很方便地阅读源码了。

注意：使用 source insight 阅读 Linux 源码的方法，请参考章节《使用 Source Insight 阅读 Linux 内核源码》。

Ubuntu 下压缩命令为(最好是下载之后马上压缩，不要编译内核后再压缩，否则文件太大了)：

```
tar  cjf Linux-4.9.88.tar.bz2  Linux-4.9.88
```

首次下载时如果提示

```
Testing colorized output (for 'repo diff', 'repo status'):
```

此时输入 y 即可，继续执行

```
../repo/repo sync -j4
```

命令即可开始同步源码(同步代码根据网络情况需要等待数分钟)。

图 2.15 为参考示例。

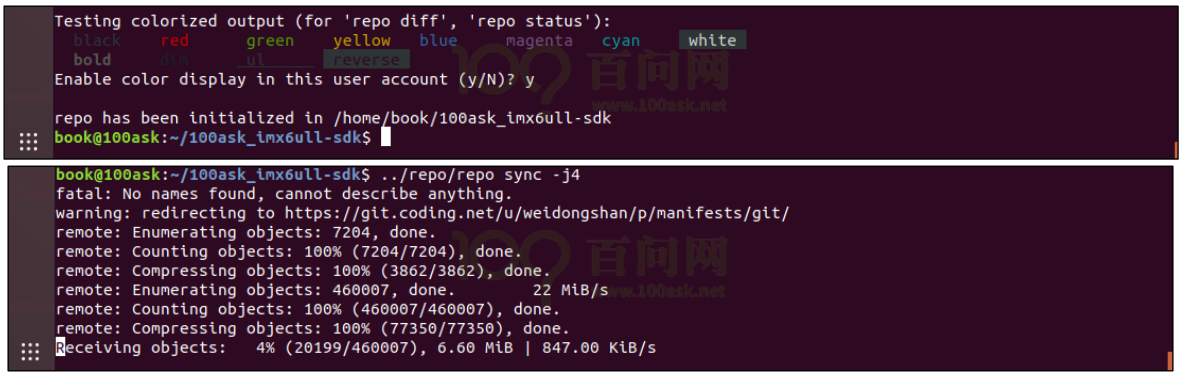


图 2.15 开始同步源码

同步完成后如图 2.16 所示

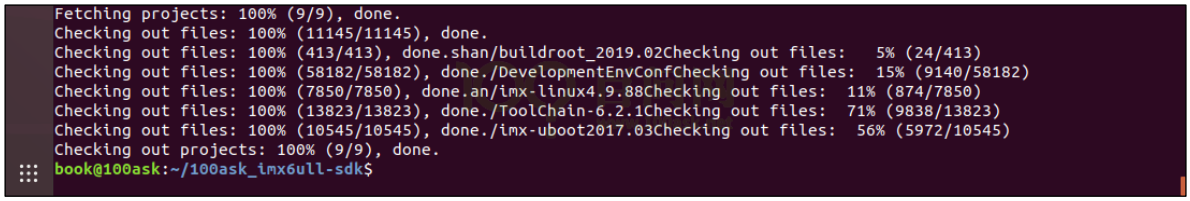


图 2.16 同步完成

注意：repo 在线下载的代码和网盘配套资料中的代码是一致的，我们会每隔一段时间更新一次源码包，如使用在线方式获取源码可以直接在

```
~/100ask_imx6ull-sdk
```

目录下执行

```
../repo/repo sync -c
```

进行同步更新最新代码！

2.6.3 配置交叉编译工具链

交叉编译工具链用来在 Ubuntu 主机上编译应用程序，而这些应用程序是在 ARM 等其他平台上运行。

设置交叉编译工具主要是设置 PATH， ARCH 和 CROSS_COMPILE 三个环境变量，下面介绍具体设置方法。

在本文档中，源码、交叉编译工具链都是存放于/home/book 目录下；如果你的目录不一样，请自行修改本节所讲述的命令。

设置这 3 个环境变量有多种方法，任意选择其中一种方法即可，建议使用“永久生效”的方法。录制视频时我会使用多种开发板，所以在视频里我总是使用“临时生效”的方法。

1 永久生效

如需永久修改，请修改用户配置文件：

```
vim  ~/.bashrc
```

注意：如果不会使用 vim 命令，可以使用图形化的编辑工具，执行：

```
gedit ~/.bashrc
```

在行尾添加或修改，加上下面几行(**请把第 3、4 行合并为一行，有些 PDF 工具无法正确复制甚至丢失“-”符号，请仔细对比**)：

```
export ARCH=arm
export CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

设置完毕后，要执行

```
source ~/.bashrc
```

命令使其生效，并通过

```
arm-buildroot-linux-gnueabihf-gcc -v
```

命令来验证是否配置成功：

```
book@100ask:~$ source  ~/.bashrc
book@100ask:~$ arm-buildroot-linux-gnueabihf-gcc -v
```

以上操作是设置 IMX6ULL Pro 版工具链的过程。

2 临时生效

也可以手工执行“export”命令设置环境变量，该设置**只对当前终端有效(另开一个终端需要再次设置)**。执行以下 3 个命令：

```
book@100ask:~$ export ARCH=arm
book@100ask:~$ export CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
book@100ask:~$ export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

2.6.4 测试交叉编译工具链

执行以下命令测试环境变量：

```
book@100ask:~$ echo $ARCH
arm
book@100ask:~$ echo $CROSS_COMPILE
arm-buildroot-linux-gnueabihf-
```

执行以下命令测试工具链，结果见后图 2.17：

```
book@100ask:~$ arm-buildroot-linux-gnueabihf-gcc -v
```

```
book@book-virtual-machine:~$ arm-buildroot-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=/home/book/100ask_imx6ull-sdk_git/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin/arm-buildroot-linux-gnueabihf-gcc.br_real
COLLECT_LTO_WRAPPER=/home/book/100ask_imx6ull-sdk_git/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin/../libexec/gcc/arm-buildroot-linux-gnueabihf/7.5.0/lto-wrapper
Target: arm-buildroot-linux-gnueabihf
Configured with: ./configure --prefix=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host --sysconfdir=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host/etc --enable-static --target=arm-buildroot-linux-gnueabihf --with-sysroot=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host/arm-buildroot-linux-gnueabihf/sysroot --enable-__cxa_atexit --with-gnu-ld --disable-libssp --disable-multilib --disable-decimal-float --with-gmp=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host --with-mpc=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host --with-mpfr=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host --with-pkgversion='Buildroot 2020.02-gee85cab' --with-bugurl=http://bugs.buildroot.net/ --disable-libquadmath --enable-tls --enable-plugins --enable-lto --enable-threads --with-isl=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host --with-abi=aapcs-linux --with-cpu=cortex-a7 --with-fpu=neon-vfpv4 --with-float=hard --with-mode=arm --enable-languages=c,c++,fortran --with-build-time-tools=/home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/host/arm-buildroot-linux-gnueabihf/bin --enable-shared --enable-libgomp
Thread model: posix
gcc version 7.5.0 (Buildroot 2020.02-gee85cab)
```

图 2.17 测试编译工具链

2.7 使用 Source Insight 阅读 Linux 内核源码

在后面开发驱动程序时，驱动程序中用到的函数都是来自内核，所以可以先在 Windows 下创建内核的 Source Insight 工程。

如果你不想学习驱动开发，那么可以不创建内核的工程。但是以后学习大型 APP 时，也可以使用 Source Insight 来阅读、编写代码，可以借鉴本节讲解的 Source Insight 用法。

2.7.1 Source Insight 简介

Source Insight 是 Source Dynamics 公司出品的源代码编辑器。Source Insight 提供语法突出显示，代码导航和可自定义的键盘快捷键。它不仅仅是一个编辑器，而是一个理解大型源代码库的工具，因此被称为“程序编辑器和分析器”。它灵活轻便，提供有用的功能，如关系，上下文和符号窗口。它在建源码工程时，构建了符号信息的内部数据库，所以还可以显示引用树，类继承图和调用树。它的最大好处是加快了对不熟悉项目的代码理解。

参考网址

- 官网主页 https://www.sourceinsight.com/
- 软件下载页面 https://www.sourceinsight.com/trial/
- 用户使用教程
  https://www.sourceinsight.com/doc/v4/userguide/index.html

2.7.2 在 Windows 上解压内核源码

前面章节《2.6.2 下载 BSP》里下载到内核后，在 Ubuntu 下压缩了内核，把压缩文件通过 FileZilla 传回 Windows，并解压。

在 Windows 解压内核时会提示一些错误，会提示是否覆盖文件，选择“覆盖”即可。这是因为 Linux 下的文件区分大小写，a.c 和 A.c 以不同的文件，但是 Windows 下不区分大小写，这 2 个文件是同一个。这些错误不会影响我们阅读源码。

2.7.3 建立工程示例

本节新建一个 linux kernel 的 source Insight 工程，你也可以为其他 APP 建立工程，方法是一样的。

1 新建工程

运行 source Insight，点击菜单“Project->New Project”，如图 2.18 所示：

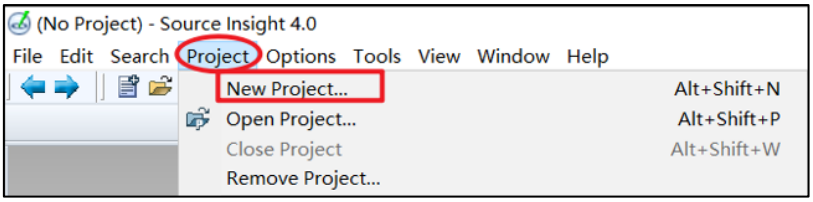


图 2.18 SI 新建工程

2 设置工程名及工程数据目录

在弹出的 New Project 对话框中设置“New project name”(项目的名称)，然后设置 Where do you want to store the project data file?(项目文件保存位置)，点击 Browse 按钮选择源码的目录即可，如图 2.19:

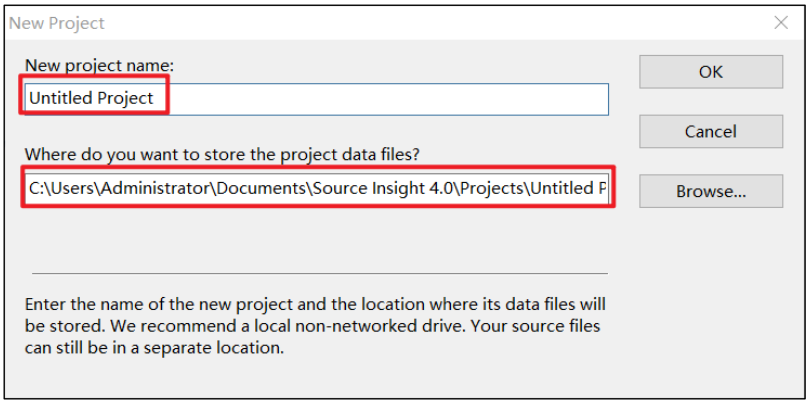


图 2.19 选择工程存储路径

3 指定源码目录

设置源码目录：

Project Source Directory - the main location of your source files”()

点击红框左边“…”选择源码目录，点击 OK，如图 2.20:

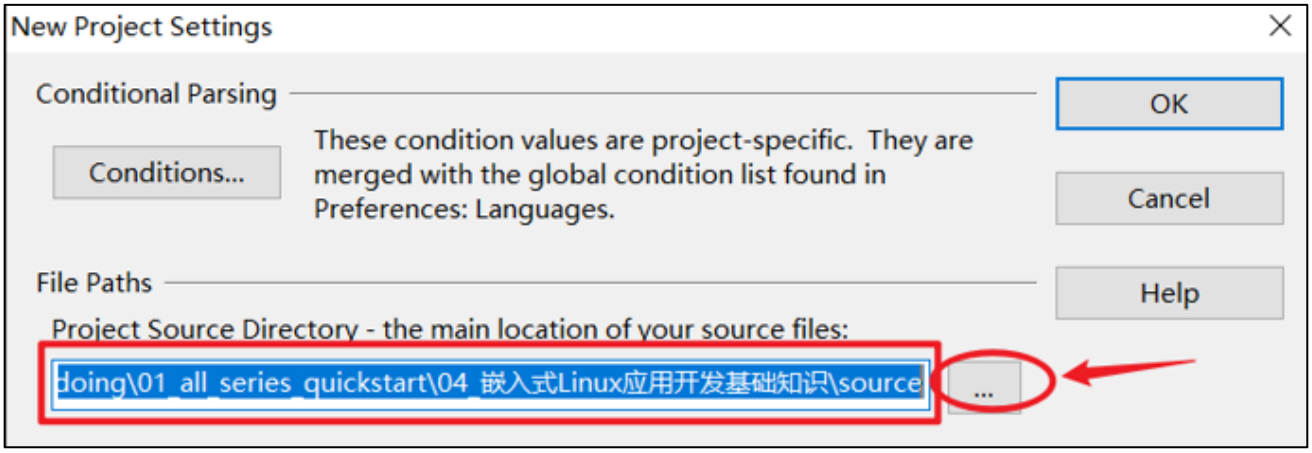


图 2.20 设置源码目录

4 添加源码

在新弹出的对话框中，点击“Add”或“Add All”。“Add”是手动选择需要添加的文件，而“Add All”是添加所有文件。我们使用“Add All”,在弹出的提示框中选中“Recursively add lower sub-directories”(递归添加下级的子目录)并点击 OK。同样的 Remove File,Remove All 是移除单个文件或者移

除所有文件，如图 2.21：

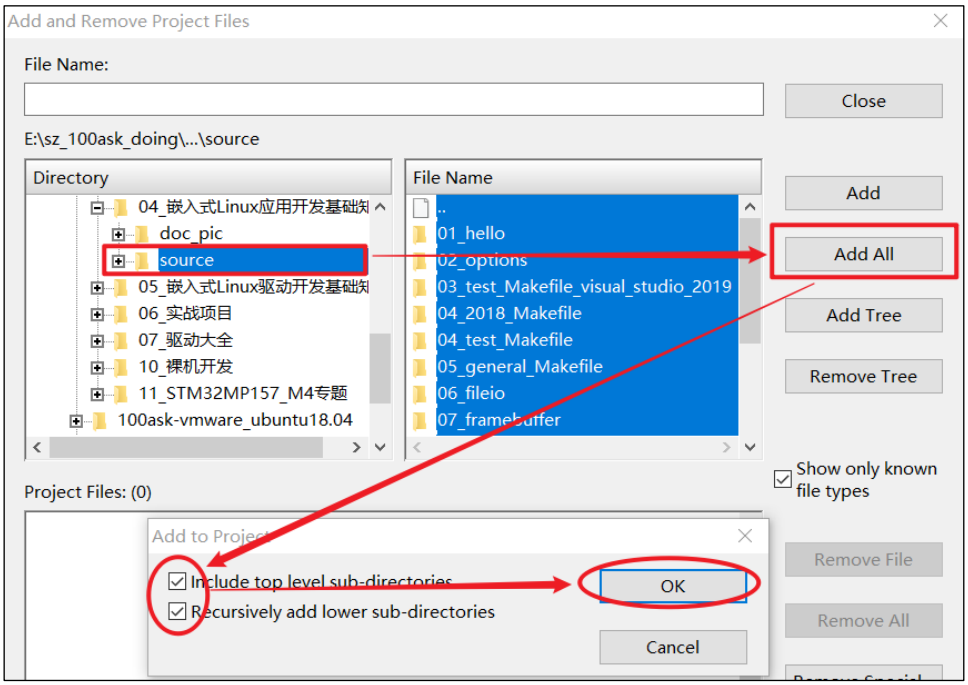
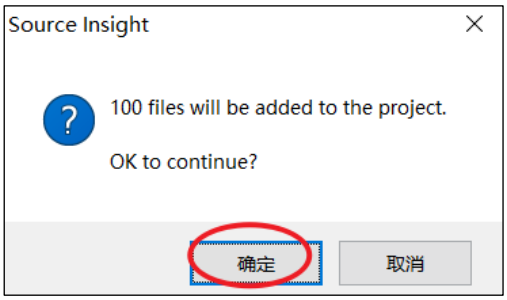


图 2.21 添加源码

添加文件完成后会弹出下面窗口，点击“确定”即可：



此时界面会返回到主界面，如图 2.22，点击“Close”：

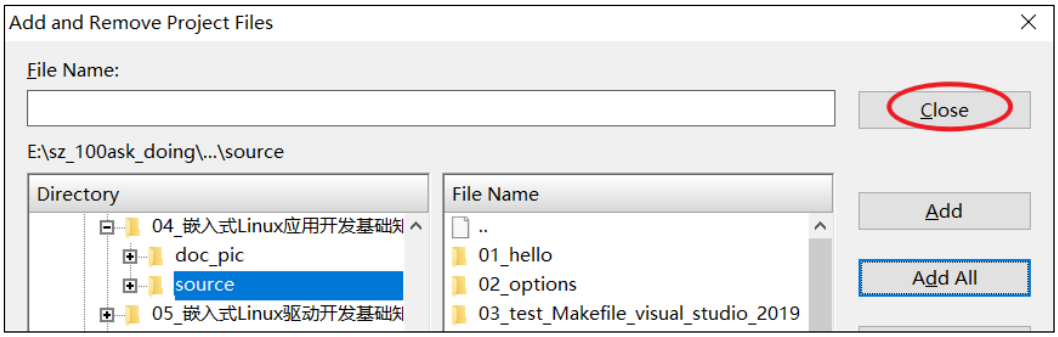


图 2.22 添加完成关闭

5 同步文件

同步文件的意思是让 Source Insight 去解析源码，生成数据库，这样有助于以后阅读源码。比如点击某个函数时就可以飞快地跳到它定义的地方。

先点击菜单“Project->Synchronize Files”，如图 2.23 所示：

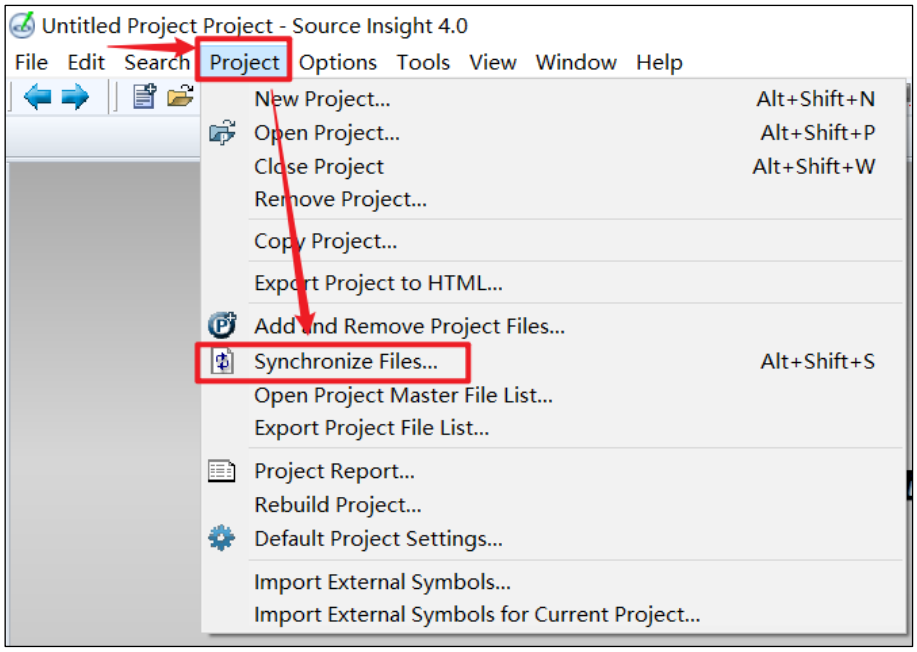


图 2.23 同步文件

在弹出的对话框中 选中“Force all files to be re-parsed”(强制解析所有文件)，并点击“Start”按钮开始同步，如图 2.24 所示：

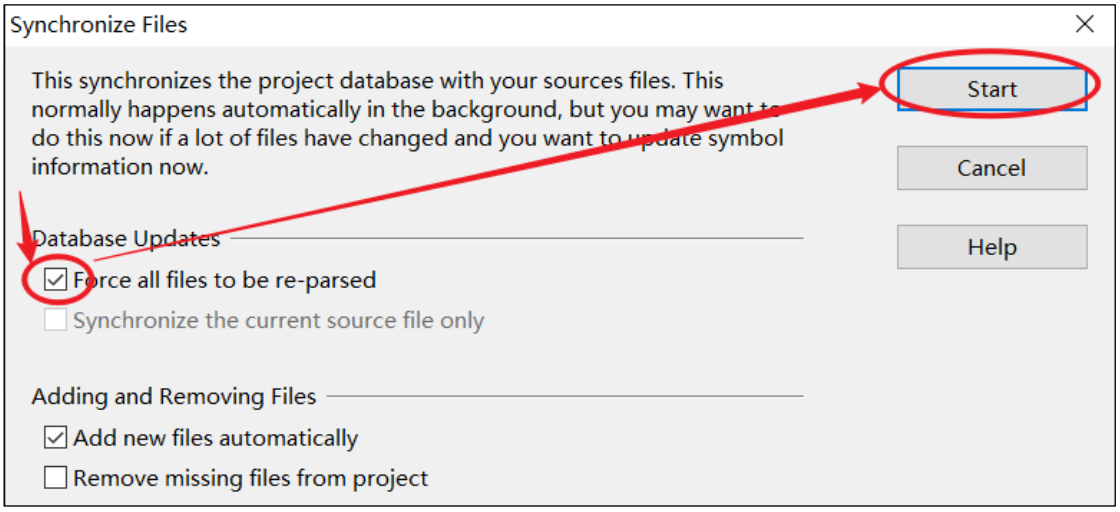
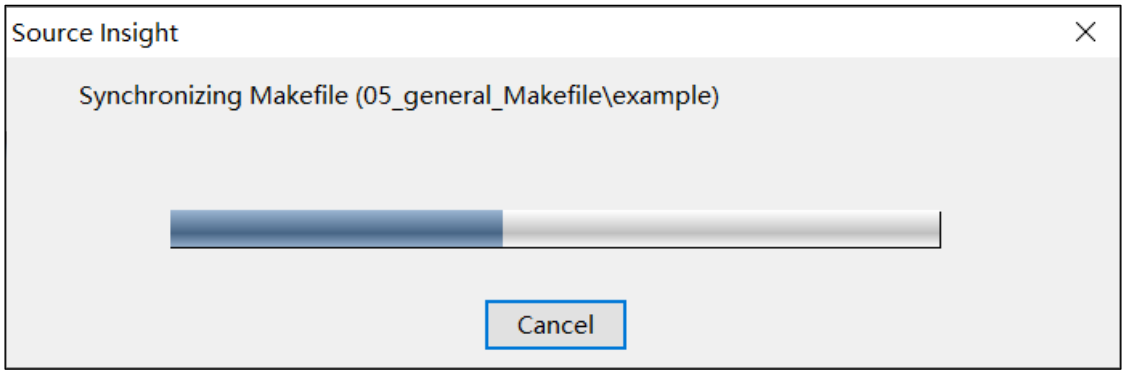


图 2.24 开始同步文件



2.7.4 操作示例

1 打开工程

前面建议工程后，就会自动打开了工程。如果下次你想打开工程，启动 Souce Insight 后，点击菜单“Project -> Open Porject”就可以在一个列表中选择以前建立的工程，如图 2.25：

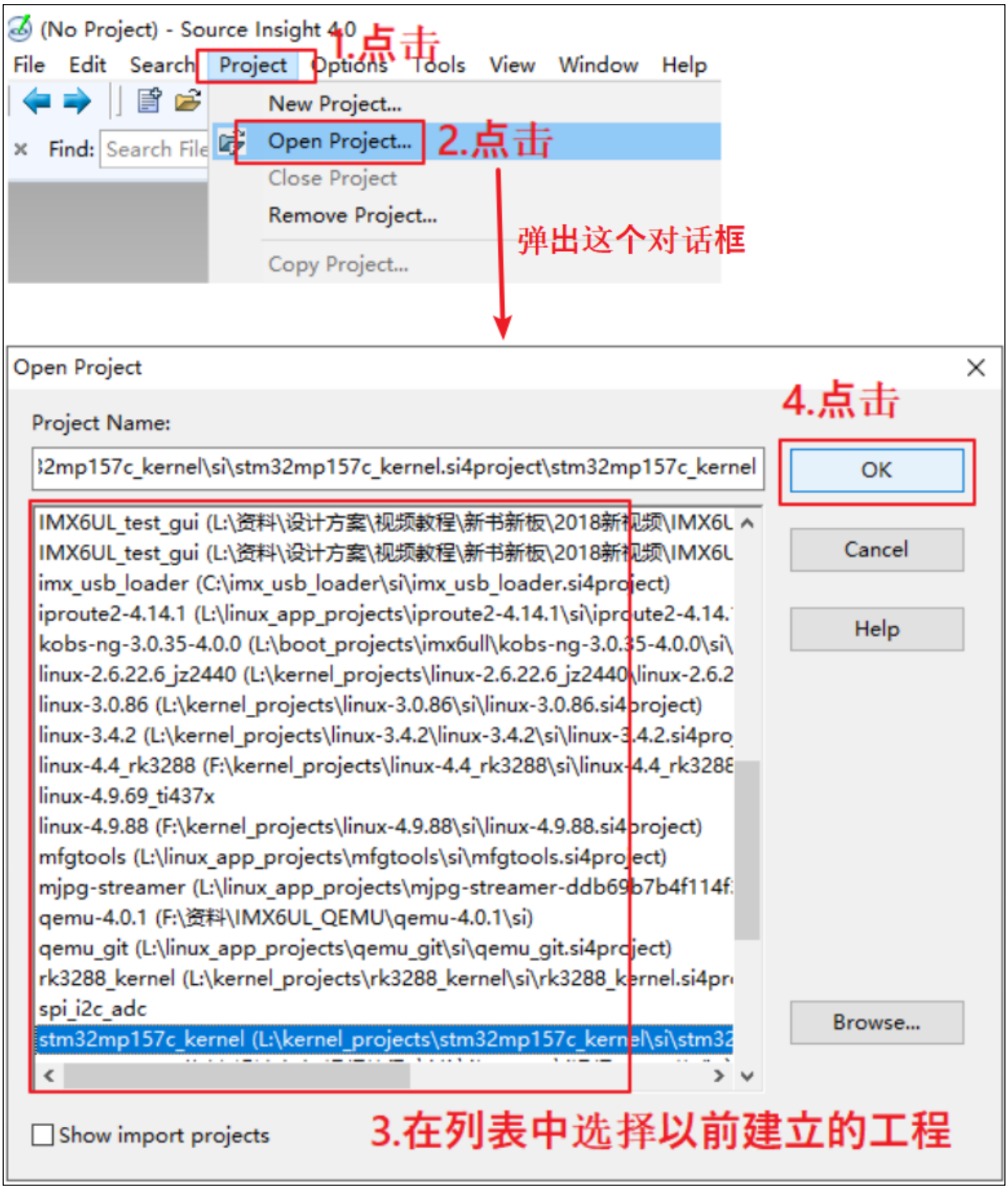


图 2.25 SI 打开工程

2 在工程中打开文件

点击"P"图标打开文件列表，双击文件打开文件，也可以输入文件名查找文件，如图 2.26 所示：

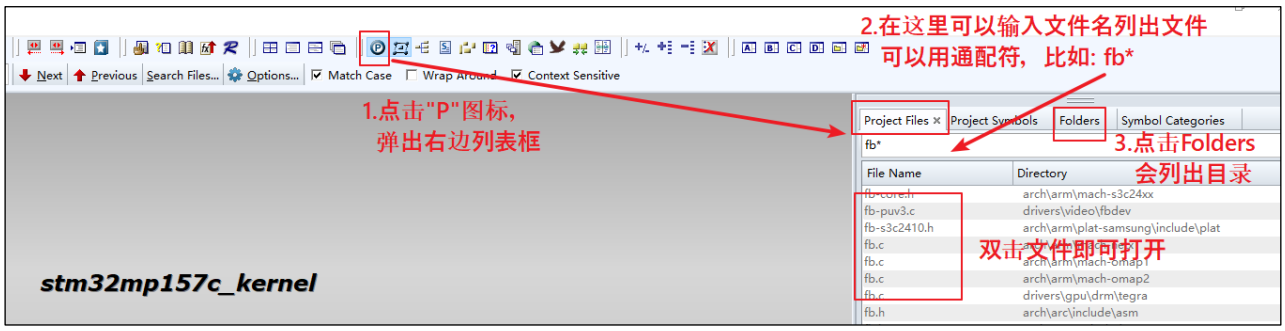


图 2.26 打开文件

3 在文件中查看函数或变量的定义

打开文件后，按住 ctrl 键的同时，用鼠标点击函数、变量，就会跳到定义它的位置，如图 2.27 所示：

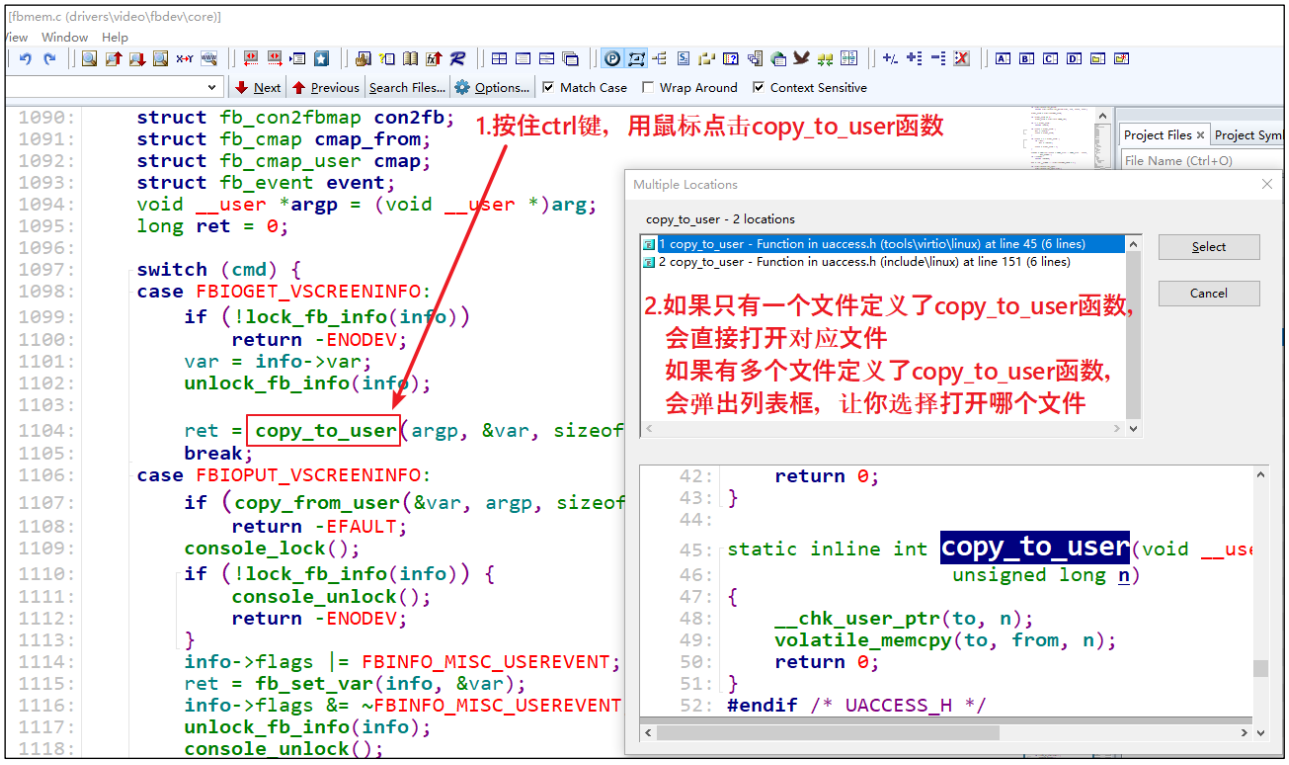


图 2.27 查看函数或变量

4 查找函数或变量的引用

双击函数，右键点击弹出对话框选择“Lookup Reference”；或者双击函数后，使用快捷键"ctrl+/"来查找引用，如图 2.28：

图 2.28 查找函数或变量

5 其他快捷键

表 2-2 SI 其它快捷键

快捷键	说明
Alt + ,	后退
Alt + .	前进
F8	高亮选中的字符
Ctrl+F	查找
F3 或 Shift+F3	往前查找
F4 或 Shift+F4	往后查找

第3章 IMX6ULL Pro 开发板基本操作

3.1 100ask_imx6ull pro 开发板硬件资源简介

开发板图片如图 3.1，各个标号对应的硬件在板子背后都写有名字：

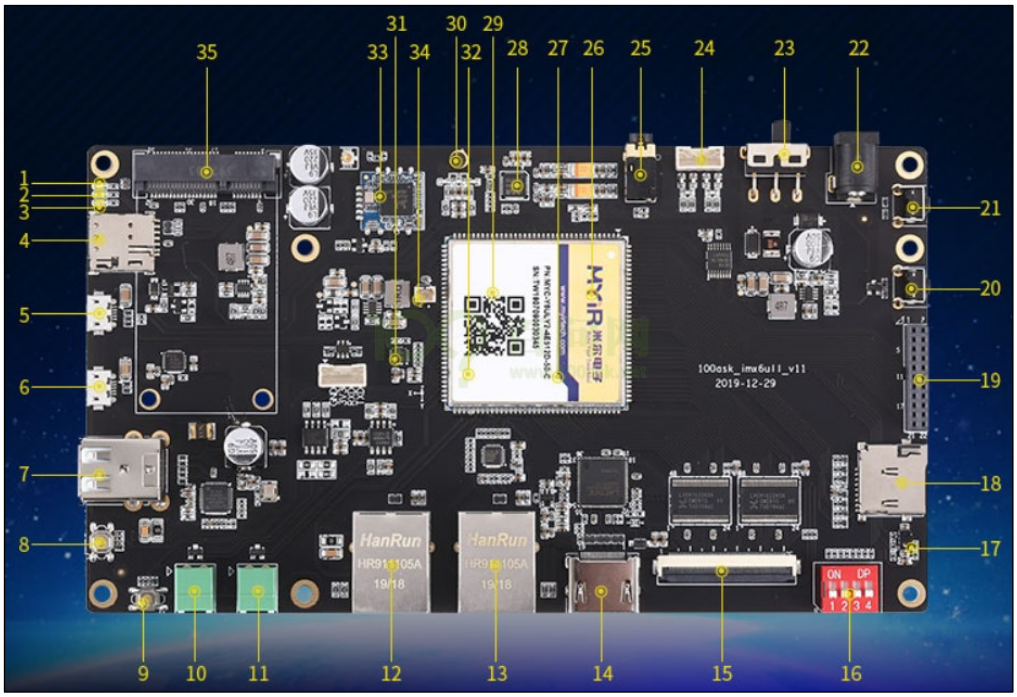


图 3.1 100ASK IMX6ULL pro 开发板

上图中，各标号的硬件含义如图 3.2 所示：

图 3.2 100ASK IMX6ULL pro 标号对应硬件

3.2 100ask_imx6ull pro 开发板软件资源简介

3.2.1 开发环境

图 3.3 开发板开发环境

3.2.2 核心软件

图 3.4 核心软件

3.2.3 文件系统

图 3.5 文件系统

3.2.4 各模块的详细内容

类别	名称	描述信息	源码
引导程序	U-boot 2017.03	第一级引导启动程序	YES
Linux内核/驱动	Linux 4.9.88	基于官方imx-4.9.88-2.0.0_ga版本	YES
	LED	GPIO LED驱动	YES
	KEY	GPIO KEY驱动	YES
	I2C	I2C 总线驱动	YES
	Ethernet	10/100Mbps以太网驱动	YES
	MMC	MMC/eMMC/TF卡存储驱动	YES
	LCD	显示驱动，支持4.3寸和7寸液晶屏	YES
	RTC	实时时钟驱动	YES
	Touch	电容(FT5316)，电阻触摸	YES
	USART	串口驱动	YES
	USB Host	USB Host驱动	YES
	Audio	WM8960驱动	YES
	CAN bus	CAN总线驱动	YES
	RS485	RS485总线驱动	YES
	Camera	OV2659驱动	YES
	WiFi/蓝牙	RTL8732a 使用usb驱动	YES
	LTE模块	仅支持移远EC20，使用USB驱动	YES
文件系统	Busybox	基于官方1.30版本	YES
		提供一键编译脚本	YES
		提供常用选项配置文件讲解	
	Yotcto rootfs	基于Yocto构建带Qt 5.6的文件系统	YES
		基于Yocto构建终端型的通用文件系统	YES
		如何使用bitbake命令	
		如何加入自己的包	
		如果创建自己的image系统	
		如何创建自己的主机配置...	
	Buildroot	基于buildroot 2019.02构建imx6ull文件系统	YES
		Audio and video applications使用示例	YES
		Compressors and decompressors示例	YES
		Debugging profiling and benchmark	YES
		Development tools示例	YES
		Filesystem and flash utilities	YES
		Fonts cursors icons sounds and themes	YES
		Graphic libraries and applications	YES
		Hardware handling	YES
		Interpreter languages and scripting	YES
		Networking applications	YES
		Package managers	YES
		System tools	YES
	发行版系统	如何制作基于arm架构ubuntu系统	YES
应用程序	GPIO KEY	GPIO按键指示例程	YES
	GPIO LED	按键指示灯例程	YES
	NET	TCP/IP Sokect C/S例程	YES
	RTC	实时时钟例程	YES
	RS232	RS232例程	YES
	Audio	Audio例程	YES
	LCD	显示屏例程	YES
系统工具	资源监视工具	Netdata Netstat Tcpdump Top Linux command line	YES
		Sysdig	YES
		Sysprof	YES
		Sysstat tool suite	YES
		Systemtap	YES
		Kmemleak	YES
	调试工具	Debugfs	YES
		Perf	YES
		Blktrace	YES
		Dmesg and Linux kernel log	YES
		Ftrace	YES
		Strace	YES
		LTTng	YES
		Trace-cmd and kernelshark trace viewer	YES
编译工具链	Cross compiler	Linaro GCC 6.2 Hardfloat	BINARY
	Cross	Yocto GCC 6.2 Hardfloat	BINARY

图 3.6 模块细节

3.3 启动方式选择

板子上的红色拨码开关用来设置启动方式，支持这 3 种方式：EMMC 启动、SD 卡启动、USB 烧写。板子背后画有一个表格，表示这 3 种方式如何设置，如表 3-1：

表 3-1 启动方式

BOOT CFG				
BOOT	SW1(LCD_DATA5)	SW2(LCD_DATA11)	SW3(BOOT_MODE0)	SW4(BOOT_MODE1)
EMMC	OFF	OFF	ON	OFF
SD	ON	ON	ON	OFF
USB	X	X	OFF	ON

这 3 种启动方式的设置示意图 3.7 如下：

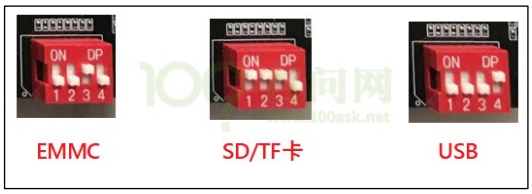


图 3.7 拨码开关与启动方式

注意：设置为 USB 启动时，不能插上 SD 卡、TF 卡；上电之后才可以插卡。刚出厂的板子在 EMMC 上烧写了系统，你可以设置为 EMMC 启动方式。

3.4 串口连接

在后面的操作里，都是通过串口与板子进行“交流”。串口是串行接口的简称，是指数据一位一位地顺序传送，其特点是通信线路简单。

3.4.1 连接串口线和电源线、配置串口工具

如图 3.8 所示将串口线与电脑、板子连接，开发板插上电源。

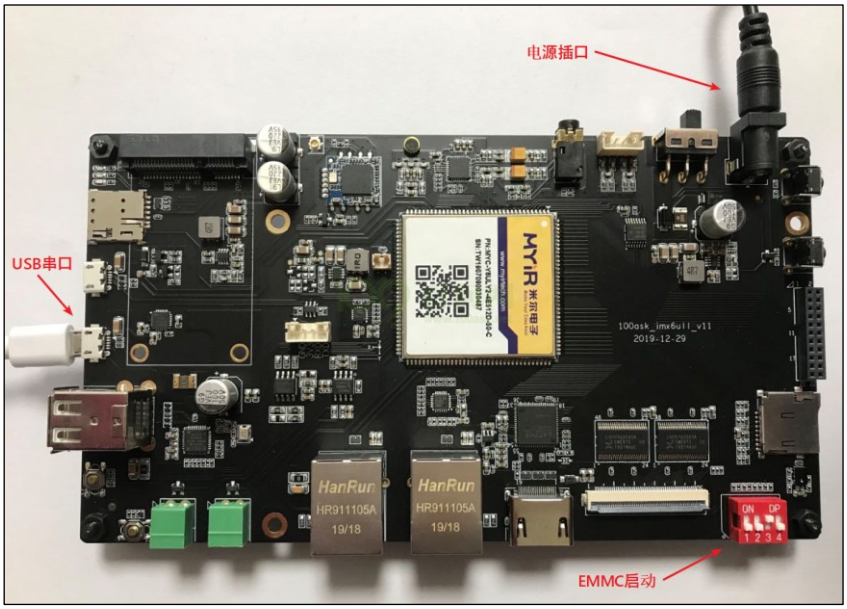


图 3.8 连接示意图

其中特别需要注意的几点：

- 板子的启动拨码开关拨到正确的方式，保证该启动方式里面有系统可以运行；
- 板子如图所示插上配套的电源到电源接口，电源开关暂时不用打开；

3.4.2 安装 USB 串口模块驱动

接好 micro usb 数据线后，Windows 会自动安装驱动(安装可能比较慢，等一分钟左右)。打开电脑的“设备管理器”，在“端口(COM 和 LPT)”项下，可以看到如图 3.9 中的“(COM17)”或“(COM19)”。开发板上的 USB 串口芯片可能是 CP210x 或 CH9102，它们的性能是一样的。你电脑上显示的 COM 序号可能不一样，记住你电脑显示的数字。

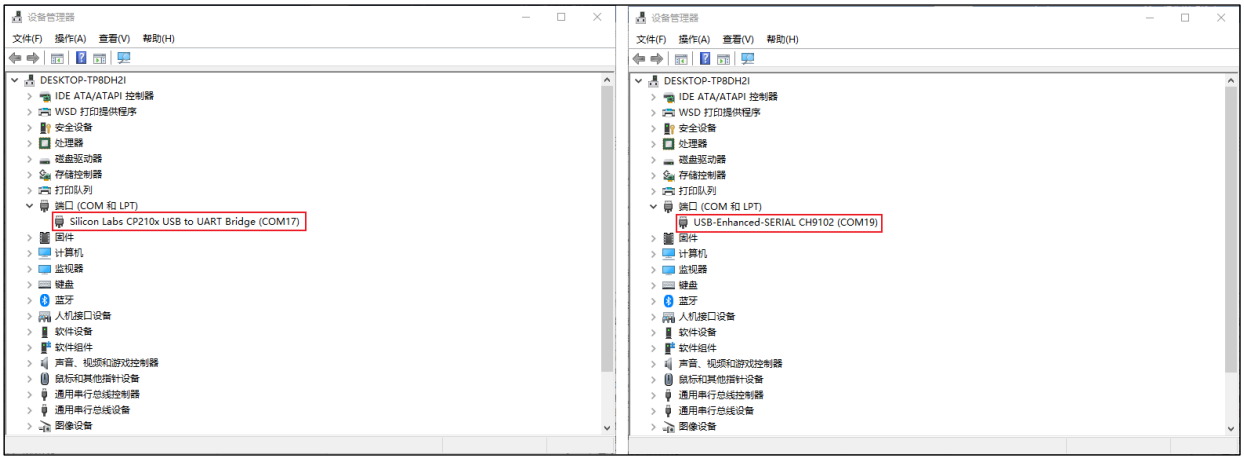


图 3.9 串口在设备管理器中的表示

如果电脑没有显示出端口号，就需要手动安装驱动，已经将驱动安装包放入网盘资料中了：

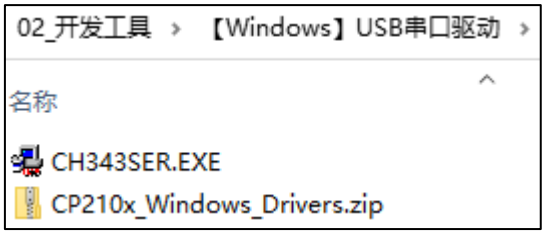


图 3.10 USB 串口驱动

如果电脑中没有自动安装驱动，在“设备管理器”会有黄色感叹号提示当前连接的是哪种类型的串口芯片，根据提示选择驱动安装。如果提示中有“CP210x”字样则选择“CP210x_Windows_Drivers.zip”，否则就选择另外一个驱动安装。

3.4.3 使用 MobaXterm 软件打开串口

打开 MobaXterm，点击左上角的“Session”，在弹出的界面选中“Serial”，如下图所示选择端口号（前面设备管理器显示的端口号 COM17 或 COM19）、波特率（Speed 115200）、流控（Flow Control: none），最后点击“OK”即可。步骤如图 3.11 所示。

注意：流控（Flow Control）一定要选择 none，否则你将无法在 MobaXterm 中

向串口输入数据。

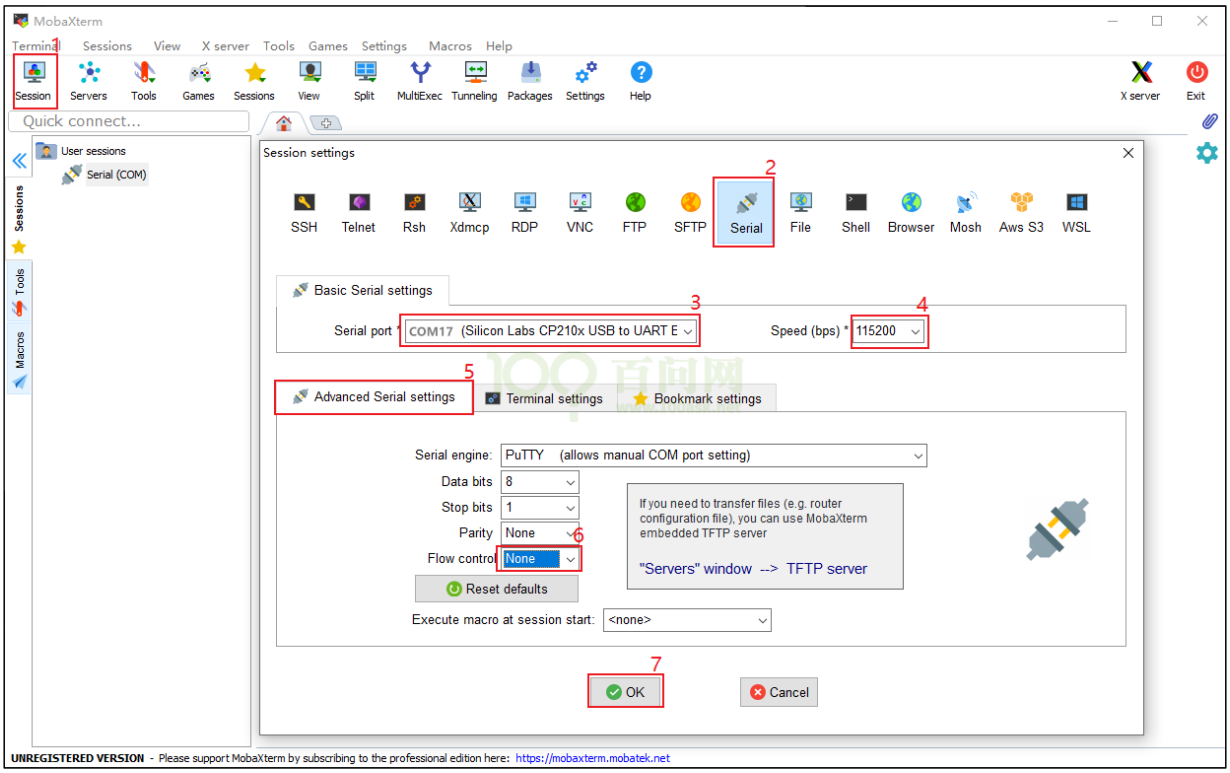


图 3.11 Mobaxterm 设置串口

随后显示一个黑色的窗口， 此时打开板子的电源开关，将收到板子串口发过来的数据，如图 3.12 所示。

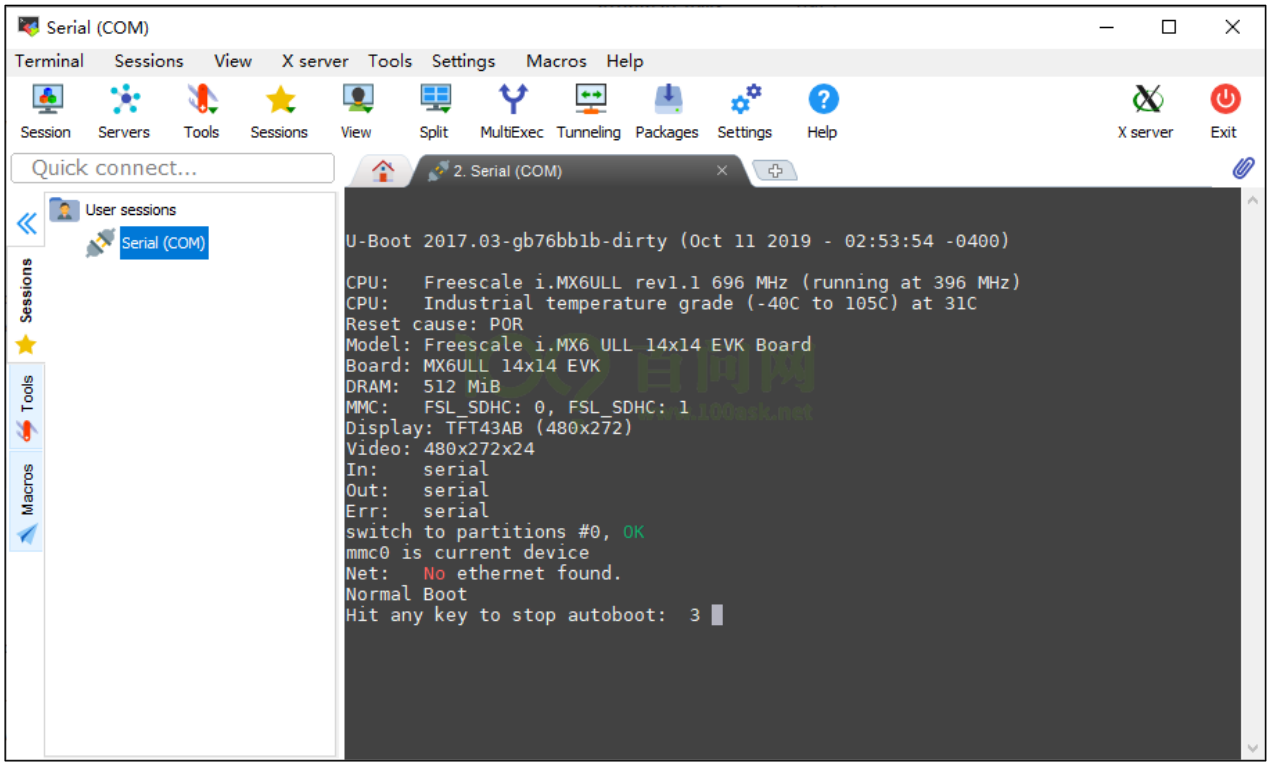


图 3.12 串口数据在 Mobaxterm 上的显示

3.4.4 开发板登录名是 root，无需密码

在串口看到“imx6ull login:”时，输入“root”并回车即可，如图 3.13 所示：

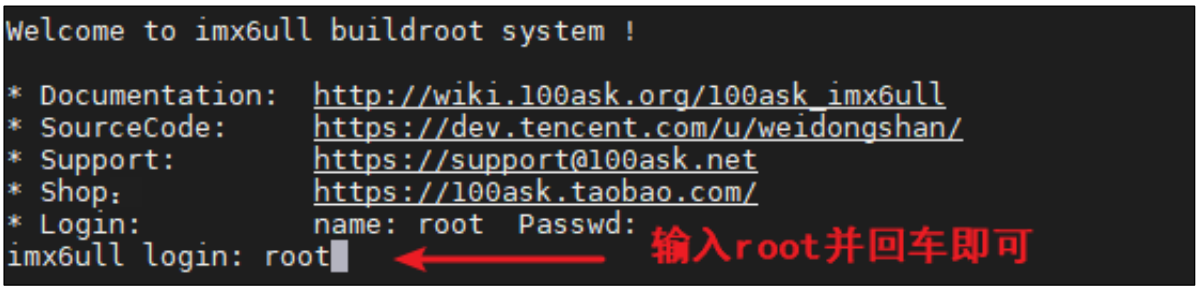


图 3.13 输入 root 登录开发板

3.5 通过串口操作开发板

在串口看到“imx6ull login:”这类登录的提示信息时，输入“root”并回车即可，然后就可以执行各种 Linux 命令了，如图 3.14 所示：

图 3.14 通过串口在开发板上体验 Linux 命令

3.6 配置桥接网卡

参考

《>>>>>>>>第三篇第 1 章 1.33 配置桥接网卡》

《>>>>>>>>第三篇第 1 章 1.44 三者互 ping 验证》

3.7 开发板挂载 Ubuntu 的 NFS 目录

什么是 NFS 协议？

NFS 实现了一个跨越网络的文件访问功能，如下图可以简要说明其原理。其整个架构为 Client-Server 架构，客户端和服务端通过 RPC 协议进行通信，RPC 协议可以简单的理解为一个基于 TCP 的应用层协议，它简化命令和数据的传输。NFS 最大的特点是将服务端的文件系统目录树映射到客户端，而在客户端访问该目录树与访问本地文件系统没有任何差别，客户端并不知道这个文件系统目录树是本地的还是远在另外一台服务器。

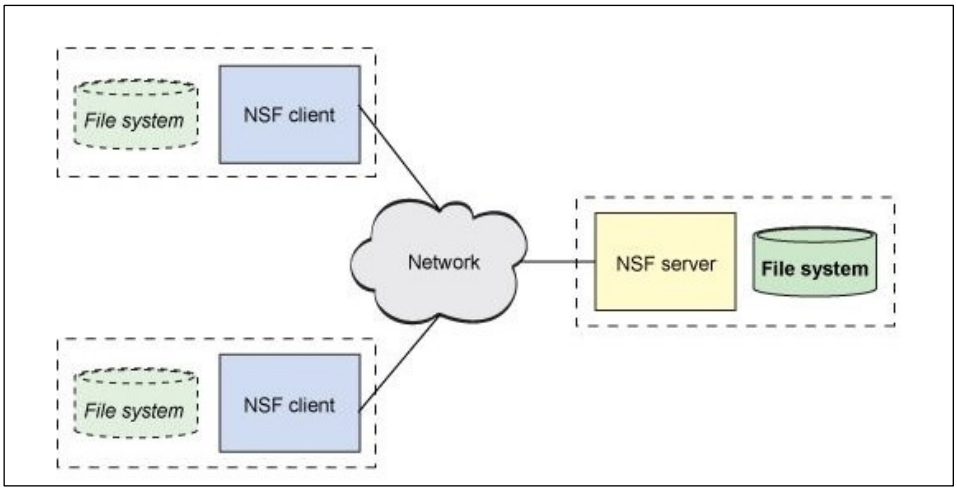


图 3.15 NFS 系统

我们为什么要挂载 ubuntu 的 nfs 目录？

我们有些时候需要多次调试开发板文件系统内的某个应用程序，这就需要多次进行编译拷贝等操作，所以我们在前期进行调试时可以直接让开发板使用 ubuntu 的 nfs 目录下文件系统来进行远程调试，用以提高调试效率，加快研发速度。

3.7.1 确定 ubuntu 的桥接网卡 IP

开发板要想访问 Ubuntu,要先确定 ubuntu 的桥接网卡的 IP，在 Ubuntu 终端下使用 ifconfig 命令来查看 IP。

```
book@100ask: ~
File  Edit  View  Search  Terminal  Help
book@100ask:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.37.133  netmask 255.255.255.0  broadcast 192.168.37.255
        inet6 fe80::6c24:cddc:c680:54e5  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:04:04:b3  txqueuelen 1000  (Ethernet)
        RX packets 104588  bytes 82668872 (82.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 55876  bytes 3642371 (3.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ens36: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.11  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::9123:8fc4:5f99:8532  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:04:04:bd  txqueuelen 1000  (Ethernet)
        RX packets 1618  bytes 385142 (385.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3300  bytes 267119 (267.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 7311  bytes 670817 (670.8 KB)
```

NAT

桥接

图 3.166 ifconfig 获取 Ubuntu 的 IP

3.7.2 在开发板上执行 mount nfs 命令

ubuntu 的 IP 是 192.168.5.11，确保开发板能 ping 通 ubuntu 后，在开发板上执行以下命令挂载 NFS：

```
[root@100ask:~]# mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
```

mount 命令用来挂载各种支持的文件系统协议到某个目录下。

mount 成功之后，开发板在/mnt 目录下读写文件时，实际上访问的就是 Ubuntu 中的/home/book/nfs_rootfs 目录，所以开发板和 Ubuntu 之间通过 NFS 可以很方便地共享文件。

在开发过程中，在 Ubuntu 中编译好程序后放入/home/book/nfs_rootfs 目录，开发板 mount nfs 后就可以直接通过/mnt 访问 Ubuntu 中的文件。

3.8 使用 FileZilla 在 Windows 和开发板之间传文件

双击打开 FileZilla 后，按图 3.177 操作：

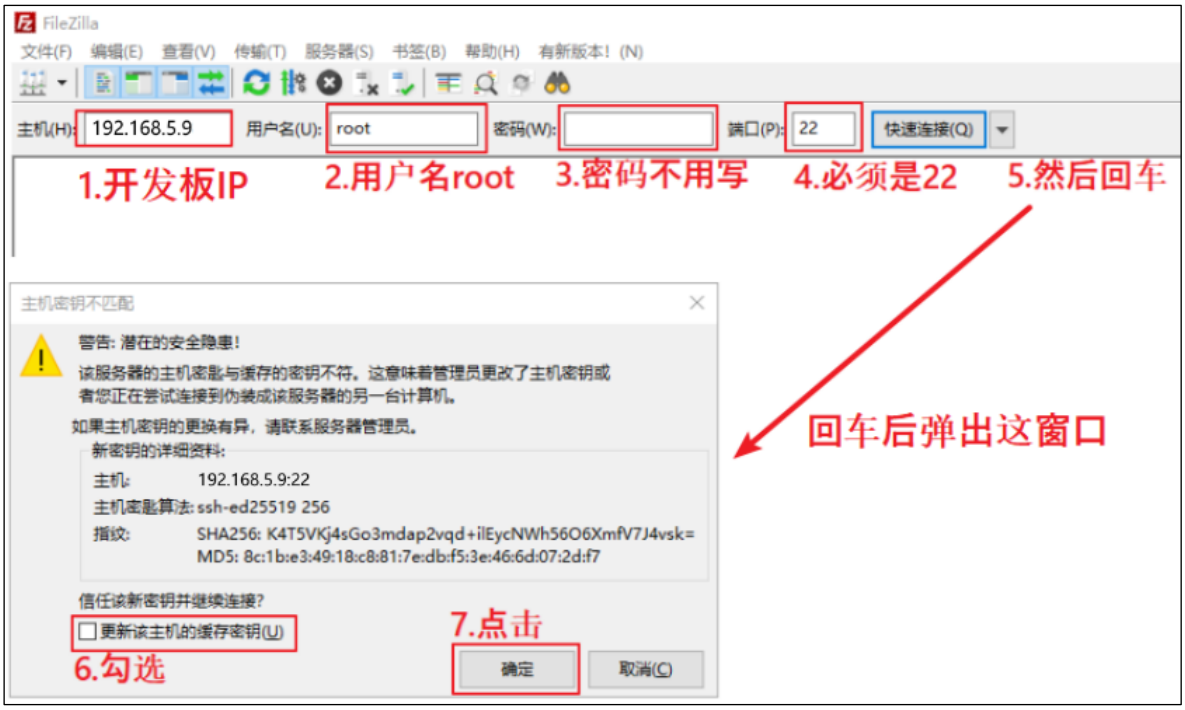


图 3.177 FileZilla 连接 Windows 和开发板

在 Filezilla 中，左边是 Windows 文件，右边是开发板的文件，如图 3.18：

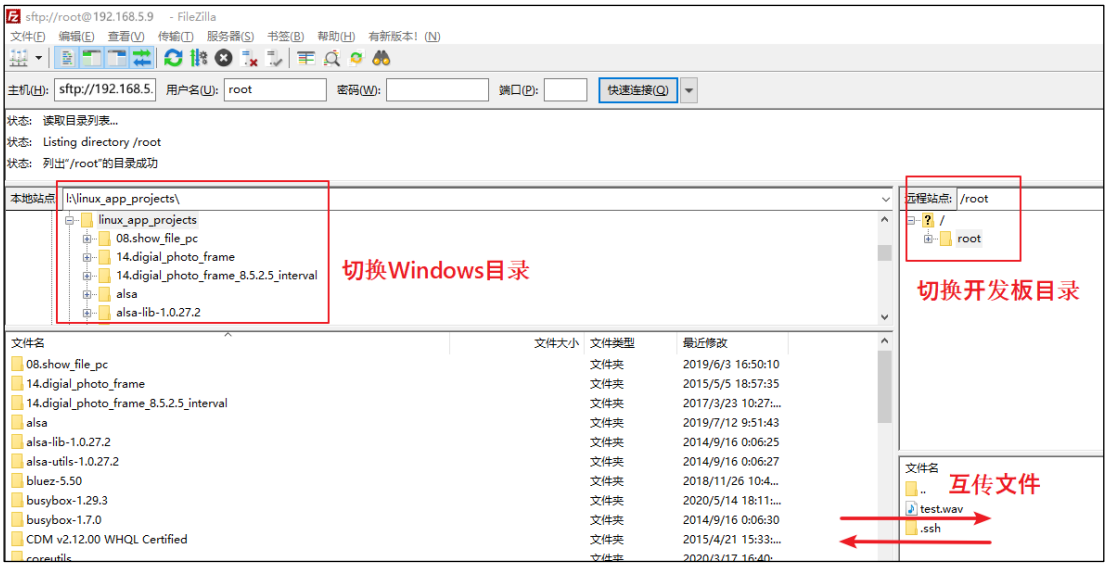


图 3.188 开发板和 Windows 互传文件

3.9 使用 TFTP 服务传输文件

开发板上可以使用 tftp 命令从 Ubuntu 或 Windows 传输文件，前提是：Ubuntu 或 Windows 上先安装 TFTP 服务。

3.9.1 在 Ubuntu 安装 TFTP 服务

在 Ubuntu 中执行以下命令安装 TFTP 服务：

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

然后，创建 TFTP 服务器工作目录，并打开 TFTP 服务配置文件，如下：

```
mkdir  -p  /home/book/tftpboot
sudo chmod 777 /home/book/tftpboot
sudo vim /etc/default/tftpd-hpa
```

在配置文件/etc/default/tftpd-hpa 中，添加以下字段：

```
TFTP_DIRECTORY="/home/book/tftpboot"
TFTP_OPTIONS="-l  -c  -s"
```

最后，重启 TFTP 服务：

```
sudo service tftpd-hpa restart
```

查看 tftp 服务是否在运行,运行如下命令，即可查看是否在后台运行。

```
ps -aux | grep “tftp”
```

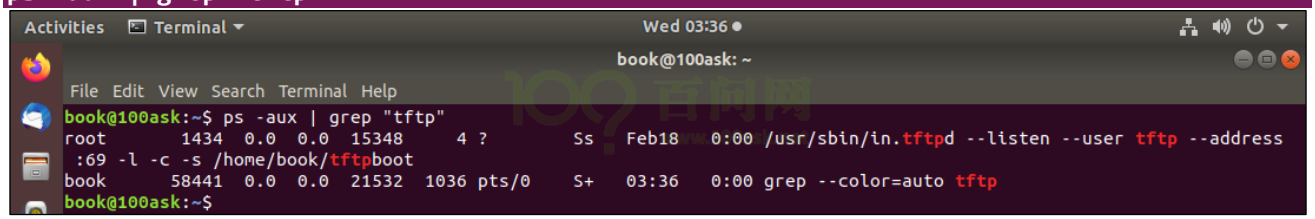


图 3.19 查看 tftp

3.9.2 在 Windows 安装 TFTP 服务

Windows 上的 TFTP 服务由一个应用程序 tftpd64 提供，下载后双击运行，

再做些设置即可。tftpd64 的前身是 tftpd32，它是 32 位的程序。对于 64 位电脑，请使用 tftpd64。

tftpd64 的官网为：http://tftpd32.jounin.net/建议下载“portable edition”版本，无需安装直接运行。官网不好打开的话，直接百度搜“tftpd64”即可，它是免费软件。

在“网盘的配套的资料\02_开发工具\【Windows】TFTP 文件传输工具”中，也有此工具。

tftpd64 的使用非常简单，运行后只需要设置 3 步：

第1步 选择目录(开发板将从这个目录读、写文件)

第2步 选择桥接用的网卡的 IP。

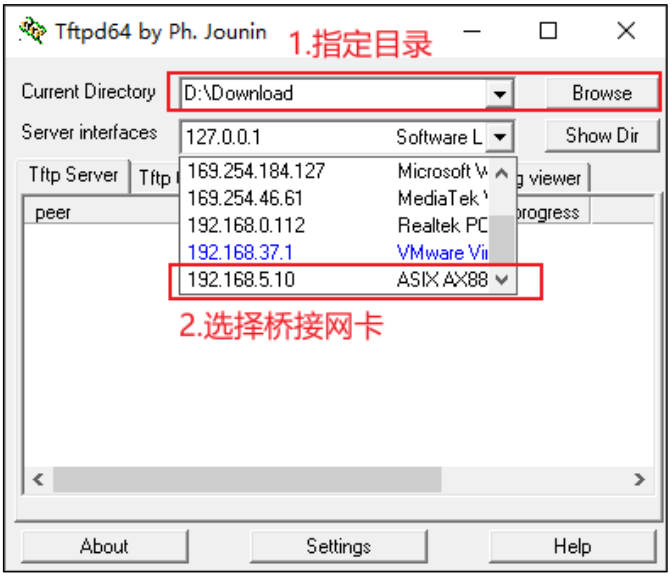


图 3.20 选择网卡

第3步 设置防火墙，直接关闭防火墙；或是允许 tftpsever 使用网络：
对于 Windows 10 可以按图 3.21 操作，对于其他操作系统，请自行百度：

图 3.21 设置防火墙

① 点击 Windows 的“开始”按钮；
② 在菜单栏点击进入“设置”界面；
③ 在“设置”界面搜索框内搜索“防火墙”；

④ 在搜索结果中选择“允许应用通过 Windows 防火墙”；
⑤ 在新界面点击“更改设置”；
⑥ 找到“TFTP Server”将其全部勾选，然后点击“确定”保存应用设置；

3.9.3 开发板通过 tftp 传输文件

以 Ubuntu 为例，Windows 也是类似的。如何下载文件到开发板上？

首先确保 Ubuntu 或 Windows 的 tftp 服务目录内，有需要下载到板子上的文件，比如：

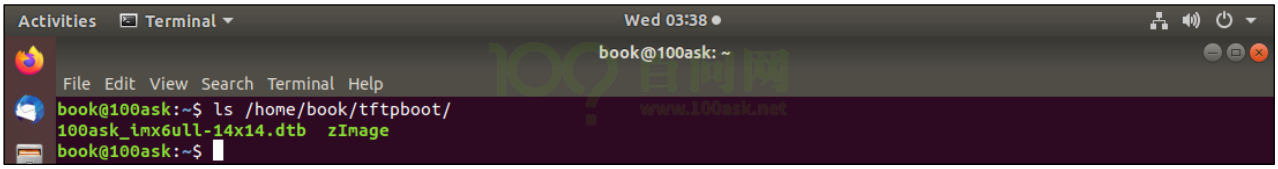


图 3.22 zImage 和 dtb

比如下载 Ubuntu 服务器下的 zImage 文件，则在开发板上执行如下命令(Ubuntu 的桥接网卡 IP 是 192.168.5.11)：

```
[root@100ask:~]# tftp -g -r zImage 192.168.5.11
```

下载后的文件如图 3.23 所示。

图 3.23 下载后开发板上的文件

如何从开发板上传文件到 Ubuntu 或 Windows？比如我们现在开发板家目录下创建一个 **1.txt** 的文本文件，然后写入 **111111…. ：**

```
[root@100ask:~]# vi 1.txt
[root@100ask:~]# cat 1.txt
11111111111111111111111
```

图 3.24 开发板上创建文件

然后在开发板上执行如下命令上传此文件到 Ubuntu 的 tftp 服务目录下(Ubuntu 的桥接网卡 IP 是 192.168.5.11)：

```
[root@100ask:~]# tftp -p -l 1.txt  192.168.5.11
```

此时我们查看 Ubuntu 服务器的 tftp 服务目录下，即可看到之前在开发板上创建的 1.txt：

图 3.25 开发板传给 Ubuntu 的文件

第4章 开发板的第 1 个 APP 实验

详细的手把手教学视频请看
https://www.bilibili.com/video/BV1zV411U7H9?p=10
交叉编译运行第一个 APP。

4.1 获取程序

Git 仓库里含有本教程的所有源码，前面已经在 Windwos 下载了 Git 仓库，为例方便编译，也可以在 Ubuntu 中再次下载它。

在 Ubuntu 终端上执行如下命令。

```
git clone https://e.coding.net/weidongshan/01_all_series_quickstart.git
```

代码获取示意图如图 4.1 所示。

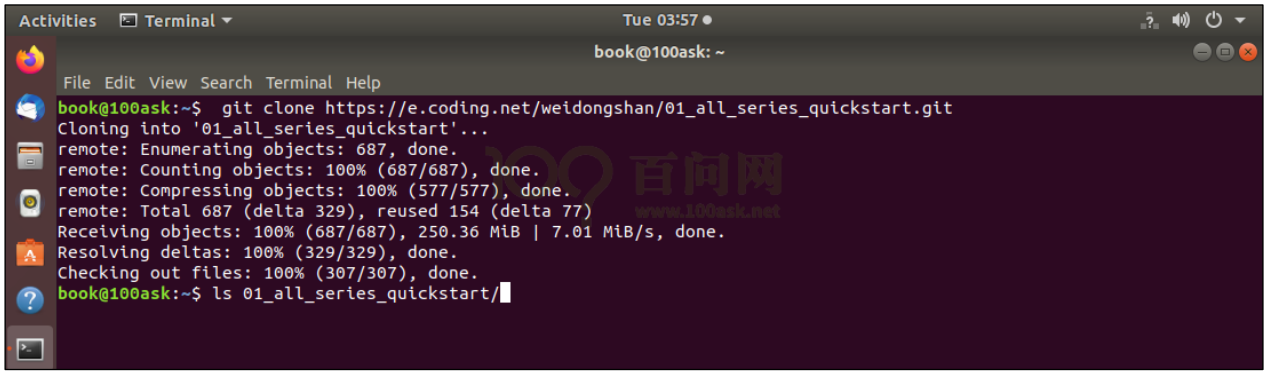


图 4.1 使用 Git 获取资料

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\source\01_hello
```

注意：如果已经使用 GTI 下载过源码，就不需要重复下载了，否则会有如图 4.2 提示：

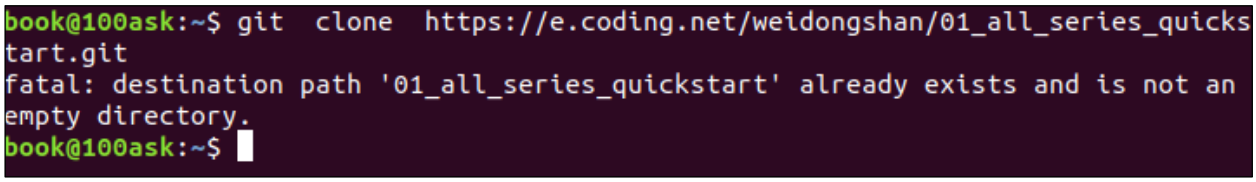


图 4.2 重复 clone 的提示

hello.c 的源码如下：

```
01 #include <stdio.h>
02
03 /* 执行命令: ./hello weidongshan
04 * argc = 2
05 * argv[0] = ./hello
06 * argv[1] = weidongshan
07 */
08
09 int main(int argc, char **argv)
10 {
11     if (argc >= 2)
```

```
12          printf("Hello, %s!\n", argv[1]);
13      else
14          printf("Hello, world!\n");
15      return 0;
16  }
```

4.2 编译程序

在 Ubuntu 中可以执行以下命令编译、执行：

```
gcc -o hello hello.c
./hello
Hello, world!
./hello weidongshan
Hello, weidongshan!
```

上述命令编译得到的可执行程序 hello 可以在 Ubuntu 中运行，但是如果把它放到 ARM 板子上去，它是无法执行的。因为它是使用 gcc 编译的，是给 PC 机编译的，里面的机器指令是 x86 的。

我们要想给 ARM 板编译出 hello 程序，需要使用交叉编译工具链。请参考《2.6.3 配置交叉编译工具链》。

执行以下命令编译程序：

```
arm-buildroot-linux-gnueabihf-gcc -o hello hello.c
```

这样编译出来的 hello 程序才可以在 ARM 板子上运行。先把编译生成的 **hello** 文件拷贝到 Ubuntu nfs 服务目录下，备用：

```
cp  hello   /home/book/nfs_rootfs
```

4.3 上传程序到开发板并运行

开发板启动后通过 nfs 挂载 Ubuntu 目录的方式，将相应的文件拷贝到开发板上。假设 Ubuntu IP 为 192.168.5.11，在开发板上执行以下命令：

```
[root@100ask:~]# mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
[root@100ask:~]# cp  /mnt/hello  ./hello
```

最后，执行如下操作添加可执行权限，并运行它：

```
[root@100ask:~]# chmod +x hello
[root@100ask:~]# ./hello
```

第5章 开发板的第 1 个驱动实验

详细的手把手教学视频请看
https://www.bilibili.com/video/BV1zV411U7H9?p=12
单独编译加载第一个驱动。

5.1 前提

为什么编译驱动程序之前要先编译内核？

- 驱动程序要用到内核文件：

比如驱动程序中这样包含头文件：#include <asm/io.h>，其中的 asm 是一个链接文件，指向 asm-arm 或 asm-mips，这需要先配置、编译内核才会生成 asm 这个链接文件。

- 编译驱动时用的内核、开发板上运行到内核，要一致：

开发板上运行到内核是出厂时烧录的，你编译驱动时用的内核是你自己编译的，这两个内核不一致时会导致一些问题。所以我们编译驱动程序前，要把自己编译出来到内核放到板子上去，替代原来的内核。

- 更换板子上的内核后，板子上的其他驱动也要更换：

板子使用新编译出来的内核时，板子上原来的其他驱动也要更换为新编译出来的。所以在编译我们自己的第 1 个驱动程序之前，要先编译内核、模块，并且放到板子上去。

5.2 编译内核

不同的开发板对应不同的配置文件，配置文件位于内核源码 arch/arm/configs/目录。kernel 的编译过程如下：

```
cd /home/book/100ask_imx6ull-sdk/Linux-4.9.88
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make mrproper
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make 100ask_imx6ull_defconfig
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make zImage  -j4
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make dtbs
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ cp arch/arm/boot/zImage ~/nfs_rootfs
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ cp arch/arm/boot/dts/100ask_imx6ull-14x14.dtb  ~/nfs_rootfs
```

编译步骤参考如图 5.1，编译完成 zImage 后才可编译设备树文件。

```
book@100ask:~/100ask_imx6ull-sdk$ cd Linux-4.9.88/
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make mrproper
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$  make 100ask_imx6ull_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make zImage -j4
```

图 5.1 编译 zImage

编译完成后生成的文件如图 5.2 所示：

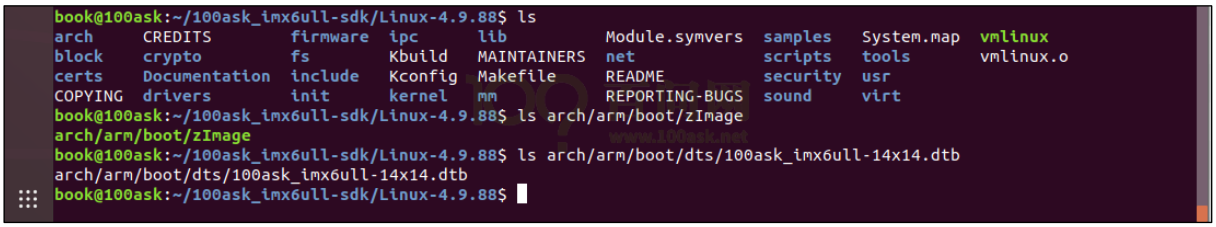


图 5.2 编译 zImage 完成

编译完成后，在 arch/arm/boot 目录下生成 zImage 内核文件，在 arch/arm/boot/dts 目录下生成设备树的二进制文件 **100ask_imx6ull-14x14.dtb**。把这 2 个文件复制到/home/book/nfs_rootfs 目录下备用，如图 5.3：

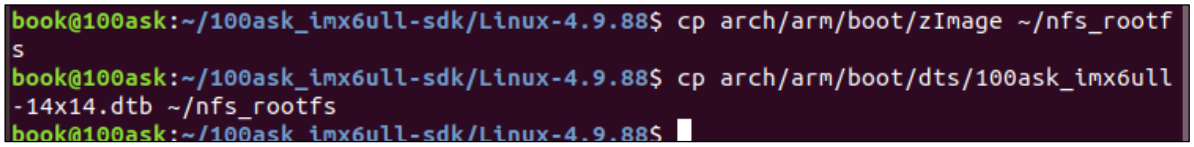


图 5.3 将 zImage 和 dtb 通过 NFS 复制到开发板

5.3 编译安装内核模块

5.3.1 编译内核模块

进入内核源码目录后，就可以编译内核模块了：

```
book@100ask: cd ~/100ask_imx6ull-sdk/Linux-4.9.88/
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make  modules
```

内核模块编译完成后如图 5.4 所示：

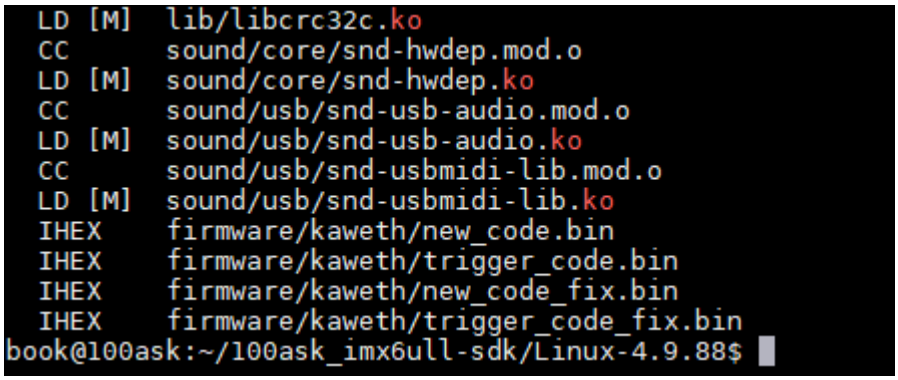


图 5.4 内核模块编译完成

5.3.2 安装内核模块到 Ubuntu 某个目录下备用

可以先把内核模块安装到 nfs 目录(/home/book/nfs_rootfs)。

注意：后面会使用 tree 命令查看目录结构，如果提示没有该命令，需要执行以下命令安装 tree 命令：

```
sudo apt install tree
```

执行以下命令安装模块：

```
book@100ask:~$ cd ~/100ask_imx6ull-sdk/Linux-4.9.88/
```

```
book@100ask:~/100ask_imx6ull-sdk/Linux-4.9.88$ make  ARCH=arm INSTALL_MOD_PATH=/home
/book/nfs_rootfs  modules_install
```

如图 5.5，把模块安装在 nfs 目录“/home/book/nfs_rootfs/”下：

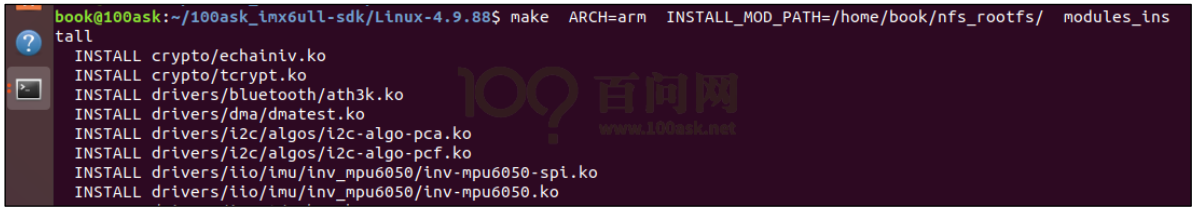


图 5.5 安装模块到 nfs_rootfs

安装好驱动后的/home/book/nfs_rootfs/目录结构如图 5.6 所示：

图 5.6 安装模块后的 nrf_rootfs

5.4 安装内核和模块到开发板上

假设：在 Ubuntu 的/home/book/nfs_rootfs 目录下，已经有了 zImage、dtb 文件，并且有 lib/modules 子目录(里面含有各种模块)。

接下来要把这些文件复制到开发板上。假设 Ubuntu IP 为 192.168.5.11，在开发板上执行以下命令：

```
mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
cp  /mnt/zImage  /boot
cp  /mnt/100ask_imx6ull-14x14.dtb   /boot
cp  /mnt/lib/modules  /lib  -rfd
sync
```

最后重启开发板，它就使用新的 zImage、dtb、模块了。

5.5 编译 led 驱动

led 驱动在 GIT 仓库里，如果没有使用 Git 克隆资料到本地，请看《>>>>>>>> 第三篇第 4 章 4.1 获取程序》目录位置如下：

```
01_all_series_quickstart/
  05_嵌入式 Linux 驱动开发基础知识/
    source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin
```

可以在 Ubuntu 中下载 Git 仓库，或者从 Windows 上通过 filezilla 把源码传到 Ubuntu。

首先，进入 100ask_imx6ull_src_bin 目录，修改 Makefile 文件“KERN_DIR”为自己的内核所在路径。如图 5.7 红框所示，如果你的内核源码不在此目录则根据你的实际情况进行修改：

```
Makefile                                                                    buffers
# 1. 使用不同的开发板内核时，一定要修改KERN_DIR
# 2. KERN_DIR中的内核要事先配置、编译，为了能编译内核，要先设置下列环境变量:
# 2.1 ARCH,          比如: export ARCH=arm64
# 2.2 CROSS_COMPILE, 比如: export CROSS_COMPILE=aarch64-linux-gnu-
# 2.3 PATH,          比如: export PATH=$PATH:/home/book/100ask_roc-rk3399-pc/ToolChain-6.3.1/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin
# 注意: 不同的开发板不同的编译器上述3个环境变量不一定相同,
#       请参考各开发板的高级用户使用手册

KERN_DIR = /home/book/100ask_imx6ull-sdk/Linux-4.9.88

all:
        make -C $(KERN_DIR) M=`pwd` modules
        $(CROSS_COMPILE)gcc -o ledtest ledtest.c

clean:
        make -C $(KERN_DIR) M=`pwd` modules clean
        rm -rf modules.order
        rm -f ledtest

# 参考内核源码drivers/char/ipmi/Makefile
# 要想把a.c, b.c编译成ab.ko, 可以这样指定:
# ab-y := a.o b.o
# obj-m += ab.o

# leddrv.c board_demo.c 编译成 100ask.ko
100ask_led-y := leddrv.o board_100ask_imx6ull.o
obj-m   += 100ask_led.o
```

图 5.7 修改源码 SDK 路径

注意：有些学员并不按照前面的文档编译内核，而是使用 Buildroot 来编译内核，那么 led 驱动程序的 Makefile 中要指定 KERN_DIR 为如下目录：

```
KERN_DIR = /home/book/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/build/linux-origin_master
```

修改完内核所在目录后，就可以编译 led 模块驱动了，如图 5.8 红框 1 所示，执行“`make all`”命令就可以编译，编译完成后会生成 **100ask_led.ko ledtest** 两个文件。

```
book@book-virtual-machine:~/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin$ make all    1
make -C /home/book/100ask_imx6ull-sdk/Linux-4.9.88 M=`pwd` modules
make[1]: Entering directory '/home/book/100ask_imx6ull-sdk/Linux-4.9.88'
  CC [M]  /home/book/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin/leddrv.o
  CC [M]  /home/book/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin/board_100ask_imx6ull.o
  LD [M]  /home/book/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin/100ask_led.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/book/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin/100ask_led.mod.o
  LD [M]  /home/book/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin/100ask_led.ko
make[1]: Leaving directory '/home/book/100ask_imx6ull-sdk/Linux-4.9.88'
arm-linux-gnueabihf-gcc -o ledtest ledtest.c
book@book-virtual-machine:~/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin$ ls 100ask_led.ko ledtest -la
-rw-rw-r-- 1 book book  8240 2月  18 17:36 100ask_led.ko
-rwxrwxr-x 1 book book 10564 2月  18 17:36 ledtest    2
book@book-virtual-machine:~/Git/01_all_series_quickstart/04_快速入门_正式开始/02_嵌入式Linux驱动开发基础知识/source/02_led_drv/02_led_drv_for_boards/100ask_imx6ull_src_bin$
```

图 5.8 编译模块驱动

此时，把这两个文件拷贝到 **Ubuntu nfs** 目录下备用：

```
cp 100ask_led.ko ledtest ~/nfs_rootfs
```

5.6 在开发板安装驱动模块

5.6.1 下载驱动程序

开发板启动后通过 nfs 挂载 Ubuntu 目录的方式，将相应的文件拷贝到开发板内。如果你

使用的是 VMware 桥接方式，假设 Ubuntu IP 为 192.168.5.11，在开发板上执行以下命令：

```
[root@100ask:~]# mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
[root@100ask:~]# cp  /mnt/100ask_led.ko    ./
[root@100ask:~]# cp  /mnt/ledtest    ./
```

5.6.2 关闭系统默认 cpu 状态灯

可能开发板上的 led 灯已被设置为 CPU 状态灯(可以看到绿色 LED 在闪)，我们需要将其关闭，才可以使用我们的驱动对其进行操作，关闭方法如下所示：

```
[root@100ask:~]# echo none >  /sys/class/leds/cpu/trigger
```

我们后面出厂的开发板，已经把 CPU 状态灯禁止了，所以上述命令可能出错：这没影响。

5.6.3 安装驱动模块

在开发板串口终端上执行如下命令，即可安装相应的驱动模块。

```
[root@100ask:~]# insmod 100ask_led.ko
```

安装完成后可以执行 lsmod 命令来查看是否安装成功，如图 5.9 所示：

```
[root@100ask:~]# lsmod
Module                  Size  Used by
100ask_led              3721  0
```

图 5.9 执行 lsmod 验证

如果你没有更新板子上的内核，会出现类似如图 5.10 所示错误：

```
[ root@100ask :/mnt]# insmod hello_drv.ko
[   46.450554] hello_drv: loading out-of-tree module taints kernel.
[   46.457500] hello_drv: disagrees about version of symbol device_create
[   46.464110] hello_drv: Unknown symbol device_create (err -22)
[   46.472176] hello_drv: disagrees about version of symbol device_destroy
[   46.479475] hello_drv: Unknown symbol device_destroy (err -22)
insmod: ERROR: could not insert module hello_drv.ko: Invalid parameters
[ root@100ask :/mnt]#
```

图 5.10 不更新内核时的错误提示

可以强行安装驱动程序，比如使用“insmod -f hello_drv.ko”这样的命令，它会提示说“内核已经被污染了”，但是不影响学习、不影响使用，如图 5.11 所示：

```
[ root@100ask   :/mnt]# insmod -f hello_drv.ko
[  201.965408] hello_drv: module_layout: kernel tainted.
[  201.970545] Disabling lock debugging due to kernel taint
[ root@100ask   :/mnt]#
```

图 5.11 内核污染提示

5.7 执行测试程序

驱动模块安装成功后，就可以使用测试程序来控制 led 灯的状态，如图 5.12 所示，操作 led 灯时可同时观察开发板串口旁的灯是否有亮灭的变化。

```
[root@100ask:~]# chmod  +x  ./ledtest
[root@100ask:~]# ./ledtest
Usage: ./ledtest <dev> <on | off>
[root@100ask:~]# ./ledtest /dev/100ask_led0 on    //打开 led0 灯
```

```
[root@100ask:~]# ./ledtest /dev/100ask_led0 off   //关闭 led0 灯
[root@100ask:~]# ./ledtest
Usage: ./ledtest <dev> <on | off>
[root@100ask:~]# ./ledtest /dev/100ask_led0 on
[root@100ask:~]# ./ledtest /dev/100ask_led0 off
```

图 5.12 操控 LED

第6章 构建 bootloader、内核、文件系统

6.1 准备

Linux 平台上有许多开源的嵌入式 linux 系统构建框架(框架的意思就是工具)，这些框架极大的方便了开发者进行嵌入式系统的定制化构建，目前比较常见的有 OpenWrt，Buildroot，Yocto,等等。其中 Buildroot 功能强大，使用简单，而且采用了类似于 linux kernel 的配置和编译框架，所以受到广大嵌入式开发人员的欢迎。

本章重点介绍使用 Buildroot_2019.02LTS 版构建文件系统和 u-boot，kernel 镜像的方法，并从这三个部分入手，描述如何使用 Buildroot 构建一个适合 100ask_imx6ull 系列开发板的嵌入式 Linux 系统。

在构建文件系统时，还简要介绍了如何通过 Buildroot 将 QT5 图形系统集成到文件系统中，方便用户后续开发 QT5 的应用程序。

确保 Ubuntu 能 ping 通外网(比如：ping news.qq.com)，如果无法访问外网请按以下章节配置 Ubuntu：

- 《>>>>>>>>第三篇 1.1 添加 NAT 网卡》

然后参考以下章节下载源码、配置工具链：

- 《>>>>>>>>第三篇 2.6 下载 BSP 及配置工具链》

6.2 解压编译 bootloader

6.2.1 Bootloader 介绍

Bootloader 是在操作系统运行之前运行的一段代码，用于引导操作系统。通常每个操作系统都有一组专属的引导加载程序。引导加载程序通常可以通过多种方式引导操作系统内核，还有各种命令用于调试或修改内核运行环境。

U-Boot 是一个开源的主引导加载程序，用于引导设备的操作系统内核，并含有多种命令以便调试系统。它适用于多种计算机体系结构，包括 68k，ARM，Blackfin，MicroBlaze，MIPS，Nios，SuperH，PPC，RISC-V 和 x86。

U-boot 官网：

https://www.denx.de/wiki/U-Boot

源码下载页面：

http://ftp.denx.de/pub/u-boot/

NXP 官方 uboot 源码 Git 地址：

https://source.codeaurora.org/external/imx/uboot-imx

本开发使用的 U-boot 位于 Git 仓库，地址为：

https://e.coding.net/weidongshan/imx-uboot2017.03.git

注意：我们使用的版本针对板子进行过修改，u-boot 官网下载的源码不能直接使用。

6.2.2 编译 u-boot 镜像

不同的开发板对应不同的配置文件，配置文件位于 u-boot 源码的“configs/”目录。对于 IMX6ULL Pro 版，u-boot 的编译过程如下(编译 uboot 前必须先配置好工具链等开发环境)：

```
cd /home/book/100ask_imx6ull-sdk
book@100ask: ~/100ask_imx6ull-sdk$cd  Uboot-2017.03
book@100ask: ~/100ask_imx6ull-sdk/Uboot-2017.03$ make distclean
book@100ask: ~/100ask_imx6ull-sdk/Uboot-2017.03$ make mx6ull_14x14_evk_defconfig
book@100ask: ~/100ask_imx6ull-sdk/Uboot-2017.03$ make
```

图 6.1 编译镜像文件为具体的执行步骤：

图 6.1 编译镜像文件

编译完成后生成的文件如图 6.2 所示。

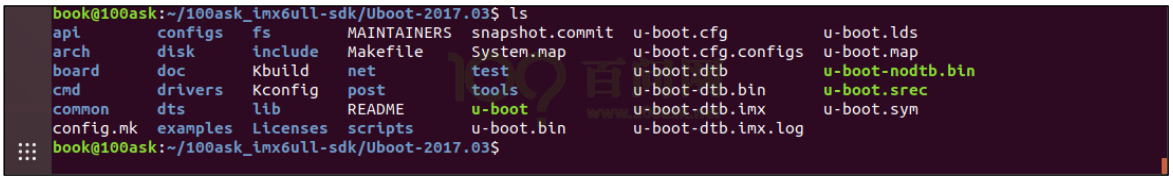


图 6.2 镜像文件编译成功

编译完成之后生成 u-boot-dtb.imx，可以烧在 TF 卡、EMMC 上。以下命令是将 u-boot-dtb.imx 文件烧写到 EMMC 上：

```
[root@100ask:~]# echo 0 > /sys/block/mmcblk1boot0/force_ro
[root@100ask:~]# dd if=u-boot-dtb.imx of=/dev/mmcblk1boot0 bs=512 seek=2
[root@100ask:~]# echo 1 > /sys/block/mmcblk1boot0/force_ro
```

6.3 编译 Linux Kernel 和模块

Linux 内核（英语：Linux kernel）是一种开源的类 Unix 操作系统宏内核。整个 Linux 操作系统家族基于该内核部署在传统计算机平台（如个人计算机和服务器，以 Linux 发行版的形式）和各种嵌入式平台，如路由器、无线接入点、专用小交换机、机顶盒、FTA 接收器、智能电视、数字视频录像机、网络附加存储（NAS）等。工作于平板电脑、智能手机及智能手表的 Android 操作系统，它的底层操作系统也是 Linux。尽管在桌面计算机的占用率较低，但基于 Linux 的操作系统统治了几乎从移动设备到主机的其他全部领域。实际 Linux 的发行版 Ubuntu，其易用性也逐渐接近 Windows。

Linux kernel 官网：

https://www.kernel.org/

linux Kernel 维基百科：

https://www.wiki.kernel.org/

在线阅读 linux kernel 源码：

https://elixir.bootlin.com/

Git 仓库地址：

https://e.coding.net/weidongshan/imx-linux4.9.88.git

NXP 官方 kernel 源码 Git 仓库地址：

https://source.codeaurora.org/external/imx/linux-imx

上述 Git 仓库是专为 100ask_imx6ull 系列开发板制定的 Linux 内核，它有如表 6-1 特性：

表 6-1 百问定制内核特征

名称	是否支持	作用	是否开源
USB HOST	支持	USB Host 驱动，支持 OHCI 和 EHCI 两种传输模式	Yes
USB OTG	支持	Usb otg 驱动，支持 device 模式和 host 模式	Yes
Network	支持	RJ-45 以太网驱动	Yes
MMC/SD	支持	MMC/SD 卡驱动	Yes
EMMC	支持	EMMC 驱动	Yes
I2C	支持	I2C 驱动	Yes
SPI	支持	SPI 驱动	Yes
LCD	支持	显示驱动	Yes
RTC	支持	内置 RTC 时钟驱动	Yes
ADC	支持	ADC 驱动	Yes
TouchScreen	支持	电容触摸	Yes
UART	支持	串口驱动	Yes
PMU	支持	电源管理驱动	Yes
LED	支持	LED 驱动，包括 GPIO LED 和 PWM	Yes
Button	支持	GPIO Button 驱动	Yes
Camera	支持	摄像头驱动	Yes

使用内核以及内核模块前需要将其编译，然后安装到 ubuntu 或开发板，详细步骤已经下面 3 个章节写明了：

- 《>>>>>>>>第三篇第 5 章 5.2 编译内核》
- 《>>>>>>>>第三篇第 5 章 5.3 编译安装内核模块》
- 《>>>>>>>>第三篇第 5 章 5.4 安装内核和模块到开发板上》

6.4 Buildroot 的用法

6.4.1 Buildroot 简介

制作根文件系统有很多种方法：

① 使用 Busybox 手工制作

Busybox 本身包含了很了 Linux 命令，但是要编译其他程序的话需要手工下载、编译，如果它需要某些依赖库，你还需要手工下载、编译这些依赖库。如果想做一个极简的文件系统，可以使用 Busybox 手工制作。

② 使用 Buildroot 自动制作

它是一个自动化程序很高的系统，可以在里面配置、编译内核，配置编译 u-boot、配置编译根文件系统。在编译某些 APP 时，它会自动去下载源码、下载它的依赖库，自动编译这些程序。

Buildroot 的语法跟一般的 Makefile 语法类似，很容易掌握。

③ 使用 Yocto

NXP、ST 等公司的官方开发包是使用 Yocto，但是 Yocto 语法复杂，并且 Yocto 动辄 10GB，下载安装都很困难，普通笔记本编译可能需要 2-3 天甚至更久，非常不适合初学者(我们不推荐使用 yocto 构建文件系统)。

基于上述特点，我们选择 Buildroot。

Buildroot 是一组 Makefile 和补丁，可简化并自动化地为嵌入式系统构建完整的、可启动的 Linux 环境(包括 bootloader、Linux 内核、包含各种 APP 的文件系统)。Buildroot 运行于 Linux 平台，可以使用交叉编译工具为多个目标板构建嵌入式 Linux 平台。Buildroot 可以自动构建所需的交叉编译工具链，创建根文件系统，编译 Linux 内核映像，并生成引导加载程序用于目标嵌入式系统，或者它可以执行这些步骤的任何独立组合。例如，可以单独使用已安装的交叉编译工具链，而 Buildroot 仅创建根文件系统。

Buildroot 参考网址：

•Buildroot 用户手册：

https://buildroot.org/downloads/manual/manual.html

• Buildroot 源码下载地址：

https://buildroot.org/downloads/

• 百问网提供的 Buildroot 现支持如下包：

表 6-2 百问网 buildroot 支持包

busybox	taglib	libopenh264	qt5_gl_available
skeleton_init_systemd	tremor	libopusenc	qt5_jscore_available
alsa_utils	libarchive	libtheora	qt5
bluez_alsa	zlib	libvpx	qt5_version_latest
faad2	libzlib	libyuv	qt5base
ffmpeg	has_zlib	live555	qt5base_network
flac	libassuan	mediastreamer	qt5base_concurrent
fluidsynth	libgcrypt	x264	qt5base_sql

gstreamer1	libgpg_error	x264_cli	qt5base_sqlite_none
jack2	Libgpgme	libcurl	qt5base_test
lame	Libksba	libnice	qt5base_xml
mjpegtools	Openssl	libnl	qt5base_gui
musepack	libopenssl	libpcap	qt5base_widgets
pulseaudio	sqlite	libshout	qt5base_opengl
twolame	libconfig	libsoup	qt5base_opengl_es2
wavpack	libconfuse	libsrtp	qt5base_opengl_lib
bzip2	libfuse	libtirpc	qt5base_linuxfb
lz4	liblockfile	neon	qt5base_eglfs
xz	bayer2rgb_neon	ortp	qt5base_default_qpa=""
gdb	bullet	rtmpdump	qt5base_printsupport
gdb_server	cairo	bctoolbox	qt5base_fontconfig
gdb_debugger	cairo_pdf	elfutils	qt5base_harfbuzz
gdb_tui	cairo_png	glibmm	qt5base_gif
spidev_test	cairo_svg	libb64	qt5base_jpeg
strace	fontconfg	libbsd	qt5base_png
libtool	freetype	libcap	qt5base_dbus
cifs_utils	gdk_pixbuf	libcroco	qt5base_icu
dosfstools	giflib	libevdev	qt5base_openssl
e2fsprogs	harfbuzz	libffi	qt5base_tslib
mmc_utils	jasper	libglib2	qt5canvas3d
nfs_utils	jpeg	libical	qt5charts

glmark2_flavor_any	kmsxx	libnpth	qt5connectivity
libva_utils	libdrm	libsigc	qt5declarative
fb_test_app	libmediaart	liburcu	qt5declarative_quick
fbdump	libpng	llvm	qt5enginio
fbset	libraw	mpdecimal	qt5graphicaleffects
	librsvg	protobuf	qt5imageformats
psplash	libsvg	tzdata	qt5location
xkeyboard_config	libsvg_cairo	icu	qt5multimedia
dbus	libva	libfribidi	qt5quickcontrols
evtest	opencv3	ncurses	qt5quickcontrols2
	has_libegl	pcre	qt5script
gptfdisk	has_libgles	pcre2	qt5scxml
i2c_tools	openjpeg	popt	qt5sensors
iostat	pango	readline	qt5serialbus
linux_serial_test	pipewire	slang	qt5serialport
memtool	pixman	bluez_tools	qt5svg
parted	tiff	bluez5_utils	qt5virtualkeyboard
spi_tools	wayland	bmon	qt5webchannel
sysstat	webp	iperf3	qt5websockets
uboot_tools	dtc	iproute2	qt5xmlpatterns
has_udev	libaio	iw	attr
usbmount	libgudev	lrzsz	keyutils
usbutils	libinput	mii_diag	kmod
luajit	libpciaccess	mjpg_streamer	kmod_tools

python3	libusb	mosquitto	start_stop_daemon
alsa_lib	libxkbcommon	net_tools	swupdate
fdk_aac	mtdev	openobex	systemd
libcodec2	tslib	openssh	util_linux
libcuefile	expat	pppd	myir_hmi_gui
libreplaygain	json_glib	rpcbind	rs485read
libsamplerate	jsoncpp	tcpdump	
libsndfile	libxml2	wget	
libvorbis	bitstream	wireless_tools	
mp4v2	libass	opkg	
opus	libcamera	bash	
portaudio	libmms	gnupg2	
sbc	libmpeg2	gnupg2_gpgv	
speex	libogg	lockfile_progs	

6.4.2 系统特征讲解

1 我们都做了哪些改进？

我们基于 buildroot 官方 2020.02 长期支持版本进行适配 imx6ull 开发板，在此基础上针对 ST yocto 发行系统做了大量的裁剪，在保证最小系统的基础上增加对 qt5.12 库的支持，同时也支持 opencv3 编程 mqtt 库 swupdate ota 升级等等比较常用的应用，我们也会提供如何在 buildroot 新增自己的软件包教程，同时源码保存在 gitee 上，链接地址为

https://gitee.com/weidongshan/Buildroot.git

大家可以在上面提交 issue 或者 pull request。

2 什么是 init 系统服务？

init（为英语：initialization 的简写）是 Unix 和 类 Unix 系统中用来产生其它所有进程的程序。它以守护进程的方式存在，其进程号为 1。Linux 系统在引导时加载 Linux 内核后，便由 Linux 内核加载 init 程序，由 init 程序完成余下的引导过程，比如加载运行级别，加载服务，引导 Shell/图形化界面等等。

3 什么是 systemv 守护进程？

System V（缩写为 SysV）在大多数 Linux 发行版中使用最广泛,在 systemv 中，在内核加载后运行的第 1 个程序被称为 init 程序。Init 做一些事情，其中之一就是加载一系列脚本来启动各种系统服务，例如网络，ssh 守护程序等。System V 中的运行级别描述了某些状态，例如：

- 运行级别 0：暂停
- 运行级别 1：单用户模式
- 运行级别 6：重新启动

systemv 的问题在于它需要仔细调整。假设您有一个要在启动时运行的网络文件系统（NFS）客户端。在网络正常工作之前运行 NFS 没有任何意义。因此，您必须等待网络已经正常工作，才能启动 systemv。Systemv init 这样做的方法是为服务启动设置严格的顺序。每个服务都分配有一个优先级编号，init 会按优先级顺序启动服务。如果需要确保网络连接后启动服务，则必须手动为服务分配更高的优先级。这必须由某人（通常是软件包维护者）针对计算机上运行的每项服务来完成。

所有 System V 初始化脚本都存储在/etc/rc.d/init.d/或/etc/init.d 目录中。这些脚本用于控制系统的启动和关闭。通常，您会在这个目录找到启动 ssh 服务器或网络的脚本,可以手工执行它们。比如使用“/etc/init.d/S50sshd start”开启 ssh 服务，使用“/etc/init.d/S50sshd stop”关闭 ssh 服务。

4 什么是 systemd 守护进程？

systemd 这一名字源于 Unix 中的一个惯例：在 Unix 中常以“d”作为系统守护进程（英语：daemon，亦称后台进程）的后缀标识。除此以外，systemd 亦是指代英文术语 D 体系，而这一术语即是用于描述一个人具有快速地适应环境并解决困难的能力。

systemd 是 Linux 电脑操作系统之下的一套中央化系统及设置管理程序（init），包括有守护进程、程序库以及应用软件，由 Lennart Poettering 带头开发。其开发目标是提供更优秀的框架以表示系统服务间的依赖关系，并依此实现系统初始化时服务的并行启动，同时达到降低 Shell 的系统开销的效果，最终代替现在常用的 System V 与 BSD 风格的 init 程序。目前绝大多数的 Linux 发行版都已采用 systemd 代替原来的 System V。

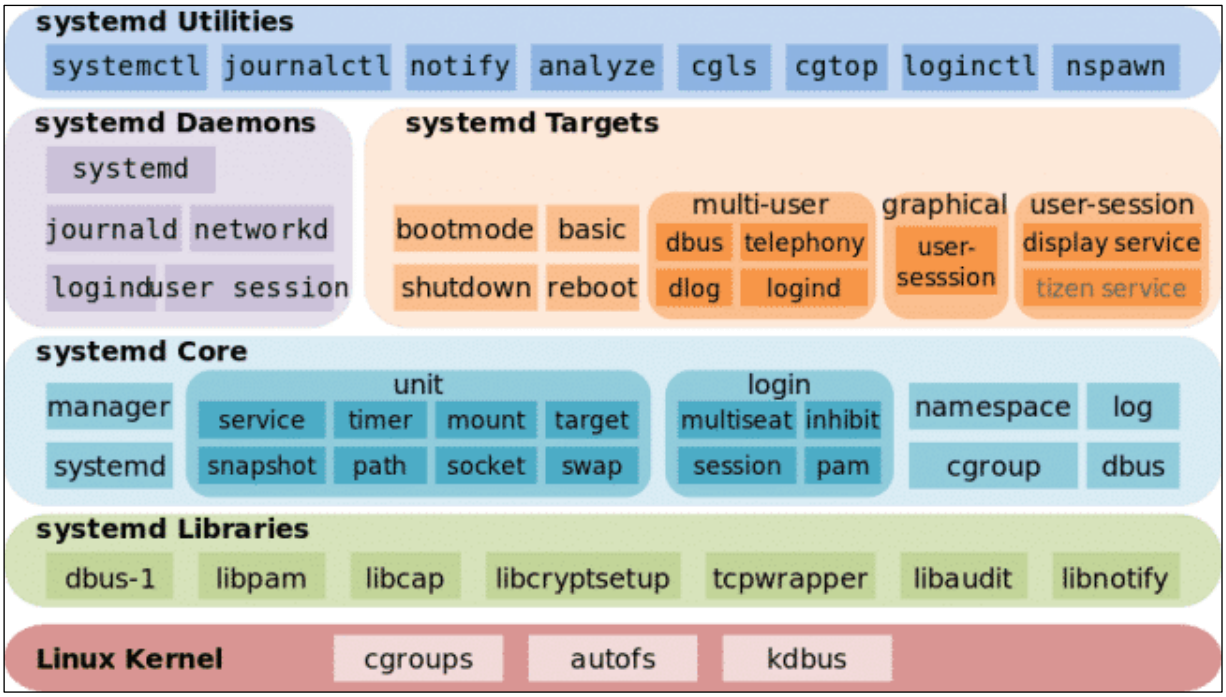


图 6.3 systemd

将 service（服务）、target（运行模式，类似于运行级别）、mount、timer、snapshot、path、socket、swap 等称为 Unit。比如，一个 auditd 服务（就是 auditd.service）就是一个 Unit，一个 multi-user.target 运行模式也是一个 Unit，其中不同的服务通过 systemctl 来进行统一管理，例如重启一个 sshd 服务，需要执行 systemctl restart sshd 命令，同样的如果添加一个启动程序需要自己定义一个 service 服务才可以。

5 你可以用我们提供的 buildroot 做些什么？

快速实现自己的想法，增加自己的应用，学习了解 busybox 以及各种包是如何编译生成，用于调试内核或者应用程序等等。

6.4.3 扩展学习

在 buildroot 下进入 menuconfig 包选择配置配置界面

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make menuconfig
```

buildroot 下单独编译内核

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make linux-rebuild
```

buildroot 下进入内核 make menuconfig 配置选项界面

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make linux-menuconfig
```

buildroot 下单独编译 u-boot

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make uboot-rebuild
```

buildroot 下单独编译某个软件包

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make <pkg>-rebuild
```

buildroot 下进入 busybox 配置界面

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make busybox-menuconfig
```

buildroot 下生成系统 sdk,最后生成的目录在 output/images/目录下

```
book@100ask: ~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make sdk
```

6.5 构建 IMX6ULL Pro 版的根文件系统

6.5.1 配置文件说明

表 6-3 配置文件说明

配置文件	含义
100ask_imx6ull_pro_ddr512m_systemV_core_defconfig	使用 systemV 守护进程的系统(包含基本工具集包)
100ask_imx6ull_pro_ddr512m_systemV_qt5_defconfig	使用 systemV 守护进程的系统，包含 Qt GUI 文件系统版本(出厂系统默认)

6.5.2 编译系统

下面以 100ask_imx6ull_pro_ddr512m_systemV_qt5_defconfig 配置文件为例，在 ubuntu 终端上说明 Buildroot 的配置过程：

```
cd /home/book/100ask_imx6ull-sdk
book@100ask:~/100ask_imx6ull-sdk$ cd Buildroot_2020.02.x
book@100ask:~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make clean
book@100ask:~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make 100ask_imx6ull_pro_ddr512m_systemV_qt5_defconfig
book@100ask:~/100ask_imx6ull-sdk/Buildroot_2020.02.x$ make all -j4
```

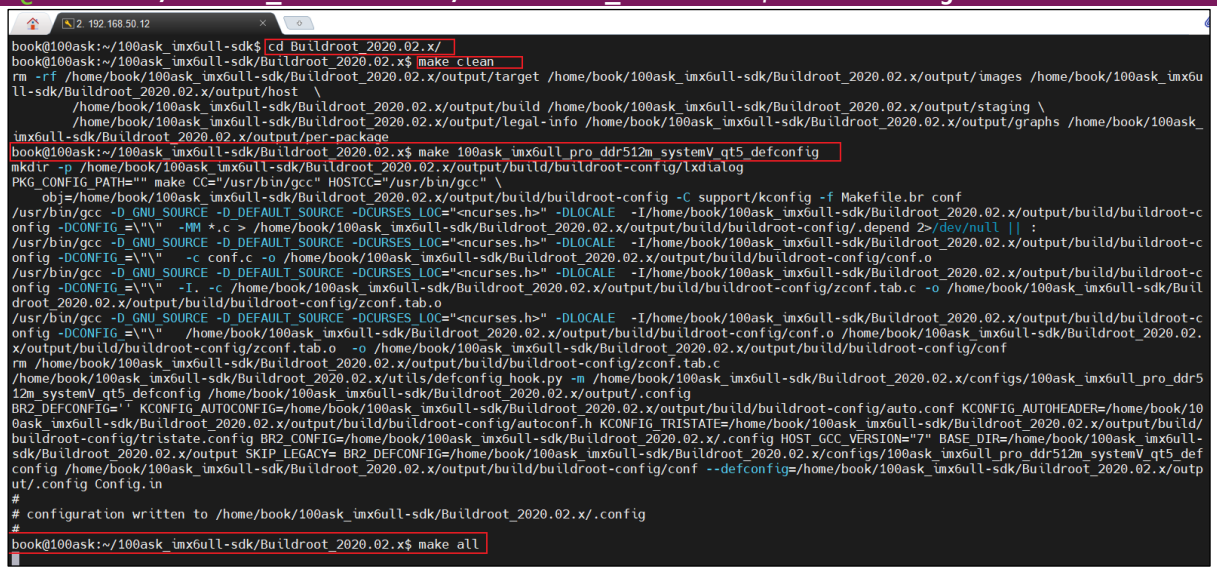


图 6.4 编译系统

注意：机器性能不同，编译时间不同。性能差的电脑，有可能需要等待 5 ~ 6 个小时。编译完成后如图 6.5 示。

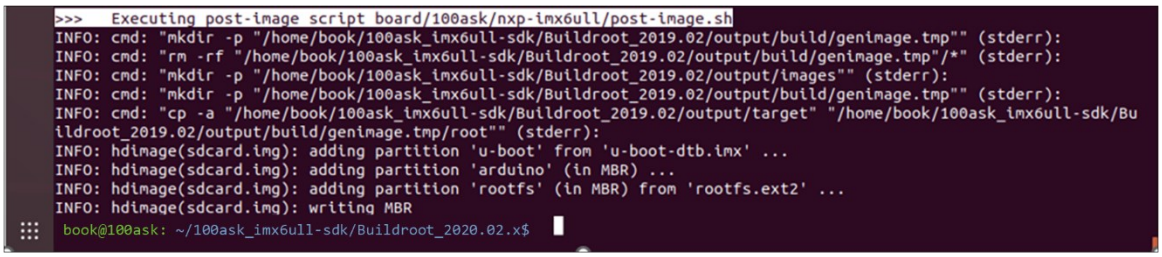


图 6.5 系统编译完成

6.5.3 镜像文件

编译成功后文件输出路径为 output/images

```
buildroot2020.02.x
    ├── output
        ├── images
            ├── 100ask_imx6ull-14x14.dtb <--设备树文件
            ├── rootfs.ext2              <--ext2 格式根文件系统
            ├── rootfs.ext4 -> rootfs.ext2      <--ext2 格式根文件系统
            ├── rootfs.tar
            ├── rootfs.tar.bz2           <--打包并压缩的根文件系统，用于
NFSROOT 启动
            ├── 100ask-imx6ull-pro-512d-systemv-v1.img          <--完整的系统
镜像(可以用来烧写 emmc 和 sd 卡)
            ├── u-boot-dtb.imx           <--u-boot 镜像
            └── zImage                  <--内核镜像
```

可以参考章节《烧写整个系统或更新部分系统》把上面编译出来的100ask-imx6ull-pro-512d-systemv-v1.img重命名为emmc.img(或sdcard.img),把emmc.img(或sdcard.img)放到“100ask_imx6ull 烧写工具”的 files 目录里，就可以使用工具烧写到 EMMC(或 SD/TF 卡)。

6.6 开发板使用 NFS 根文件系统

注意：所谓根文件系统就是类似 Windows 的 C 盘，里面存放有必须的 APP、库文件、配置文件。通过 NFS 可以把 Ubuntu 的某个目录，当作板子的“C 盘”——Linux 中称之为根文件系统。

Buildroot 编译完成之后生成的 rootfs.tar.bz2，可以解压之后放到 NFS 服务器上作为 NFS 文件系统供开发板使用。使用 NFS 文件系统，便于程序的开发调试。所谓 NFS 服务器，就是我们在 VMWare 上运行的 Ubuntu。

使用 NFS 根文件文件系统时，我们一般还会在 u-boot 使用 tftpboot 命令从 Ubuntu 或 Windows 中下载内核文件 zImage 和设备树文件。这时，Ubuntu 上既要配置 NFS 服务，也要配置 TFTP 服务。

请参考《第 3 章 IMX6ULL Pro 开发板基本操作》的章节确认 NFS、TFTP 服务可以使用。

6.6.1 准备好文件

使用 NFS 根文件系统时，涉及 3 个文件：zImage、设备树(比如 100ask_imx6ull-14x14.dtb)、rootfs.tar.bz2(解压到 Ubuntu 某个目录,比如/home/book/nfs_rootfs)。

在 U-Boot 中通过 tftpboot 命令从 Ubuntu/Windows 中下载内核文件 zImagen、100ask_imx6ull-14x14.dtb，设置 Uboot 启动参数使用 Ubuntu 的某个目录(比如/home/book/nfs_rootfs)作为根文件系统。

第4步 拷贝内核和设备树文件到 tftp 目录：

将出厂镜像或者自行编译的 zImage 和设备树文件 100ask_imx6ull-14x14.dtb，拷贝到 Ubuntu 的 /home/book/tftpboot 目录。

第5步 把根文件系统压缩包解压到 NFS 目录：

把使用 buildroot 构建得到的根文件系统 rootfs.tar.bz2(在 buildroot 系统 output/images 目录下)，复制、解压到 Ubuntu 的/etc/exports 文件中指定的目录里，即复制到/home/book/nfs_rootfs 目录下，并解压(注意：解压时要用 sudo)：

```
book@100ask:~/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/images$ cp -rf  rootfs.tar.bz2  ~/nfs_rootfs/
book@100ask:~/100ask_imx6ull-sdk/Buildroot_2020.02.x/output/images$ cd ~/nfs_rootfs
book@100ask:~/nfs_rootfs$ sudo tar  xjf  rootfs.tar.bz2
```

6.6.2 进入 uboot 界面

100ASK_IMX6ULL_Pro 开发板有 2 个网口，使用图 6.6 所示的 eth0：

图 6.6 开发板的两个网口

使用 mobaxterm 工具打开开发板串口，此时将开发板上电，在打印 u-boot 时按下任意键进入 uboot 界面。

图 6.7 进入 uboot

1 测试开发板与 Ubuntu/Windows 是否联通

查看 Ubuntu 系统 IP 地址，如图 6.8 所示在 Ubuntu 系统终端里使用 ifconfig 查看所有网络设备的信息，获取到当前桥接网卡设备的 IP 地址为 192.168.5.11：

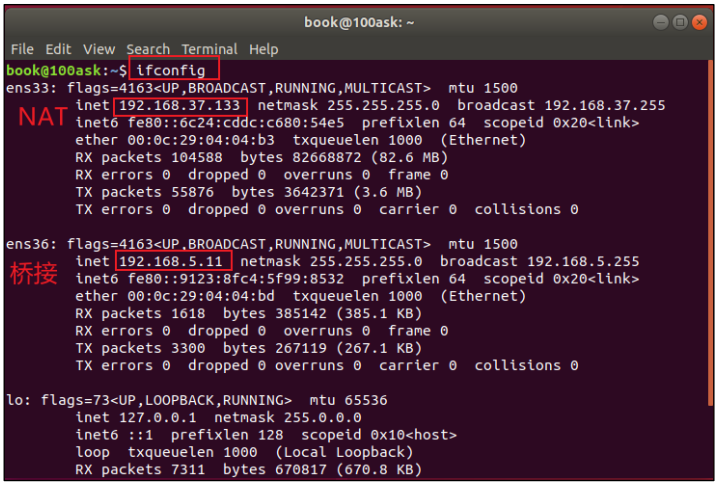


图 6.8 ifconfig 查看 ip

先在 u-boot 中设置开发板 IP 为 192.168.5.9,然后在 u-boot 中使用 ping 命令测试开发板与 Ubuntu 系统是否连通(出现“alive”就表示联通):命令如下:

```
=> setenv ipaddr 192.168.5.9
=> ping 192.168.5.11
```

如果提示 **host is alive** 就表示开发板和 Ubuntu 系统可以互通，如图 6.9 所示。

```
Warning: ethernet@020b4000 (eth1) using random MAC address - 1e:94:29:03:a2:61
ethernet@020b4000 Waiting for PHY auto negotiation to complete.... done
Using ethernet@020b4000 device
host 192.168.5.11 is alive
=>
```

图 6.9 设置 ip

2 使用网络启动文件系统

先在 Ubuntu 的 TFTP 目录中放入 zImage 和设备树文件，再在 /home/book/nfs_rootfs 目录下解压好根文件系统。

然后在 U-Boot 控制台执行以下命令启动单板，假设 Ubuntu IP 是 192.168.5.11:

```
=> setenv serverip 192.168.5.11      //设置服务器的 IP 地址，这里指的是 Ubuntu 主机 IP
=> setenv ipaddr 192.168.5.9         //设置开发板的 IP 地址。
=> setenv nfsroot /home/book/nfs_rootfs    //设置 nfs 文件系统所在目录。
=> run netboot              //设置完成后，运行网络启动系统命令
```

见下页图 6.10。

```
Hit any key to stop autoboot:  0
=> setenv serverip 192.168.1.244
=> setenv ipaddr 192.168.1.112
=> setenv nfsroot /home/book/nfs_rootfs
=> run netboot
Booting from net ...

Warning: ethernet@020b4000 (eth1) using random MAC address - ba:68:68:5a:49:58
ethernet@020b4000 Waiting for PHY auto negotiation to complete.... done
Using ethernet@020b4000 device
TFTP from server 192.168.1.244; our IP address is 192.168.1.112
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##########################################
         1.9 MiB/s
done
Bytes transferred = 7378144 (7094e0 hex)
Using ethernet@020b4000 device
TFTP from server 192.168.1.244; our IP address is 192.168.1.112
Filename '100ask_imx6ull-14x14.dtb'.
Load address: 0x83000000
Loading: ###
         1.4 MiB/s
done
Bytes transferred = 37659 (931b hex)
Kernel image @ 0x80800000 [ 0x000000 - 0x7094e0 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300c31a
Modify /soc/aips-bus@02200000/epdc@0228c000:status disabled
ft_system_setup for mx6

Starting kernel ...

** 96 printk messages dropped ** [    0.110734] OF: Duplicate name in lcdif@021c8000, renamed to "display#1"
```

图 6.10 网络启动文件系统

第7章 烧写整个系统或更新部分系统

详细的手把手教学视频请看

https://www.bilibili.com/video/BV1zV411U7H9?p=14

百问网开发了一款烧写软件：100ask_imx6ull_flashing_tool，它的界面如图 7.1：

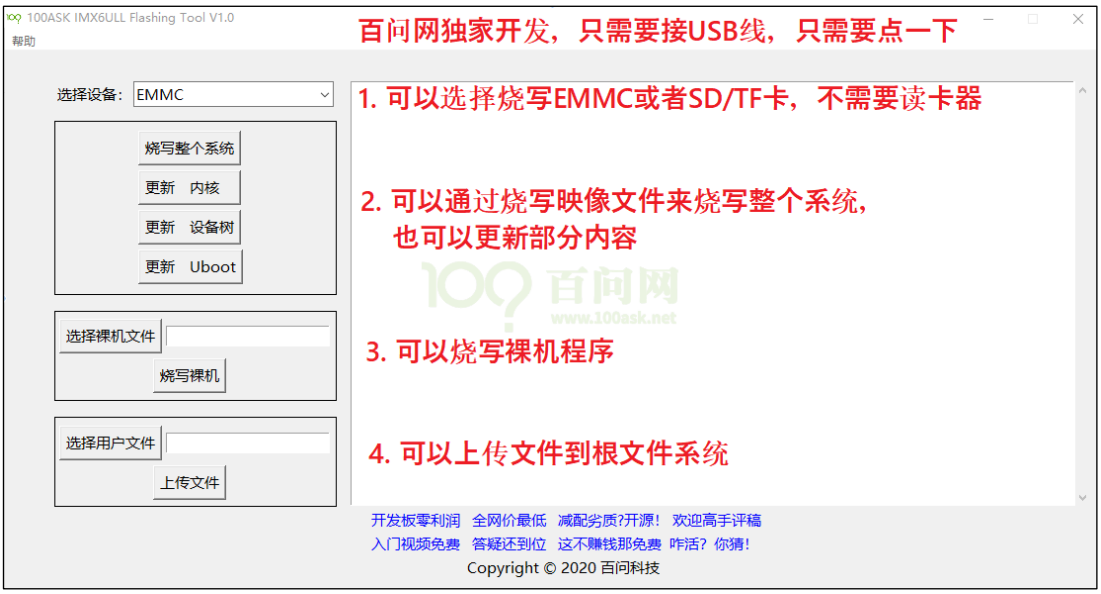


图 7.1 百问网烧写工具

使用这软件，只需要一条 USB 线连接电脑和开发板 USB OTG 口，只需要点击一个按钮就可以实现某项烧写功能。

100ask_imx6ull_flashing_tool 是一个 GUI 前端，它调用 NXP 官方的烧写工具 uuu：Universal Update Utility(又名 mfgtools 3.0)。

市面上的 imx6ull 开发板多是使用 mfgtools 2.0 来烧写，效率不高，脚本复杂；uuu 有所改进，但是烧写效率还是不高。

我们做了很多改进，烧写更方便、效率更高。使用 mfgtools 烧写 500M 的映象文件到 EMMC，耗时 5 分钟；而使用 100ask_imx6ull_flashing_tool 只需要 1 分钟。

本章会首先介绍 100ask_imx6ull_flashing_tool 的使用，后面也会介绍常规的烧写方法。

7.1 怎么更新整个系统或某些部件

如果你要重新烧写板子的整个系统，请参考：

- 《>>>>>>>>第三篇第 7 章 7.3 在 Windows 使用 USB 烧写工具》
- 《>>>>>>>>第三篇第 7 章 7.7 在 Windows 下烧写 SD/TF 卡》

如果你要单独更新 u-boot、内核、设备树，请参考：

- 《>>>>>>>>第三篇第 7 章 7.3 在 Windows 使用 USB 烧写工具》
- 《>>>>>>>>第三篇第 7 章 7.6 使用板子的 Linux 系统来烧写》

其他章节可以忽略，有必要时再看。

7.2 100ASK_IMX6ULL 启动开关

100ask_imx6ull Pro 版、mini emmc 版，都支持 USB、EMMC、SD/TF 卡三种启动方式。板子背后都画有一个表格，表示这 3 种方式如何设置。表格如下：

表 7-1 启动方式说明

	BOOT CFG			
BOOT	SW1(LCD_DATA5)	SW2(LCD_DATA11)	SW3(BOOT_MODE0)	SW4(BOOT_MODE1)
EMMC	OFF	OFF	ON	OFF
SD	ON	ON	ON	OFF
USB	X	X	OFF	ON

这 3 种启动方式的设置示意图如图 7.2：

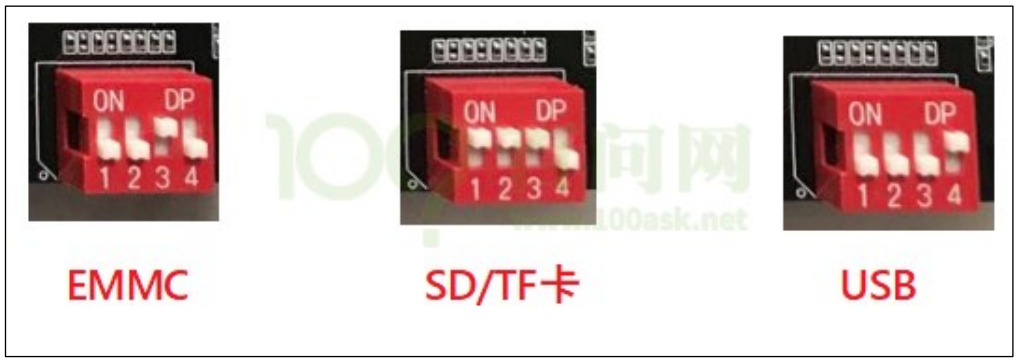


图 7.2 启动拨码开关

其中的 USB 启动模式主要用来烧写系统。

注意：设置为 USB 启动时，不能先插上 SD/TF 卡。要在 USB 启动模式下烧写 SD/TF 卡时，需要先上电再插卡，先上电再插卡，先上电再插卡，。

注意：使用后 EMMC、SD/TF 卡启动方式之前，需要先在 EMMC 或 SD/TF 卡上烧写系统。

7.3 在 Windows 使用 USB 烧写工具

NXP 公司给 IMX6ULL 提供了烧写工具：mfgtools，它的使用界面如图 7.3：

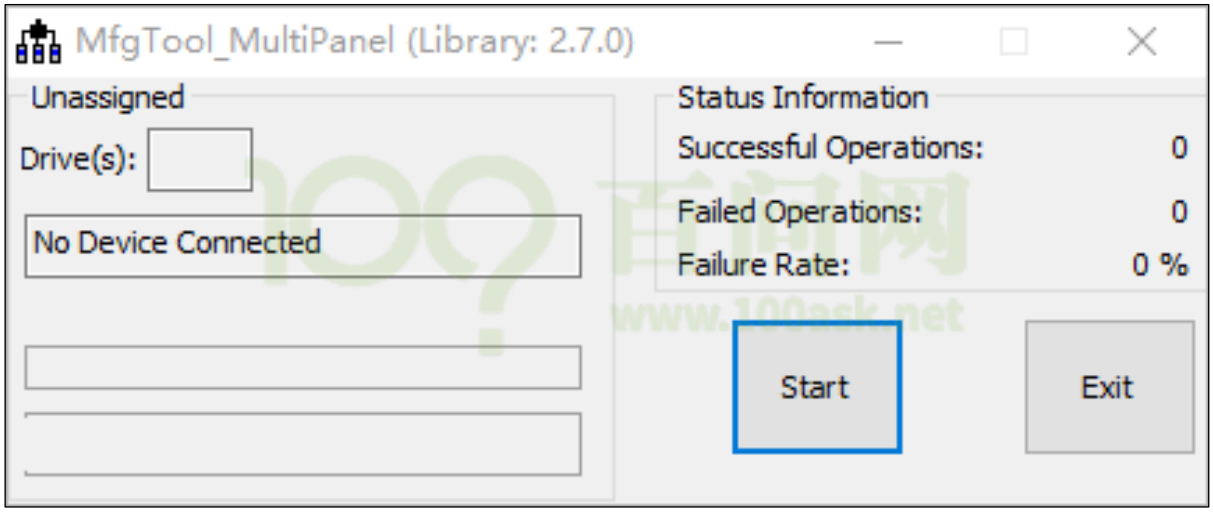


图 7.3 mfgtools

操作很简单，一键烧写整个映象文件。但是，缺点也很多：

④ 不能单独烧写 bootloader、内核、设备树或者说，可以实现这些功能，但是需要你去修改 xml 配置文件，对初学者不友善，对老手也显得麻烦。

⑤ 烧写速度慢，烧写 500M 的系统，耗时 5 分钟(我们的工具可以在 1 分钟内烧完)。

基于上述缺点，我们开发了新的烧写工具 100ask imx6ull flashing tool，并且完全开源。它有如下特点：

- 可以烧写整个系统，也可以分开烧写 bootloader、内核、设备树
- 可以上传用户文件到开发板系统中任意目录里
- 烧写速度是原厂工具的 5 倍
- 支持所有厂家的开发板烧写，注意：不只是支持 100ASK_IMX6ULL，还支持其他厂家的 IMX6ULL 开发板。

在“网盘的配套的资料/02_开发工具/100ask_imx6ull 烧写工具”目录下双击运行“100ask_imx6ull_flashing_tool.exe”。它有“基础版”、“专业版”两个页面，如图 7.4 所示

图 7.4 百问烧写工具界面说明

“基础版”是专为 100ASK_IMX6ULL 设计的，点击一下即可完成某项烧写。

“专业版”功能更强大，特别是它可以上传文件到某个分区、某个目录。有些厂家的开发板，zImage 和设备树是在第 1 个分区里的，而 100ASK_IMX6ULL 的 zImage 和设备树是在第 2 个分区里，所以这些厂家的开发板就无法使用基础版来烧写，需要使用专业版，指定分区、指定分区格式、指定路径，然后再上传文件。

注意：开发板必须设置为 USB 模式，如果要用 SD/TF 卡，必须先上电再插卡；不能先插卡再上电，不能先插卡再上电，不能先插卡再上电！

7.3.1 烧写工具目录详解

在 100ASK_IMX6ULL 的配套资料中，有文件：“02_开发工具/100ask_imx6ull_pro 开发板系统烧写工具.zip”，把它解压后可得如图 7.5 所示目录：

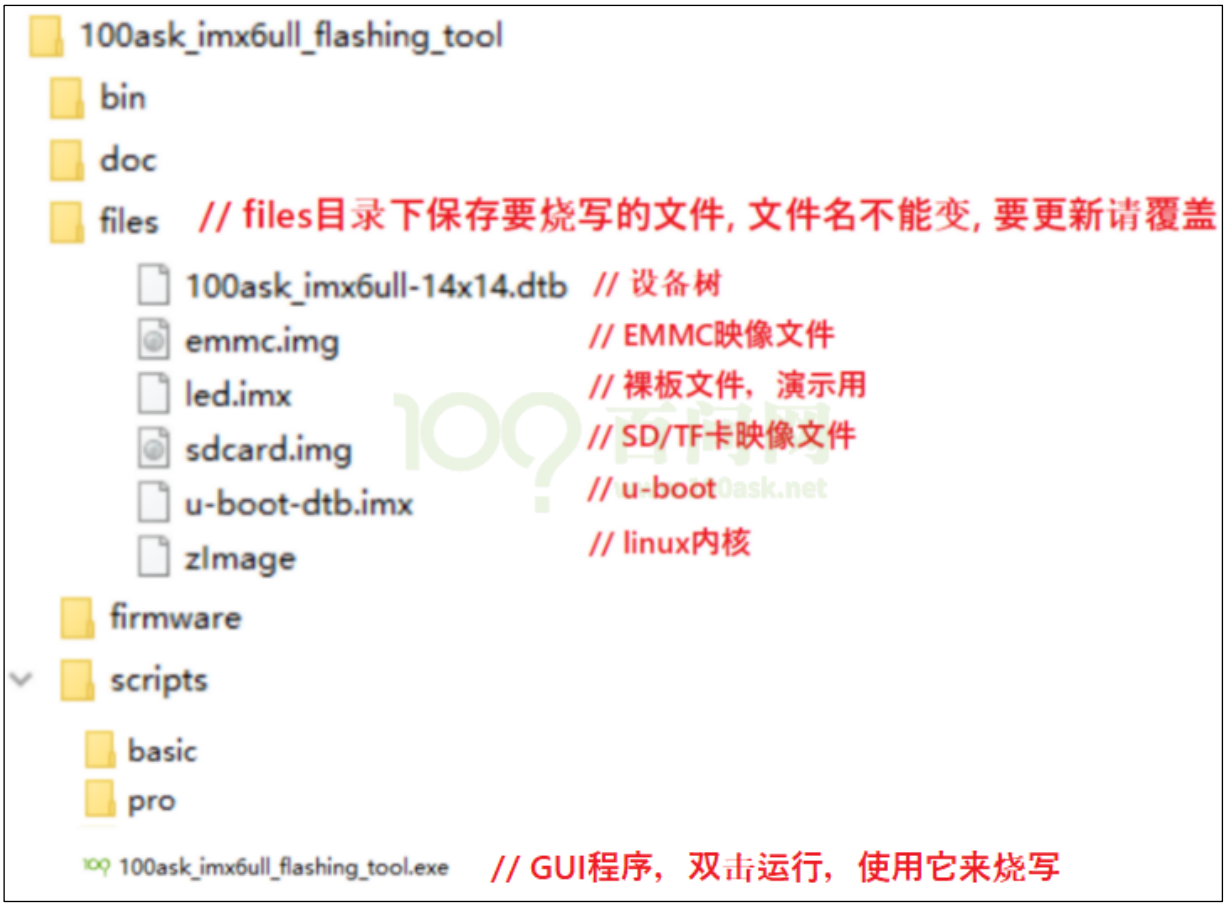


图 7.5 烧写工具目录

要留意的是 files 目录下的文件，各文件的作用在上图中列出来了，文件名不能改变，要更新某文件时需要覆盖旧文件。

这个烧写工具不断更新，上述截图中 files 目录下内容可能会增加，更多文件内容请看表 7-2

表 7-2 烧写工具其他文件说明

目录	文件	说明
file	u-boot-dtb.imx	IMX6ULL Pro 版的 u-boot，烧到 EMMC 或 SD/TF 卡
	u-boot-dtb_nand.imx	IMX6ULL mini nand 版的 u-boot，烧到 Nand
	u-boot-dtb_nandsd.imx	IMX6ULL mini nand 版的 u-boot，烧到 SD/TF 卡
	zImage	内核，适用于多个板子
	100ask_imx6ull-14x14.dtb	IMX6ULL Pro 版的设备树文件
	100ask_myir_imx6ull_mini.dtb	IMX6ULL mini nand 版的设备树文件
	emmc.img	IMX6ULL Pro 版的 EMMC 映像文件
	sdcard.img	IMX6ULL Pro 版的 SD/TF 卡映像文件
	rootfs.ubi	IMX6ULL mini nand 版的 Nand Flash 映像文件

7.3.2 连接 USB OTG 线

100ASK_IMX6ULL_PRO 接线、设置 USB 启动的方式如图 7.6：

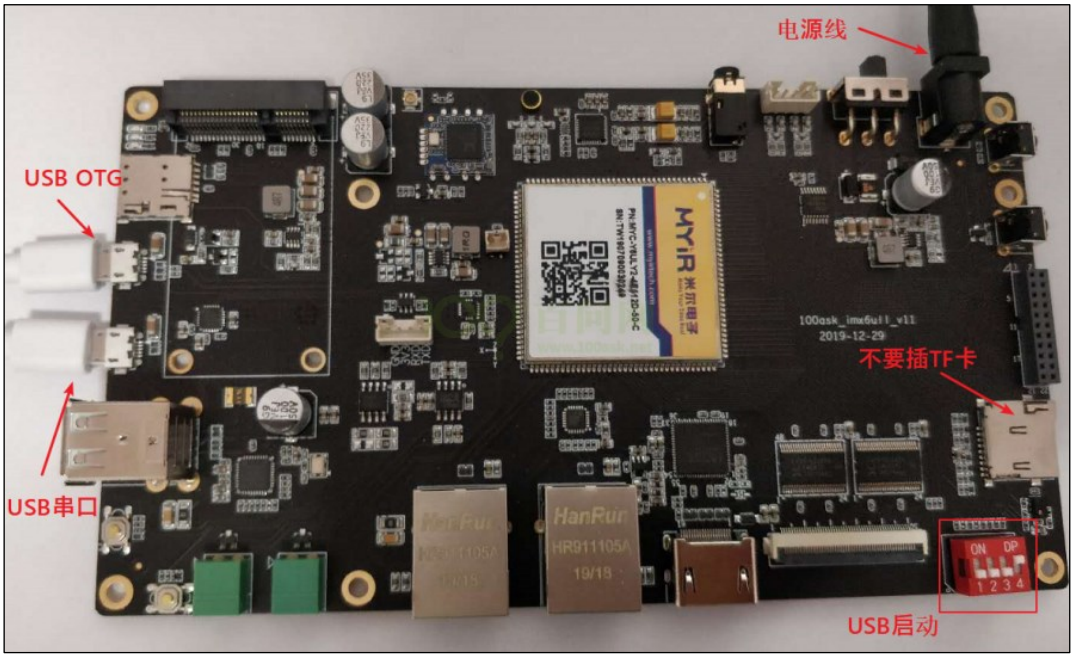


图 7.6 100ASK_IMX6ULL_PRO 接线

注意：USB 串口线可接可不接，接上只是为了观察烧写过程。
注意：设置为 USB 启动时，先不要接 TF 卡

7.3.3 安装 IMX6ULL 的 USB 驱动程序

通过 USB 下载或是烧写程序时，需要把开发板的 OTG 口用 USB 线连接到电脑。一般都会自动安装驱动，烧写软件的绿灯不亮时，则很有可能是驱动程序没有安装好，这时再来看本节文档。

1 VMWARE 可能会截取 USB 设备导致烧写失败

如果你运行了 VMWARE，它会截取 USB 设备，可能会有如图 7.7 提示：

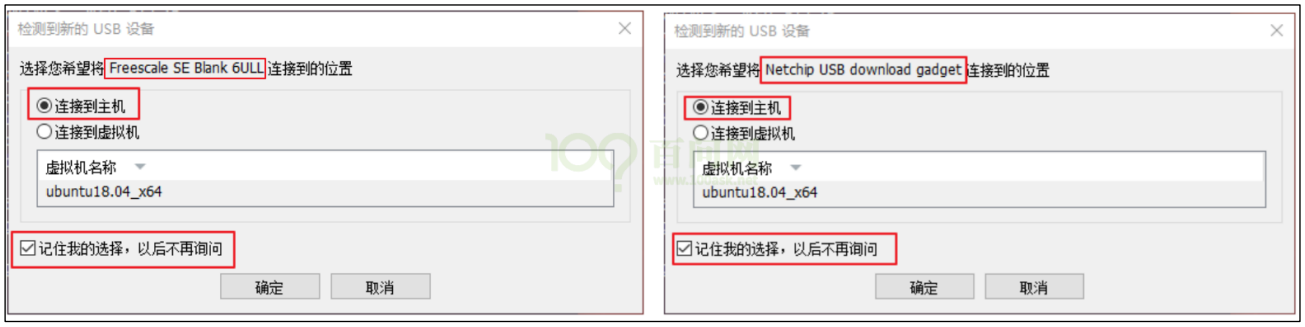


图 7.7 连接 USB 到主机

要选择“连接到主机”，勾选“记住我的选择，以后不再询问”。也许你不慎点错了“连接到虚拟机”，那也没关系，在 VMWARE 的菜单中把“Freescale SE Blank 6ULL”或“Netchip USB download gadget”断开连接，如下图所示：

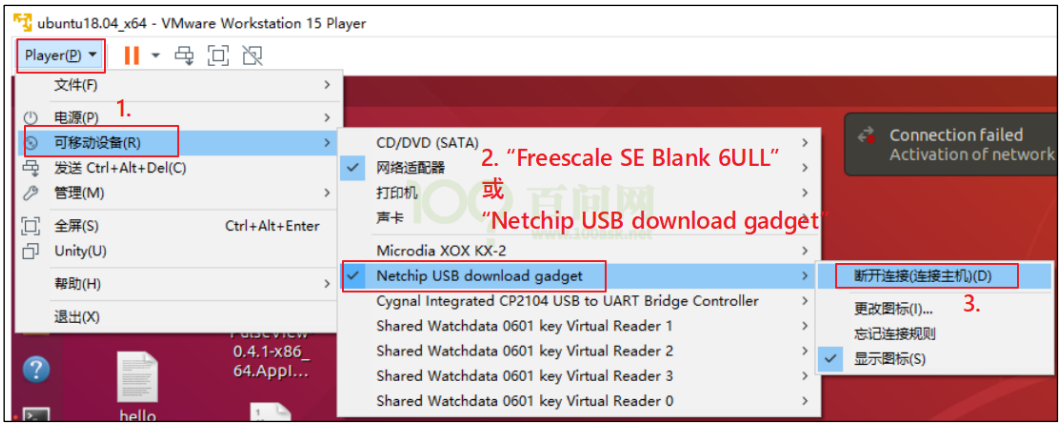


图 7.8 断开 USB 在虚拟机上的连接

2 USB 驱动没有自动安装

通过 USB 下载或是烧写程序时，需要把开发板的 OTG 口用 USB 线连接到电脑。IMX6ULL 开发板会通过这一个 OTG 口，模拟出 2 个 USB 设备(先后模拟，不是同时)："Freescale SE Blank 6ULL"、"USB download gadget"。一般会自动安装好驱动程序，如果驱动程序没安装好，你可以去下载"zadig"来安装驱动。

- **安装第 1 个驱动**：只要开发板设为 USB 启动(不要插 SD/TF 卡)并上电，电脑就会识别出"Freescale SE Blank 6ULL"设备，一般都会自动给它安装驱动程序。

如果没有自动安装好驱动程序("设备已连接"绿灯没亮)，要使用 zadig 安装 libusb-win32 驱动。

先去 https://zadig.akeo.ie/下载 zadig 并运行，然后参考图 7.9 安装驱动程序：

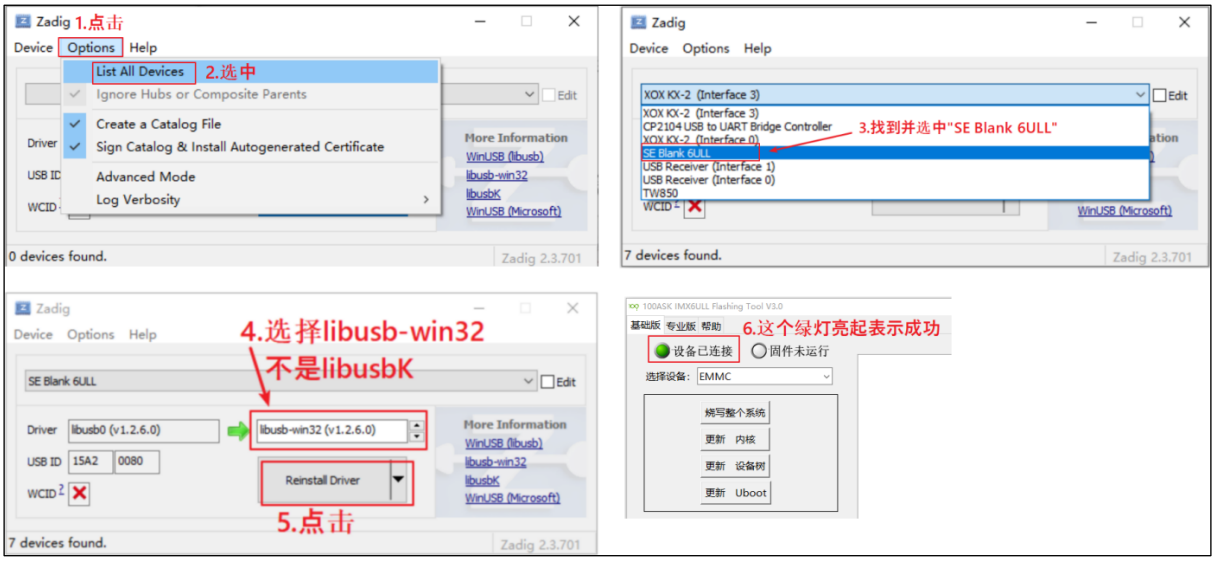


图 7.9 zadig 安装驱动

● **安装第 2 个驱动**：当烧写工具的“设备已连接”绿灯亮起，就可以在“专业版”点击“运行”按钮，这时电脑会识别出“USB download gadget”设备，一般都会自动给它安装驱动程序，如图 7.10：

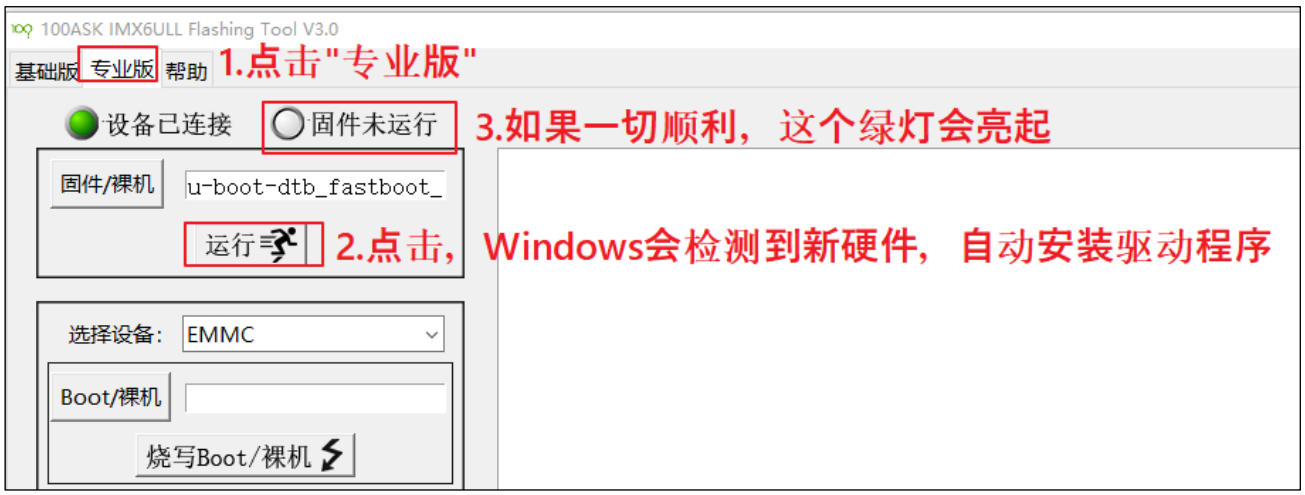


图 7.10 烧写工具烧写程序

如果没有自动安装好驱动程序（“固件已运行”绿灯没亮），先去 https://zadig.akeo.ie/ 下载 zadig 并运行，然后参考图 7.11 安装驱动程序：

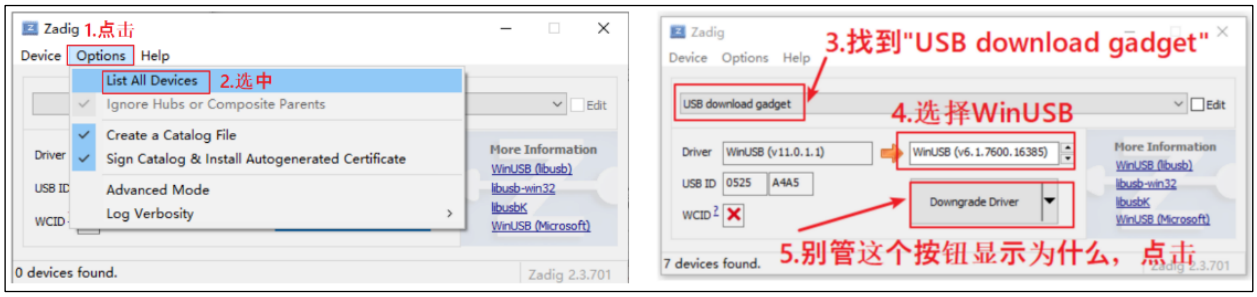


图 7.11 zadig 安装驱动

如果一切正常，烧写工具的 2 个绿灯都会亮，如图 7.12：

图 7.12 驱动正常烧写工具提示

这就表示所有驱动都安装好了，可以重启开发板，使用后面 2 节的方法来烧写。

7.3.4 基础版烧写方法

要更新某个文件，就把它复制到烧写工具的 files 目录，比如 zImage、100ask_imx6ull-14x14.dtb。

举例：要更新内核，先把新的 zImage 覆盖 files 目录下的 zImage，然后接线，上电，选择设备，点击“更新内核”即可。

板子复位或重新上电后，在 APP 里操作即可，一个图就可以列清楚所有步骤：

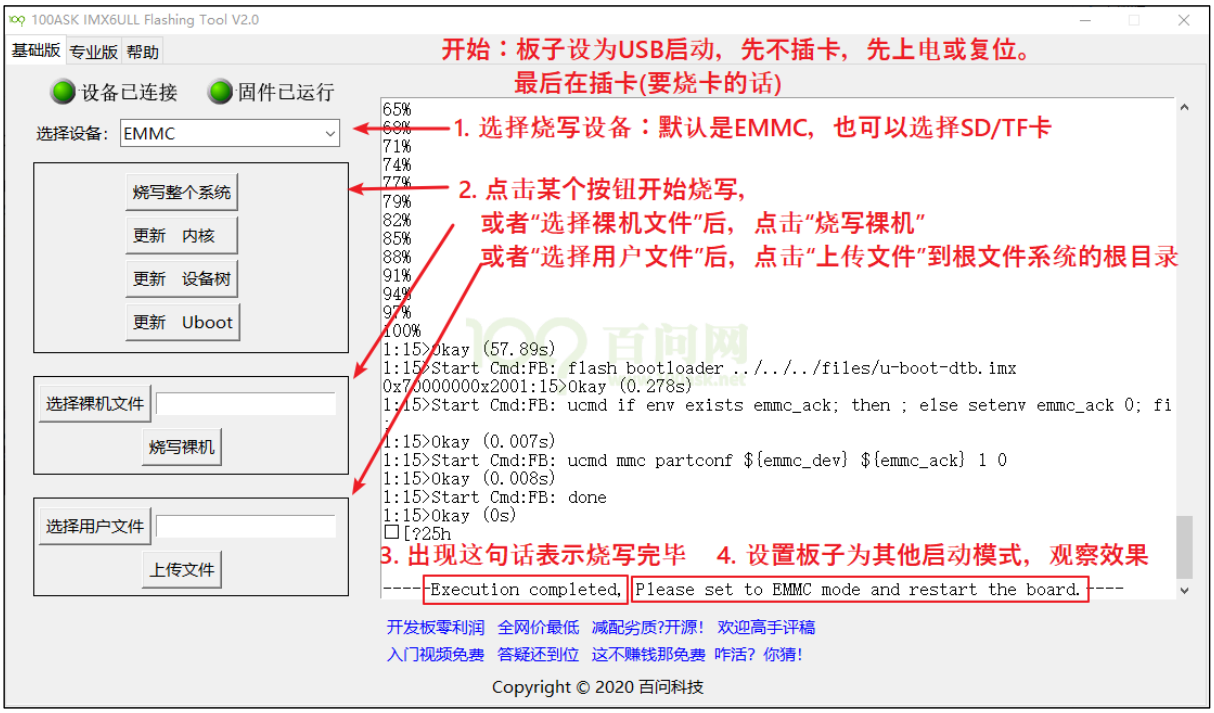


图 7.13 烧写工具烧基础版写程序示意

当开发板设为 USB 模式，上图的烧写工具中“设备已连接”的绿灯亮起，就可以通过按钮一键烧写了，各按钮作用如表 7-3：

表 7-3 烧写工具按钮作用说明

按钮	作用
烧写整个系统	● “选择设备”为 EMMC 时，把 emmc.img 烧到 EMMC 上； ● “选择设备”为 SD/TF 时，把 sdcard.img 烧到 SD/TF 卡上； ● “选择设备”为 NAND 时，把 rootfs.ubi 烧到 Nand Flash 上； 并且会烧写对应的 U-Boot，请看下面的“更新 Uboot”按钮说明。
更新内核	把 zImage 上传到根文件系统的/boot 目录(对于 Nand，是直接烧到内核分区)
更新设备树	把 100ask_imx6ull-14x14.dtb 上传到根文件系统的/boot 目录(对于 Nand，是直接烧到设备树分区)
更新 Uboot	对于 IMX6ULL Pro 版： ● “选择设备”为 EMMC 时，把 u-boot-dtb.imx 烧写到 EMMC ● “选择设备”为 SD/TF 时，把 u-boot-dtb.imx 烧写到 SD/TF 卡 对于 IMX6ULL mini nand 版： ● “选择设备”为 NAND 时，把 u-boot-dtb_nand.imx 烧写到 Nand Flash ● “选择设备”为 SD/TF 时，把 u-boot-dtb_nandsd.imx 烧写到 SD/TF 卡
烧写裸机	把所选裸机文件，烧写到 EMMC、SD/TF 卡或 Nand Flash
上传文件	把所选用户文件，上传到根文件系统的/目录 对于 imx6ull mini nand 版，无法上传文件(只支持 ext4 文件系统，而它不是)

7.3.5 专业版烧写方法

专业版的强大在于烧写文件时可以选择任意文件，上传文件时可以指定分区、分区格式、目标路径。用法也很简单，一图足以说明：

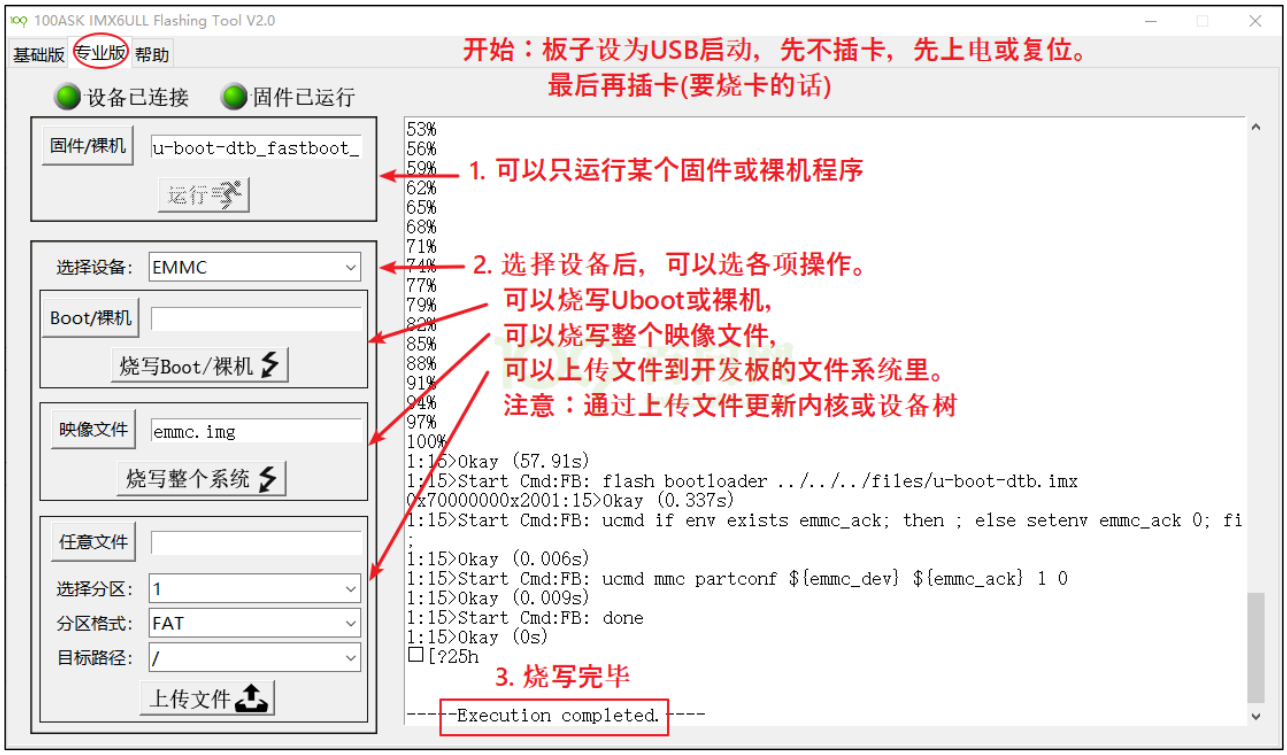


图 7.14 烧写工具专业版烧写示意

有些开发板厂家把内核 zImage、设备树放在第 1 个分区里，它通常是 FAT 分区。那么可以使用专业版来更新内核、更新设备树。比如：

图 7.15 更新内核、设备树

几乎所有的开发板的第 2 个分区都是 EXT4 格式，我们可以上传文件到它的任意目录下，比如：

图 7.16 更新内核、设备树示意

7.4 在 Ubuntu 下使用 USB 来烧写

Windows 下的 100ask_imx6ull_flashing_tool 实际上只是一个 GUI 前

端，它通过调用命令行工具 uuu 来烧写系统。

我们没有给 Ubuntu 编写 GUI，只能使用命令行来操作；这个方法同样适用于 Windows。

解压“100ask_imx6ull 烧写工具.tar.bz2”后，可得如下文件：

```
100ask_imx6ull 烧写工具
├─── bin                  // 烧写用的程序
│    ├─── uuu                        // Linux 下的烧写工具
│    └─── uuu.exe                    // Windows 下的烧写工具
├─── doc
│    └─── UUU.pdf
├─── files                // 在 files 目录下保存要烧写的文件，文件名不能改，要更新请覆盖旧文件
│    ├─── 100ask_imx6ull-14x14.dtb   // 设备树
│    ├─── emmc.img                   // emmc 映象文件
│    ├─── led.imx                    // 裸板文件，示例
│    ├─── sdcard.img                 // sd/tf 卡映象文件
│    ├─── u-boot-dtb.imx             // u-boot
│    └─── zImage                     // 内核
├─── firmware             // 固件，uuu 要借助这个目录下的 u-boot 来烧写，这是我们特制的
│    └─── u-boot-dtb_fastboot_100ask.imx
└─── scripts/basic        // 基础版脚本
     ├─── emmc             // 烧写 emmc 的脚本
     │    ├─── write_all.clst          // 烧写 EMMC 的整个映象文件：files/emmc.img
     │    ├─── write_boot.clst         // 烧写 u-boot：files/u-boot-dtb.imx
     │    ├─── write_dtb.clst          // 烧写设备树：files/100ask_imx6ull-14x14.dtb
     │    └─── write_kernel.clst       // 烧写内核：files/zImage
     ├─── nand             // 烧写 nand 的脚本，我们还没实现，我们的板子没有 nand
     └─── sd               // 烧写 sd/tf 卡的脚本
          ├─── write_all.clst          // 烧写 sd/tf 卡的整个映象文件：files/sdcard.img
          ├─── write_boot.clst         // 烧写 u-boot：files/u-boot-dtb.imx
          ├─── write_dtb.clst          // 烧写设备树：files/100ask_imx6ull-14x14.dtb
          └─── write_kernel.clst       // 烧写内核：files/zImage
```

注意事项：

① 用 USB 线连接电脑和开发板的 USB OTG 接口，USB 串口可接可不接，建议接上观察效果。

② 要烧写 SD/TF 卡，设置好 USB 启动方式后，必须先上电再插卡，绝对不能先插卡再上电。

③ 上电一次，只能点一次按钮烧写；要再次烧写，需要重新上电。

④ 在 Ubuntu 下执行命令时前面要加 **sudo**。

使用 uuu 烧写的步骤很简单，简单地说就是：

第6步 设置 USB 启动模式，

第7步 接线，

第8步 复位(或重新上电)，

第9步 插卡(要用卡的话)，

第10步 执行 uuu 命令。

如果要再次执行 uuu 命令来烧写，必须重来一次：

第1步 拔卡(有的话)

第2步 复位(或重新上电)

第3步 插卡(要用卡的话)

第4步 执行 uuu 命令。

uuu 命令用法：在 100ask_imx6ull 烧写工具目录下，执行命令：

```
# Ubuntu 下才需要加 sudo，Windows 下不用加
sudo  ./bin/uuu    scripts 目录下的某个脚本
```

比如要烧写 EMMC 的整个系统，即把 files/emmc.img 烧写到 EMMC 上，执行如下命令：

```
sudo  ./bin/uuu    scripts/basic/emmc/wirte_all.clst
```

对于裸机程序，因为裸机程序的名字各有不同，我们没有提供固定的脚本。命令解析如下：

```
# 借助 u-boot 把 led.imx 烧到 EMMC
sudo  ./bin/uuu  -b  emmc  firmware/u-boot-dtb_fastboot_100ask.imx  files/led.imx
```

或

```
# 借助 u-boot 把 led.imx 烧到 sd/tf 卡
sudo  ./bin/uuu  -b   sd    firmware/u-boot-dtb_fastboot_100ask.imx   files/led.imx
```

7.5 更新根文件系统

并不需要单独去更新根文件系统，使用我们提供的 Buildroot 系统可以制作出 emmc.img、sdcard.img，把它们烧写到 EMMC 或 SD/TF 卡上去就可以了。可以先把 emmc.img 或 sdcard.img 复制到烧工具的 files 目录，然后用烧写工具的“基础版”来更新系统。

7.6 使用板子的 Linux 系统来烧写

Linux 系统功能强大，IMX6ULL 启动 Linux 后，当然也可以烧写 EMMC、SD/TF 卡，当然就可以更新 u-boot、内核、设备树等等。这种方法在开发驱动程序时经常使用。

7.6.1 烧写 u-boot 到 EMMC 或 SD/TF 卡

首先将编译好的 uboot 镜像文件拷贝至 Ubuntu nfs 目录，启动开发板系统，在开发板上挂载 Ubuntu nfs 目录。

如果你使用的是 VMware 桥接方式，假设 Ubuntu IP 为 192.168.5.11，在开发板上执行以下命令：

```
[root@100ask:~]# mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
```

接下来在 Ubuntu 上先将编译好的 u-boot-dtb.imx 复制到 /home/book/nfs_rootfs 目录，如图 7.17 所示：

```
book@100ask:~/100ask_imx6ull-sdk/Uboot-2017.03$ cp ./u-boot-dtb.imx /home/book/nfs_rootfs/
book@100ask:~/100ask_imx6ull-sdk/Uboot-2017.03$
```

图 7.17 复制 imx 到 nfs_rootfs

最后在开发板上把 NFS 目录中的 **u-boot-dtb.imx** 复制到开发板家目录下：

```
root@100ask:~]# cp /mnt/u-boot-dtb.imx .
root@100ask:~]# sync
root@100ask:~]#
```

图 7.18 复制 imx 到开发板

1 拷贝成功后，如果要烧写 EMMC，执行如下命令：

```
[root@100ask:~]# echo 0 > /sys/block/mmcblk1boot0/force_ro     //取消此分区的只读保护
[root@100ask:~]# dd if=u-boot-dtb.imx of=/dev/mmcblk1boot0 bs=512 seek=2 //实际烧写命令
[root@100ask:~]# echo 1 > /sys/block/mmcblk1boot0/force_ro     //打开此分区的只读保护
```

```
root@100ask:~]# echo 0 > /sys/block/mmcblk1boot0/force_ro
root@100ask:~]# dd if=u-boot-dtb.imx of=/dev/mmcblk1boot0 bs=512 seek=2
1022+0 records in
1022+0 records out
523264 bytes (523 kB, 511 KiB) copied, 0.252293 s, 2.1 MB/s
root@100ask:~]# echo 1 > /sys/block/mmcblk1boot0/force_ro
root@100ask:~]#
```

图 7.19 烧写 emmc

烧写完成后，确保开发板设置为 EMMC 启动，可以观察到刚烧写的 U-boot 的启动信息。

2 拷贝成功后，当前目录有 u-boot-dtb.imx，如果要烧写 SD/TF 卡，执行如下命令：

```
[root@100ask:~]# dd if=u-boot-dtb.imx  of=/dev/mmcblk0 bs=512 seek=2
```

烧写完成后，确保开发板设置为 SD/TF 卡启动，可以观察到刚烧写的 U-boot 的启动信息。

7.6.2 开发板挂载 NFS 目录后更新内核或设备树

开发板使用的内核名为 zImage，设备树名为 100ask_imx6ull-14x14.dtb。它们保存在开发板的/boot 目录中。

更新它们很简单：只要替换/boot 目录下的文件即可。

假设 Ubuntu IP 为 192.168.5.11，在开发板上执行以下命令：

```
[root@100ask:~]# mount -t nfs -o nolock,vers=3 192.168.5.11:/home/book/nfs_rootfs /mnt
```

挂载成功后将已经拷贝到 Ubuntu nfs 目录中的内核、设备树文件拷贝到开发板/boot 目录下，替换掉原来的文件：

```
[root@100ask:~]# cp /mnt/zImage  /boot
[root@100ask:~]# cp /mnt/100ask_imx6ull-14x14.dtb /boot
[root@100ask:~]# sync
```

等待同步完成后重启开发板即可。

注意：更新完成新编译的 zImage 内核镜像和设备树文件后，还需要更新内核模块驱动文件，否则可能会出现某些外围设备无法使用的问题，内核模块编译方法和安装方法请参考：

- 《>>>>>>>>第三篇第 5 章 5.3 编译安装内核模块》
- 《>>>>>>>>第三篇第 5 章 5.4 安装内核和模块到开发板上》

7.7 在 Windows 下烧写 SD/TF 卡

7.7.1 格式化 SD/TF 卡

1 安装 sd 卡格式化工具

在网盘资料目录：\02_开发工具\【Windows】SD 卡格式化工具下双击运行图 7.20 所示的安装程序：

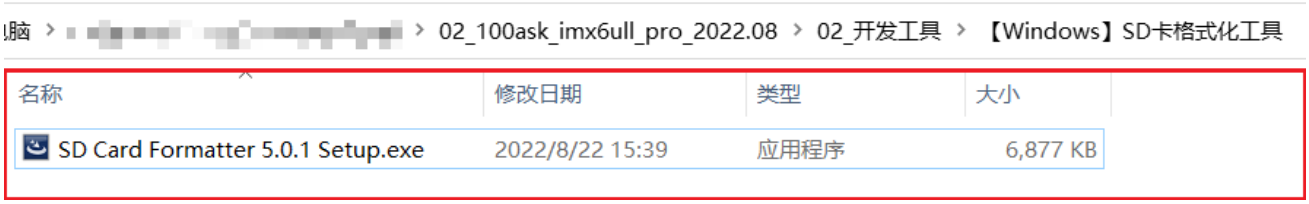


图 7.20 SD 卡格式化工具安装程序

安装步骤就省略了，都是傻瓜式的安装流程，下面开始格式化 SD/TF 卡。

2 格式化 SD 卡

将 SD 卡插到读卡器中，再将读卡器插到电脑的任意 USB 口，建议不要使用 USB 扩展坞或类似的 USB 扩展工具，容易检测不到 SD 卡。

电脑成功检测到 SD 卡后，类似如图所示：

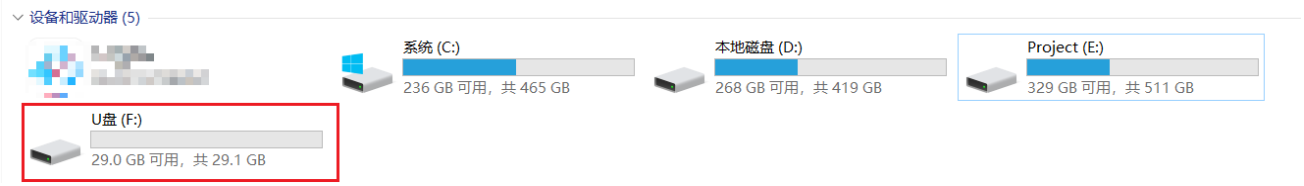


图 7.21 电脑读取到 SD 卡

不同的电脑对于 U 盘的读取显示可能不同，但是自己还是能很容易的区分它是不是插入的 U 盘或 SD 卡的。检测到之后就打开上一步安装好的 SD 卡格式化工具“SD Card Formatter”：

图 7.22 管理员身份运行程序

运行后可能会提示升级：

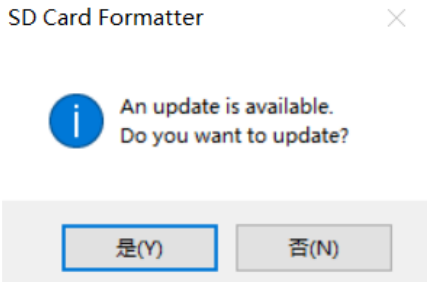


图 7.23 提示升级

选择否即可，然后就弹出了程序主体界面：

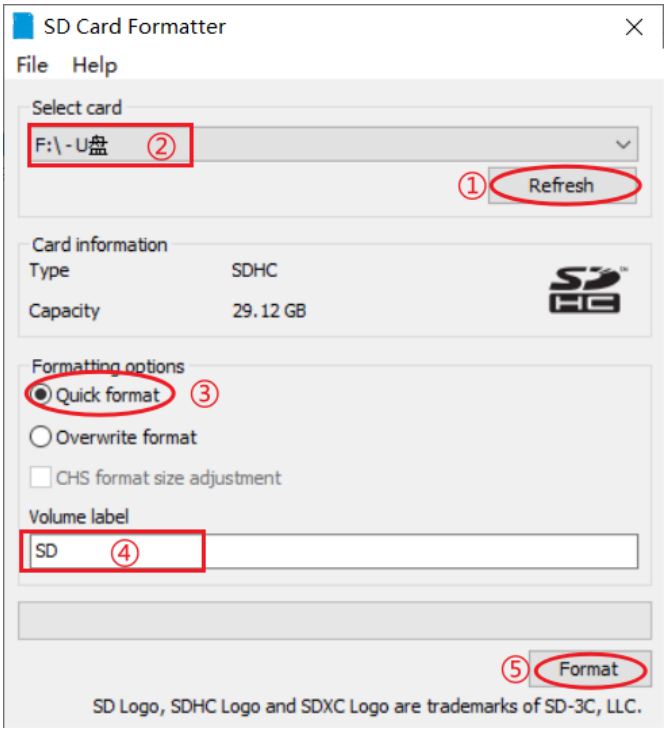


图 7.24 SD Card Formatter 界面

根据图中步骤操即可：

① 刷新 SD 卡/U 盘；
② 找到要格式化的目标盘；
③ 选择“Quick format”；
④ 自定义格式化后的名称，不能有中文！
⑤ 点击“Format”开始格式化；

随后会弹出警告，警示我们会将此盘的内容全部清除，是否继续等等，如图所示：

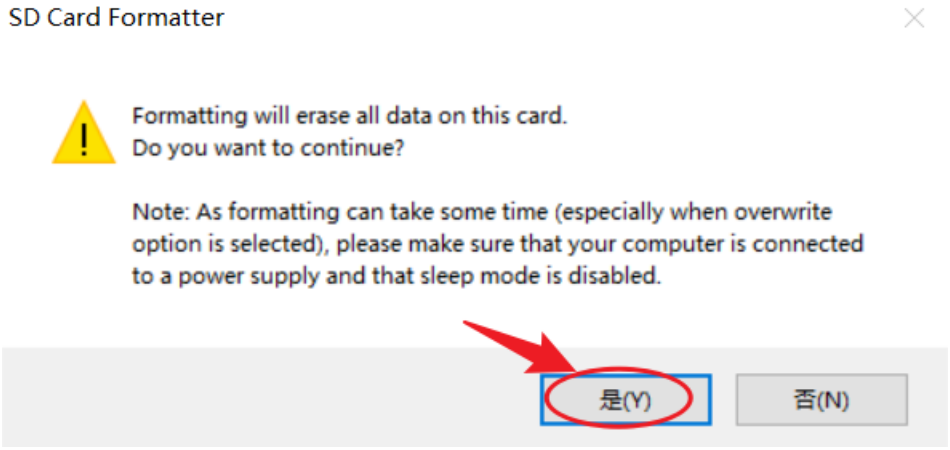


图 7.25 警示提示

点击“是”后开始格式化：

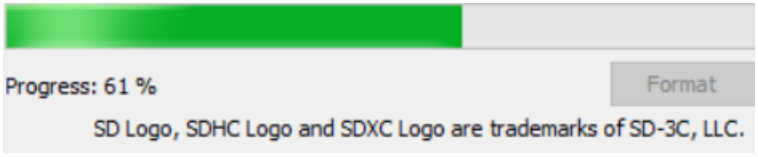


图 7.26 正在格式化

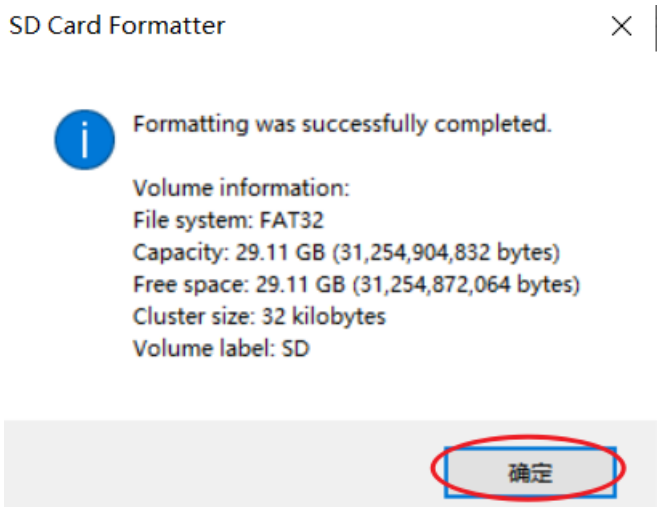


图 7.27 格式化成功

提示格式化成功后点击“确定”关掉提示，然后再关闭格式化工具，至此 SD 卡格式化完成，接下来开始烧写镜像到 SD 卡将其设置为开发板的启动盘。

7.7.2 烧写镜像到 SD/TF 卡

1 安装 SD 卡 IMG 系统镜像烧写工具

在网盘资料：\02_开发工具\【Windows】SD 卡 IMG 系统镜像烧写工具目录下有如图 7.28 所示的安装程序：

图 7.28 SD 镜像烧写工具安装程序

双击运行此安装程序即可，安装步骤很简单，同样也省略跳过，安装完成后打开该软件：

图 7.29 管理员身份运行 win32diskimager

2 烧录镜像到 SD 卡

在上一步已经打开了 win32diskimager 软件，图所示是其主界面：

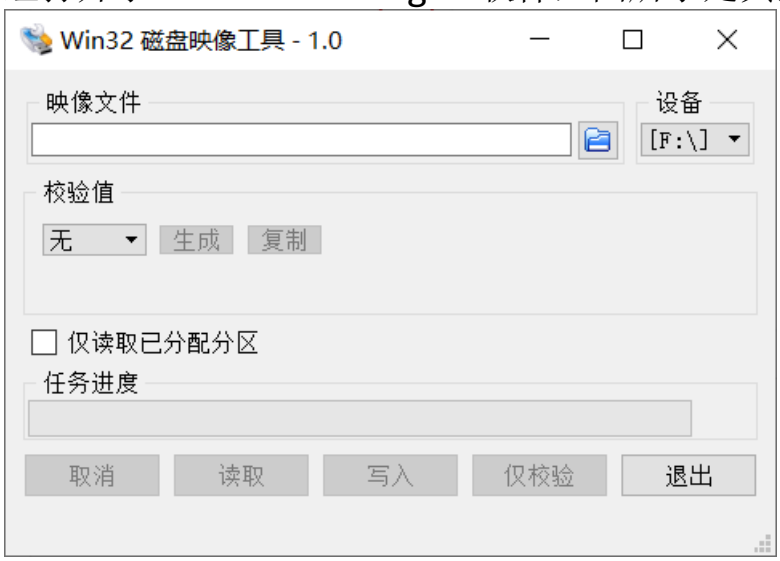


图 7.30 win32diskimager 主界面

我们需要先选择要烧写的 img 映像，然后选择要烧录的目标 SD 卡：

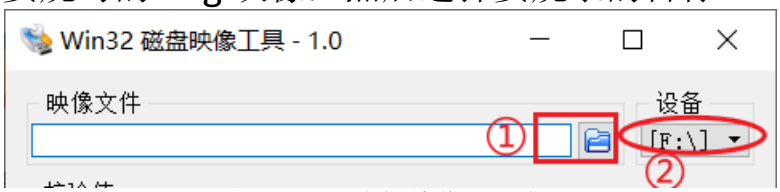


图 7.31 选择镜像和设备

百问网提供的默认 SD 镜像在网盘资料目录：

\03_开发板系统固件\Sdcard

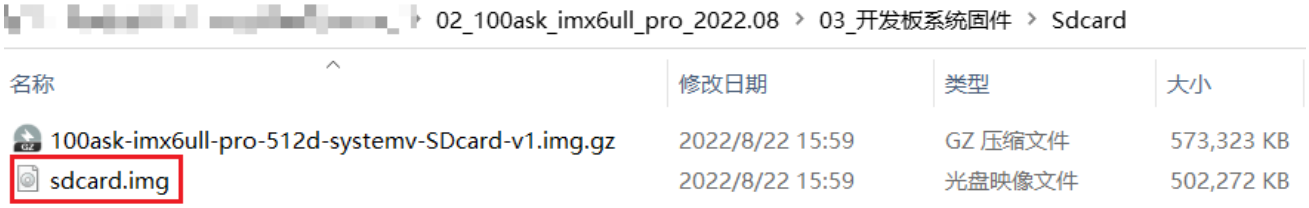


图 7.32 百问网官方提供 SD 镜像

这里就以此镜像为例，实际开发移植中就选择自己配置编译好的 SD 镜像：

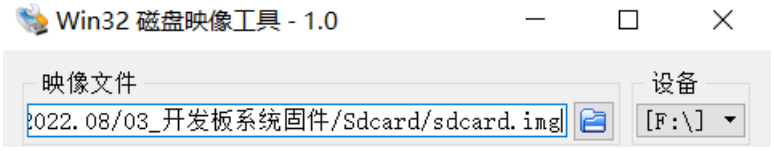


图 7.33 选择好镜像

选择好镜像和要烧写的目标 SD 卡后，就选择“写入”开始烧写：

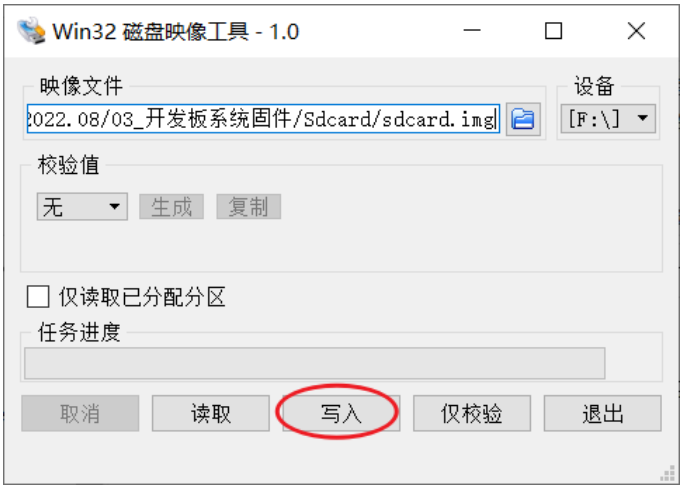


图 7.34 选择“写入”

随后会弹出如图所示的提示，选择“Yes”即可：

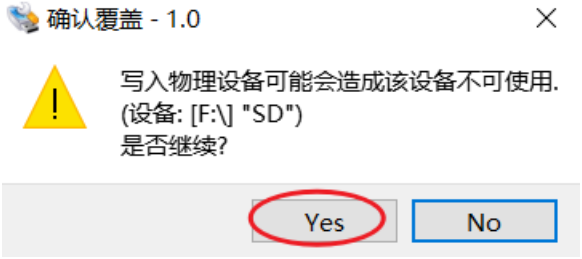


图 7.35

然后就等待烧写完成：

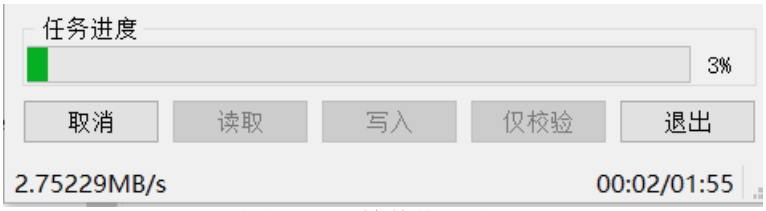


图 7.36 等待烧写完成

直到提示“写入成功”点击“OK”即完成烧写镜像到 SD 卡：

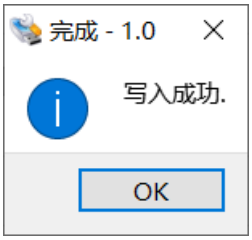


图 7.37 写入成功

烧写完成后会弹出下图的提示：

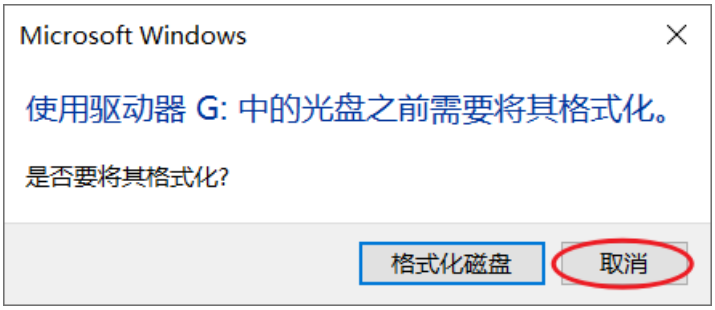


图 7.38 烧写镜像到 SD 卡后的提示

一律选择“取消”，千万不要“格式化磁盘”。

至此，镜像就成功的烧写到 SD 卡内了，接下来就是将 SD 卡插入到开发板上的 SD/TF 卡槽内，开发板的启动方式选择 SD 启动，再上电即可。

7.7.3 SD 启动验证烧写结果

启动方式请看《>>>>>>>>第三篇第 3 章 3.3 启动方式选择》，连接串口且使用串口助手打开该串口，上电查看打印信息：

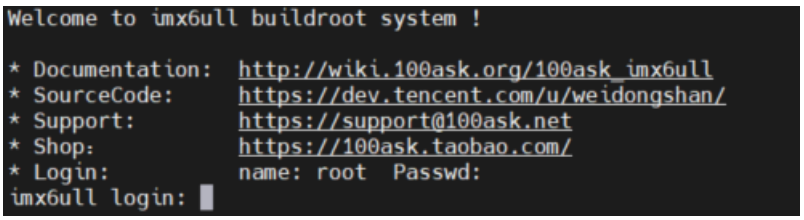


图 7.39 验证成功

7.8 在 Ubuntu 下烧写 SD/TF 卡

在 Ubuntu 下可以更精细地操作 SD/TF 卡：可以把 sdcoard.img 整个烧写到卡上，也可以单独烧写 u-boot 到卡上，甚至挂接卡上的文件系统后单独更新

里面的文件。

7.8.1 识别 SD/TF 卡

首先 TF 卡插入读卡器，再把读卡器插入电脑。VMWare 有时候会自动弹出对话框，选择“连接到虚拟机”即可；如果没有对话框，可以通过菜单“可移动设备”找到 SD 卡，点击“连接”。

然后使用 dmesg 命令获取设备挂载的设备节点，根据 log 信息获得该磁盘设备有两个分区，分别是 /dev/sdb1、/dev/sdb2。/dev/sdb 表示整个卡。如图 7.40 所示：

```
book@100ask: ~/100ask_imx6ull-sdk/Uboot-2017.03$ dmesg
```

```
[20862.255215] usb 1-1: new high-speed USB device number 3 using ehci-pci
[20862.636666] usb 1-1: New USB device found, idVendor=05e3, idProduct=0723, bcdDevice=94.51
[20862.636668] usb 1-1: New USB device strings: Mfr=3, Product=4, SerialNumber=2
[20862.636669] usb 1-1: Product: USB Storage
[20862.636690] usb 1-1: Manufacturer: Generic
[20862.636691] usb 1-1: SerialNumber: 000000009451
[20862.638968] usb-storage 1-1:1.0: USB Mass Storage device detected
[20862.639315] usb-storage 1-1:1.0: Quirks match for vid 05e3 pid 0723: 8000
[20862.639397] scsi host33: usb-storage 1-1:1.0
[20863.652551] scsi 33:0:0:0: Direct-Access     Generic  STORAGE DEVICE   9451 PQ: 0 ANSI: 0
[20863.653067] sd 33:0:0:0: Attached scsi generic sg2 type 0
[20863.786938] sd 33:0:0:0: [sdb] 15499264 512-byte logical blocks: (7.94 GB/7.39 GiB)
[20863.795903] sd 33:0:0:0: [sdb] Write Protect is off
[20863.795904] sd 33:0:0:0: [sdb] Mode Sense: 03 00 00 00
[20863.802863] sd 33:0:0:0: [sdb] No Caching mode page found
[20863.802865] sd 33:0:0:0: [sdb] Assuming drive cache: write through
[20863.838022]  sdb: sdb1 sdb2
[20863.862714] sd 33:0:0:0: [sdb] Attached SCSI removable disk
[20864.627165] FAT-fs (sdb2): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
[20864.760128] EXT4-fs (sdb1): recovery complete
[20864.765887] EXT4-fs (sdb1): mounted filesystem with ordered data mode. Opts: (null)
```

图 7.40 获取 sdb

7.8.2 烧写整个 SD/TF 卡

使用 dd 命令烧写 sdcard.img 镜像文件到 /dev/sdb 设备：

```
sudo  dd if=sdcard.img  of=/dev/sdb
```

烧写成功后，把 SD/TF 卡插到开发板上，设置在 SD/TF 卡启动，上电即可看到启动信息。

7.8.3 烧写 u-boot 到 SD/TF 卡

可以只更新卡上的 u-boot，使用 dd 命令烧写 imx 镜像文件到 /dev/sdb 设备：

```
book@100ask: ~/100ask_imx6ull-sdk/uboot2017.03$ sudo  dd if=u-boot-dtb.imx of=/dev/sdb bs=1k seek=1 conv=fsync
```

注意：是烧写到 1K 偏移处。烧写成功后，把 SD/TF 卡插到开发板上，设置在 SD/TF 卡启动，上电即可看到 u-boot 信息。

7.8.4 更新 SD/TF 卡中的内核和设备树

对于曾经烧写过的 SD/TF 卡，上面已经有分区。使用 vmware 连接 TF 卡设备后，Ubuntu 系统系统会自动挂载 tf 卡内的分区文件系统，如图 7.41 所示，

本示例的 TF 卡第二个分区被挂载到了**/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32 (每个人的挂载目录可能会不一样)**目录下，此时可以进入此目录进行操作。

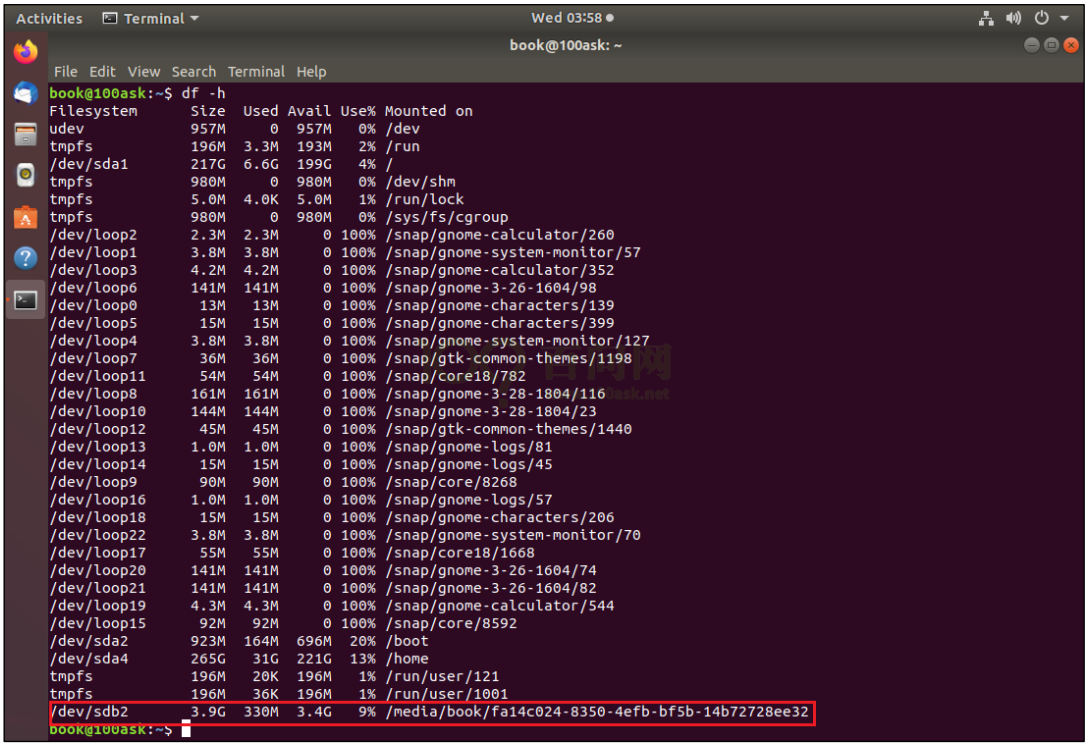


图 7.41 SD 卡已有分区

我们进入这个已经挂载的 TF 卡第二个分区目录，通过 ls 命令查看一下目录下都有哪些文件：

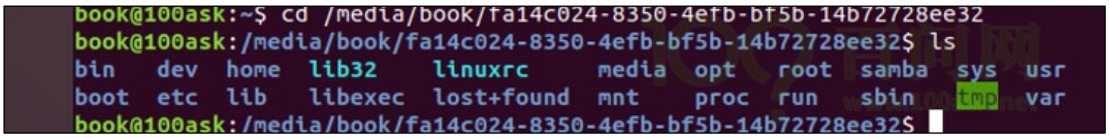


图 7.42 查看分区文件

之后我们需要拷贝之前编译好的 zImage 文件到 TF 卡的 boot 目录下，替换已有的，执行步骤如下：

第1步 进入 **boot** 目录

第2步 使用 **sudo** 命令拷贝之前编译好的目录到 TF 卡文件系统的 boot 目录下。此时会提示你输入 Ubuntu 用户密码，输入后确认，此时开始拷贝文件。

第3步 拷贝完成后使用 **sync** 命令同步数据缓存，

第4步 最后切换到另一个目录，使用 **sudo umount** 命令卸载 TF 最后就可以取出 TF 卡，插入开发板等待启动系统。

```
book@100ask:/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32$ cd boot/
book@100ask:/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32/boot$ ls
100ask_imx6ull-14x14.dtb  zImage
book@100ask:/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32/boot$ sudo  cp ~/zImage  .
[sudo] password for book:
book@100ask:/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32/boot$ sync
book@100ask:/media/book/fa14c024-8350-4efb-bf5b-14b72728ee32/boot$ cd
book@100ask:~$ sudo umount /media/book/fa14c024-8350-4efb-bf5b-14b72728ee32
book@100ask:~$
```

图 7.43 卸载 SD 卡

7.9 老方法：使用 mfgtool 烧写整个系统到 EMMC 上

解压缩网盘配套资料目录下

02_100ask_imx6ull_pro_2022.08\03_开发板系统固件\Emmc\100ask_imx6ull-mfgtools.zip

到任意文件夹。然后就可以开始烧写了，步骤如下：

第1步 设置开发板为 USB 启动，接线，上电，如图 7.44 所示，

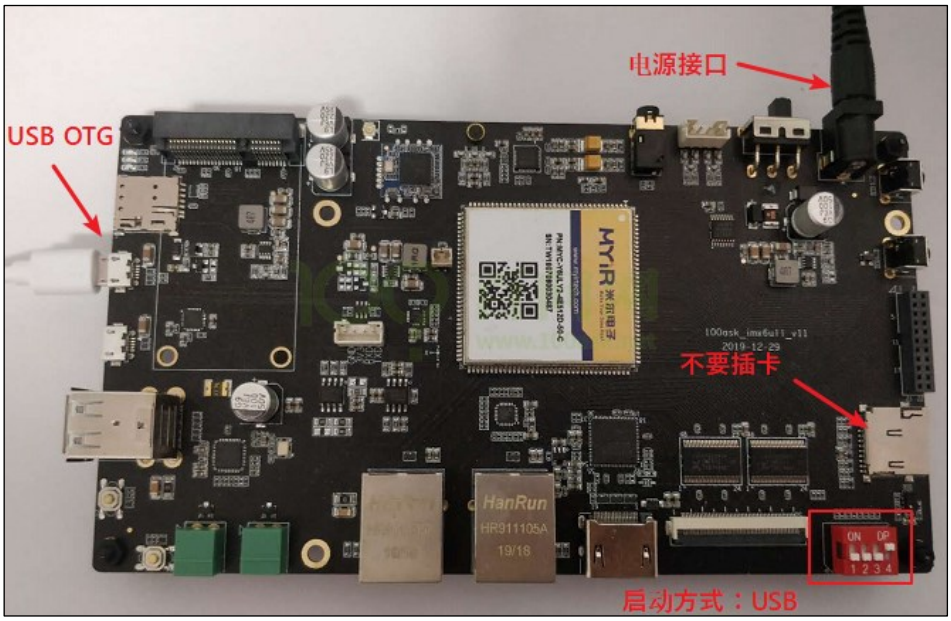


图 7.44 接线

注意：设置为 USB 启动时，不要接 TF 卡；图 7.44 中 USB OTG 线是连接到电脑的。

注意：USB OTG 线不要接入 USB 扩展坞或扩展设备的接口，要直接接入电脑的 USB 接口！

第2步 双击运行 mfgtools。进入解压后的 100ask_imx6ull-mfgtools 文件夹下，双击 **buildroot-image-100ask_100ask-ddr512m-emmc4g.vbs** 打开烧写程序，烧写过程中保持 usb 连接。

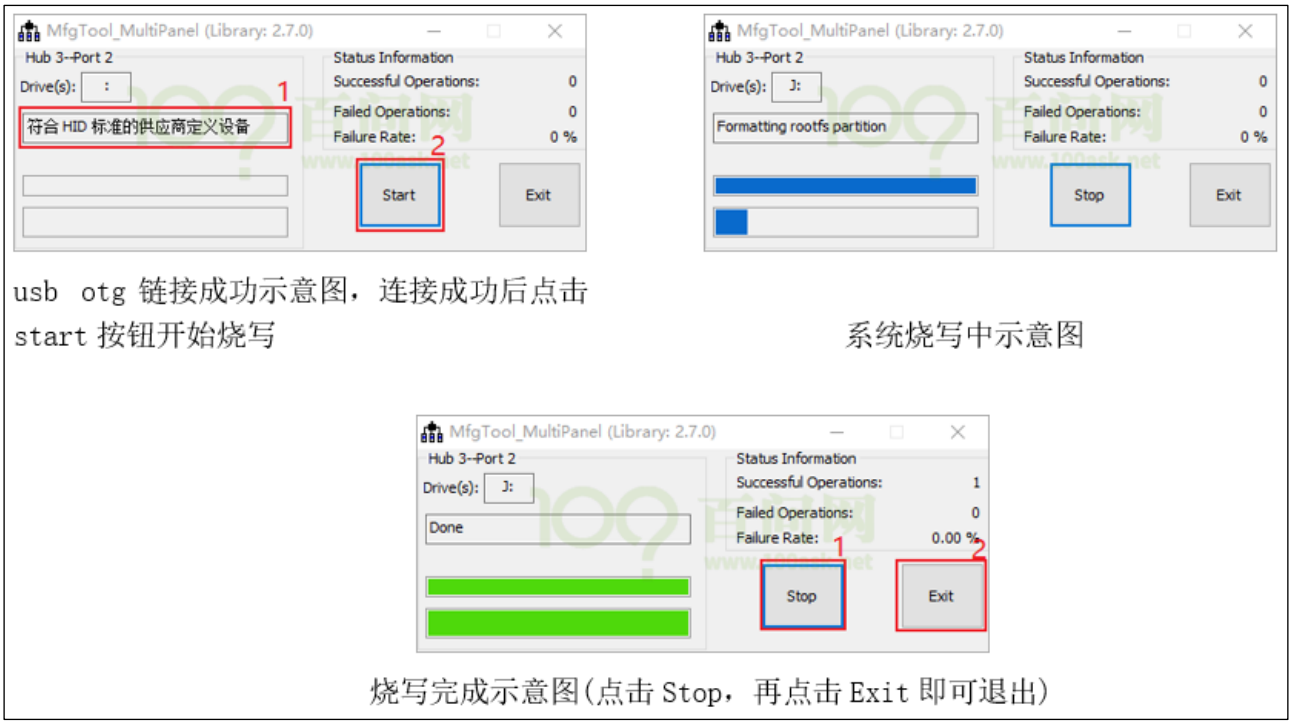


图 7.45 mfgtools 烧写

第3步 设置开发板启动方式为 Emmc，重新上电即可观察到效果(启动方式设置在开发板背面印有丝印说明)。

>>>>>>>第四篇 嵌入式 Linux 应用开发基础知识<<<<<<<

操作视频链接：https://www.bilibili.com/video/BV1kk4y117Tu

对于应用开发基础，有录播、直播两部分教程。录播视频的配套文档，就是本编文档，写得比较详细。直播视频对应的文档，并没有整理成 word，而是放在 GIT 仓库里。录播视频、直播视频配套的源码，都放在 GIT 仓库里。

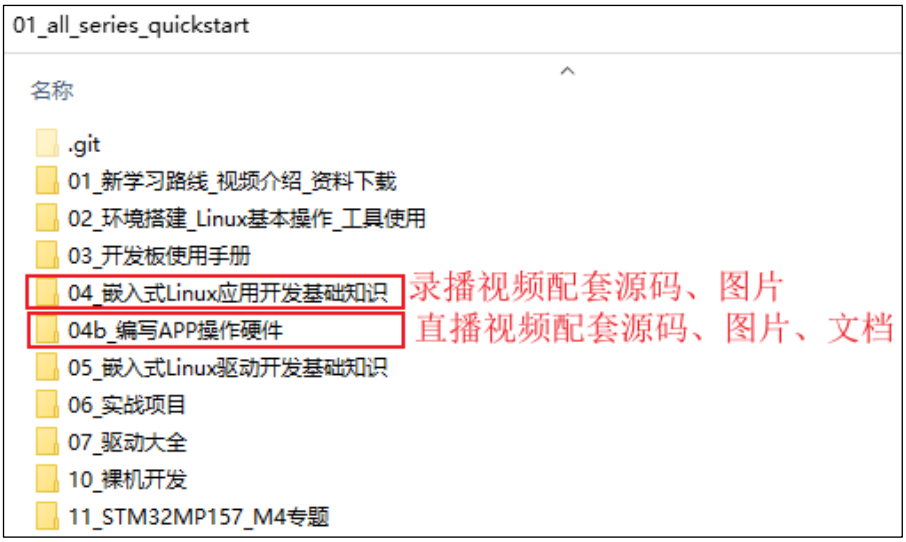


图 7.46 资料 git 仓库目录

第1章 HelloWorld 背后没那么简单

1.1 交叉编译 hello.c

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\source\01_hello
```

hello.c 的源码如下：

```
01 #include <stdio.h>
02
03 /* 执行命令: ./hello weidongshan
04  * argc = 2
05  * argv[0] = ./hello
06  * argv[1] = weidongshan
07  */
08
09 int main(int argc, char **argv)
10 {
11     if (argc >= 2)
12         printf("Hello, %s!\n", argv[1]);
13     else
14         printf("Hello, world!\n");
15     return 0;
16 }
```

在 Ubuntu 中可以执行以下命令编译、执行：

```
gcc -o hello hello.c
./hello
```

```
Hello, world!
./hello weidongshan
Hello, weidongshan!
```

上述命令编译得到的可执行程序 hello 可以在 Ubuntu 中运行，但是如果把它放到 ARM 板子上去，它是无法执行的。因为它是使用 gcc 编译的，是给 PC 机编译的，里面的机器指令是 x86 的。

我们要想给 ARM 板编译出 hello 程序，需要使用交叉编译工具链，比如：

```
arm-buildroot-linux-gnueabihf-gcc -o hello hello.c
```

这样编译出来的 hello 程序才可以在 ARM 板子上运行。

1.2 请回答这几个问题

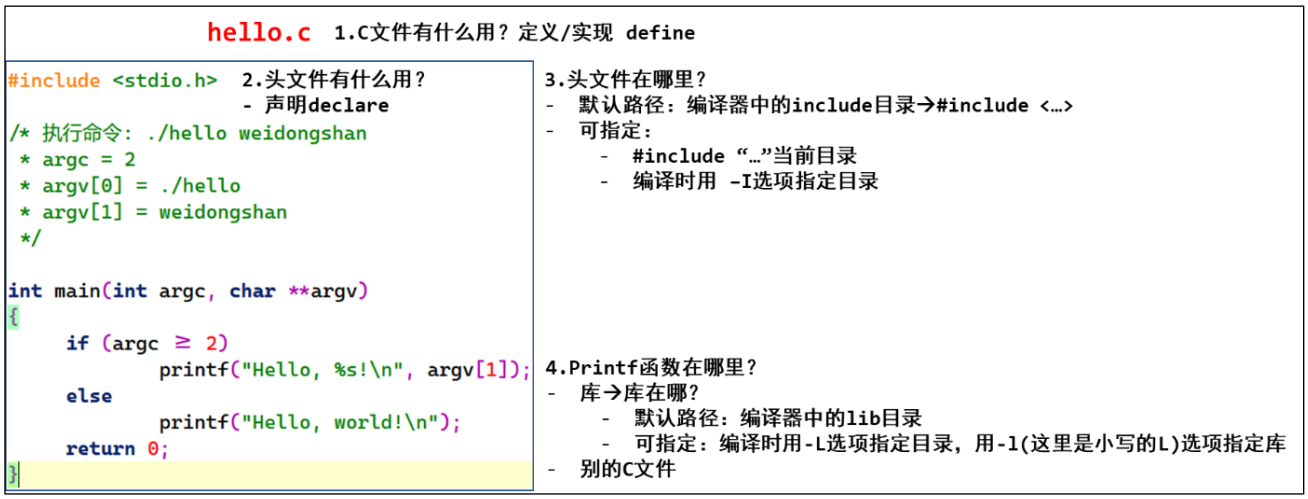


图 1.1 hello.c 中的几个问题

① 怎么确定交叉编译器中头文件的默认路径？

进入交叉编译器的目录里，执行：find -name "stdio.h"，它位于一个"include"目录下的根目录里。这个"include"目录，就是要找的路径。

② 怎么自己指定头文件目录？

编译时，加上"-I <头文件目录>"这样的选项。

③ 怎么确定交叉编译器中库文件的默认路径？

进入交叉编译器的目录里，执行：find -name lib，可以得到 xxxx/lib、xxxx/usr/lib，一般来说这 2 个目录就是要找的路径。如果有很多类似的 lib，进去看看，有很多 so 文件的目录一般就是要找的路径。

④ 怎么自己指定库文件目录、指定要用的库文件？

编译时，加上"-L <库文件目录>"这样的选项，用来指定库目录；

编译时，加上"-labc"这样的选项，用来指定库文件 libabc.so。

第2章 GCC 编译器的使用

源文件需要经过编译才能生成可执行文件。在 Windows 下进行开发时，只需要点几个按钮即可编译，集成开发环境(比如 Visual studio)已经将各种编译工具的使用封装好了。Linux 下也有很优秀的集成开发工具，但是更多的时候是直接使用编译工具；即使使用集成开发工具，也需要掌握一些编译选项。

PC 机上的编译工具链为 gcc、ld、objcopy、objdump 等，它们编译出来的程序在 x86 平台上运行。要编译出能在 ARM 平台上运行的程序，必须使用交叉编译工具 xxx-gcc、xxx-ld 等(不同版本的编译器的前缀不一样，比如 arm-linux-gcc)，下面分别介绍。

2.1 GCC 编译过程(精简版)

一个 C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和链接(linking)等 4 步才能变成可执行文件。

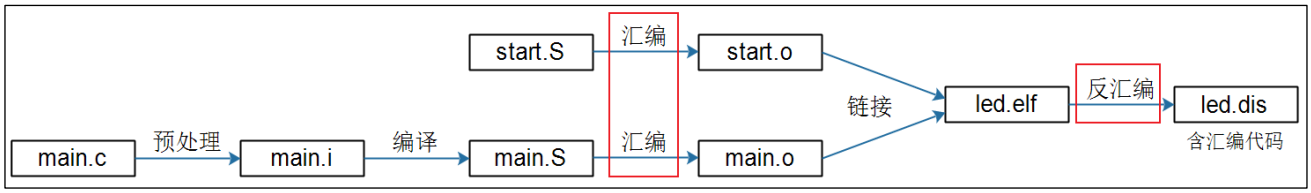


图 2.1 编译过程

通过不同的 gcc 选项可以控制这些过程：

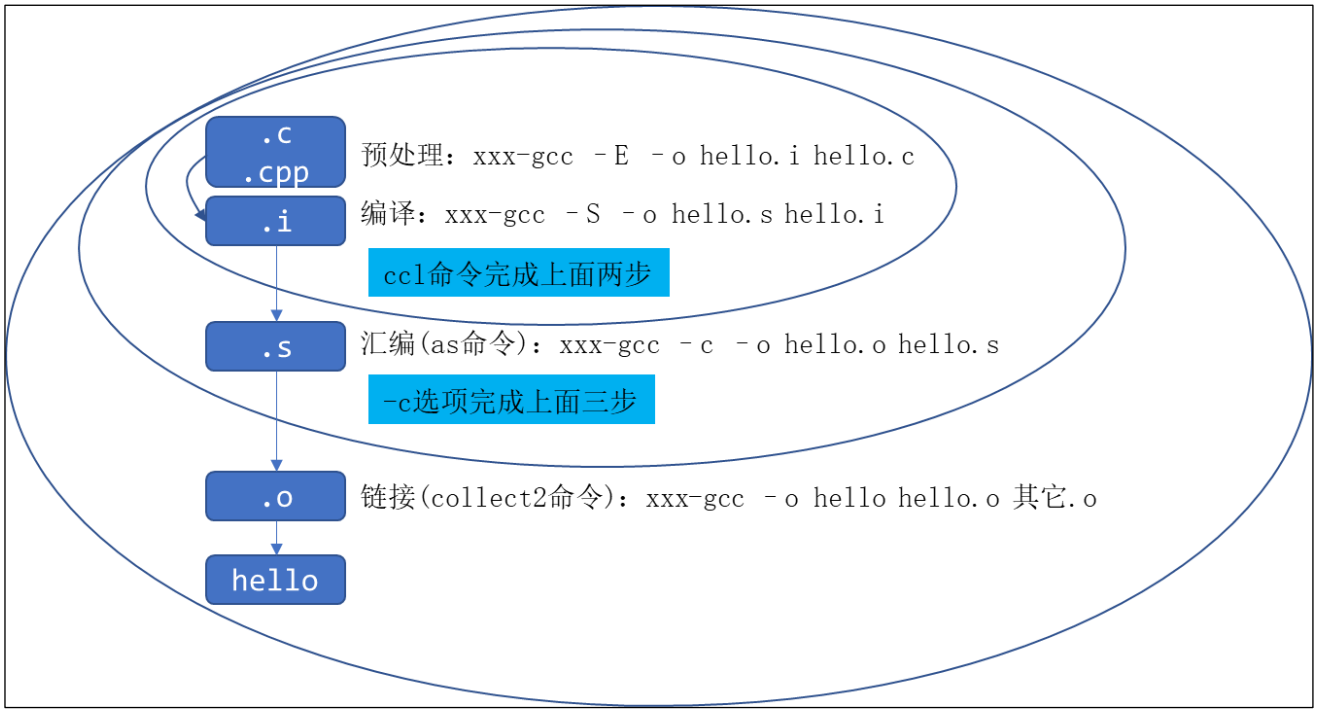


图 2.2 gcc 选项过程说明

在日常交流中通常使用“编译”统称这 4 个步骤，如果不是特指这 4 个步骤中的某一个，本教程也依惯例使用“编译”这个统称。

gcc 使用示例：

```
gcc hello.c                  // 输出一个名为 a.out 的可执行程序，然后可以执行./a.out
gcc -o hello hello.c          // 输出名为 hello 的可执行程序，然后可以执行./hello
gcc -o hello hello.c -static  // 静态链接
gcc -c -o hello.o hello.c  // 先编译(不链接)
gcc -o hello hello.o       // 再链接
```

执行“gcc -o hello hello.c -v”时，可以查看到这些步骤：

```
cc1 main.c -o /tmp/ccXCx1YG.s
as        -o /tmp/ccZfdaDo.o /tmp/ccXCx1YG.s

cc1 sub.c -o /tmp/ccXCx1YG.s
as        -o /tmp/ccn8Cjq6.o /tmp/ccXCx1YG.s

collect2 -o test /tmp/ccZfdaDo.o /tmp/ccn8Cjq6.o ....
```

可以手工执行以下命令体验一下：

```
gcc -E -o hello.i hello.c
gcc -S -o hello.s hello.i
gcc -c -o hello.o hello.s
gcc -o hello hello.o
```

2.1.1 常用编译选项

在学习时，我们暂时只需要了解表 2-1 中的选项：

表 2-1 GCC 常用编译选项

常用选项	描述
-E	预处理，开发过程中想快速确定某个宏可以使用“-E -dM”
-c	把预处理、编译、汇编都做了，但是不链接
-o	指定输出文件
-I	指定头文件目录
-L	指定链接时库文件目录
-l	指定链接哪一个库文件

2.1.2 怎么编译多个文件

① 一起编译、链接：

```
gcc  -o test  main.c  sub.c
```

② 分开编译，统一链接：

```
gcc -c -o main.o  main.c
gcc -c -o sub.o   sub.c
gcc -o test main.o sub.o
```

2.1.3 制作、使用动态库

第1步 制作、编译：

```
gcc -c -o main.o  main.c
gcc -c -o sub.o   sub.c
gcc -shared  -o libsub.so  sub.o  sub2.o  sub3.o(可以使用多个.o 生成动态库)
gcc -o test main.o  -lsub  -L /libsub.so/所在目录/
```

第2步 运行：
① 先把 libsub.so 放到 Ubuntu 的/lib 目录，然后就可以运行 test 程序。
② 如果不想把 libsub.so 放到/lib，也可以放在某个目录比如/a，然后如下执行：

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

2.1.4 制作、使用静态库

```
gcc -c -o main.o  main.c
gcc -c -o sub.o   sub.c
ar  crs  libsub.a  sub.o  sub2.o  sub3.o(可以使用多个.o 生成静态库)
gcc  -o  test  main.o  libsub.a  (如果.a 不在当前目录下，需要指定它的绝对或相对路径)
```

运行：不需要把静态库 libsub.a 放到板子上。
注意：执行 `arm-buildroot-linux-gnueabihf-gcc -c -o sub.o  sub.c` 交叉编译需要在最后面加上`-fPIC`参数。

2.1.5 很有用的选项

```
gcc -E main.c   // 查看预处理结果，比如头文件是哪个
gcc -E -dM main.c  > 1.txt  // 把所有的宏展开，存在 1.txt 里
gcc -Wp,-MD,abc.dep -c -o main.o main.c  // 生成依赖文件 abc.dep，后面 Makefile 会用
echo 'main(){}'| gcc -E -v -  // 它会列出头文件目录、库目录(LIBRARY_PATH)
```

下面的资料来自 GCC 官方文档及一些中文资料，没必要逐一研读，用到时再翻翻。

2.2 GCC 编译过程

一个 C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和链接(linking)等 4 步才能变成可执行文件，如图 2.1 和图 2.2 所示。

在日常交流中通常使用“编译”统称这 4 个步骤，如果不是特指这 4 个步骤中的某一个，本教程也依惯例使用“编译”这个统称。

本节文档使用 x86 上的 gcc 来试验，使用 ARM 板的交叉编译工具链做实验时效果也是类似的。不同的交叉编译器工具链前缀可能不同，比如 `arm-linux-gcc`。

2.2.1 预处理

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。

2.2.2 编译

编译就是把 C/C++代码(比如上述的“.i”文件)“翻译”成汇编代码，所用到的工具为 cc1(它的名字就是 cc1，x86 有自己的 cc1 命令，ARM 板也有自己的 cc1 命令)。

2.2.3 汇编

汇编就是将第二步输出的汇编代码翻译成符合一定格式的机器代码，在 Linux 系统上一般表现为 ELF 目标文件(OBJ 文件)，用到的工具为 as。x86 有自己的 as 命令，ARM 版也有自己的 as 命令，也可能是 xxxx-as（比如 arm-linux-as）。

“反汇编”是指将机器代码转换为汇编代码，这在调试程序时常常用到。

2.2.4 链接

链接就是将上步生成的 OBJ 文件和系统库的 OBJ 文件、库文件链接起来，最终生成了可以在特定平台运行的可执行文件，用到的工具为 ld 或 collect2。

编译程序时，加上-v 选项就可以看到这几个步骤。比如：

```
gcc -o hello hello.c -v
```

可以看到很多输出结果，我们把其中的主要信息摘出来：

```
cc1 hello.c -o /tmp/cctETob7.s
as -o /tmp/ccvv2KbL.o /tmp/cctETob7.s
collect2 -o hello crt1.o crti.o crtbegin.o /tmp/ccvv2KbL.o crtend.o crtn.o
```

以上 3 个命令分别对应于编译步骤中的预处理+编译、汇编和链接，ld 被 collect2 调用来链接程序。预处理和编译被放在了一个命令（cc1）中进行的，可以把它再次拆分为以下两步：

```
cpp -o hello.i hello.c
cc1 hello.i -o /tmp/cctETob7.s
```

我们不需要手工去执行 cpp、cc1、collect2 等命令，我们直接执行 gcc 并指定不同的参数就可以了。

可以通过如表 2-1 中的各种选项来控制 gcc 的动作。

2.3 GCC 总体选项(Overall Option)

① -c

预处理、编译和汇编源文件，但是不作链接，编译器根据源文件生成 OBJ 文件。缺省情况下，GCC 通过用`.o'替换源文件名的后缀`.c'，`.i'，`.s'等，产生 OBJ 文件名。可以使用-o 选项选择其他名字。GCC 忽略-c 选项后面任何无法识别的输入文件。

② -S

编译后即停止，不进行汇编。对于每个输入的非汇编语言文件，输出结果是汇编语言文件。缺省情况下，GCC 通过用`.s'替换源文件名后缀`.c'，`.i'等

等，产生汇编文件名。可以使用-o 选项选择其他名字。GCC 忽略任何不需要汇编的输入文件。

③ -E

预处理后即停止，不进行编译。预处理后的代码送往标准输出。

④ -o file

指定输出文件为 file。无论是预处理、编译、汇编还是链接，这个选项都可以使用。如果没有使用`-o'选项，默认的输出结果是：可执行文件为`a.out'；修改输入文件的名称是`source.suffix'，则它的 OBJ 文件是`source.o'，汇编文件是 `source.s'，而预处理后的 C 源代码送往标准输出。

⑤ -v

显示制作 GCC 工具自身时的配置命令；同时显示编译器驱动程序、预处理器、编译器的版本号。以一个程序为例，它包含三个文件，代码在 02_options 目录下。下面列出源码：

```
File: main.c
01 #include <stdio.h>
02 #include "sub.h"
03
04 int main(int argc, char *argv[])
05 {
06        int i;
07        printf("Main fun!\n");
08        sub_fun();
09        return 0;
10 }
11

File: sub.h
01 void sub_fun(void);
02

File: sub.c
01 void sub_fun(void)
02 {
03        printf("Sub fun!\n");
04 }
05
```

ARM版本的编译工具与gcc、ld等工具的使用方法相似，很多选项是一样的。本节使用 gcc、ld 等工具进行编译、链接，这样可以在 PC 上直接看到运行结果。使用上面介绍的选项进行编译，命令如下：

```
gcc  -c  -o  main.o  main.c
gcc  -c  -o  sub.o  sub.c
gcc  -o  test  main.o  sub.o
```

其中，main.o、sub.o 是经过了预处理、编译、汇编后生成的 OBJ 文件，它们还没有被链接成可执行文件；最后一步将它们链接成可执行文件 test，可以直接运行以下命令：

```
./test
Main fun!
```

```
Sub fun!
```

现在试试其他选项，以下命令生成的 main.s 是 main.c 的汇编语言文件：

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

以下命令对 main.c 进行预处理，并将得到的结果打印出来。里面扩展了所有包含的文件、所有定义的宏。在编写程序时，有时候查找某个宏定义是非常繁琐的事，可以使用`-dM –E’选项来查看。命令如下：

```
gcc  -E  main.c
```

2.4 警告选项(Warning Option)-Wall

这个选项基本打开了所有需要注意的警告信息，比如没有指定类型的声明、在声明之前就使用的函数、局部变量除了声明就没再使用等。

上面的 main.c 文件中，第 6 行定义的变量 i 没有被使用，但是使用“`gcc -c -o main.o main.c`”进行编译时并没有出现提示。

可以加上-Wall 选项，例子如下：

```
gcc -Wall -c main.c
```

执行上述命令后，得到如下警告信息：

```
main.c: In function `main':
main.c:6: warning: unused variable `i'
```

这个警告虽然对程序没有坏的影响，但是有些警告需要加以关注，比如类型匹配的警告等。

2.5 调试选项(Debugging Option)-g

以操作系统的本地格式(stabs，COFF，XCOFF，或 DWARF)产生调试信息，GDB 能够使用这些调试信息。在大多数使用 stabs 格式的系统上，`-g'选项加入只有 GDB 才使用的额外调试信息。可以使用下面的选项来生成额外的信息：`-gstabs+'，`-gstabs'，`-gxcoff+'，`-gxcoff'，`-gdwarf+'或`-gdwarf'，具体用法请读者参考 GCC 手册。

2.6 优化选项(Optimization Option)

① -O 或-O1

优化：对于大函数，优化编译的过程将占用稍微多的时间和相当大的内存。不使用`-O'或`-O1'选项的目的是减少编译的开销，使编译结果能够调试、语句是独立的：如果在两条语句之间用断点中止程序，可以对任何变量重新赋值，或者在函数体内把程序计数器指到其他语句，以及从源程序中精确地获取你所期待的结果。

不使用`-O'或`-O1'选项时，只有声明了 register 的变量才分配使用寄存器。

使用了`-O'或`-O1'选项，编译器会试图减少目标码的大小和执行时间。如

果指定了`-O'或`-O1'选项,,`-fthread-jumps'和`-fdefer-pop'选项将被打开。在有 delay slot 的机器上,`-fdelayed-branch'选项将被打开。在即使没有帧指针 (frame pointer)也支持调试的机器上,`-fomit-frame-pointer'选项将被打开。某些机器上还可能会打开其他选项。

② -O2

多优化一些。除了涉及空间和速度交换的优化选项,执行几乎所有的优化工作。例如不进行循环展开(loop unrolling)和函数内嵌(inlining)。和`-O'或`-O1'选项比较,这个选项既增加了编译时间,也提高了生成代码的运行效果。

③ -O3

优化的更多。除了打开-O2 所做的一切,它还打开了-finline-functions 选项。

④ -O0

不优化。

如果指定了多个-O 选项,不管带不带数字,生效的是最后一个选项。

在一般应用中,经常使用-O2 选项,比如对于 options 程序:

```
gcc -O2 -c -o main.o main.c
gcc -O2 -c -o sub.o sub.c
gcc -o test main.o sub.o
```

2.7 链接器选项(Linker Option)

下面的选项用于链接 OBJ 文件,输出可执行文件或库文件。

① object-file-name

如果某些文件没有特别明确的后缀(a special recognized suffix),GCC 就认为他们是 OBJ 文件或库文件(根据文件内容,链接器能够区分 OBJ 文件和库文件)。如果 GCC 执行链接操作,这些 OBJ 文件将成为链接器的输入文件。

比如上面的“gcc -o test main.o sub.o”中,main.o、sub.o 就是输入的文件。

② -llibrary

链接名为 library 的库文件。链接器在标准搜索目录中寻找这个库文件,库文件的真正名字是`liblibrary.a'。搜索目录除了一些系统标准目录外,还包括用户以`-L'选项指定的路径。一般说来用这个方法找到的文件是库文件——即由 OBJ 文件组成的归档文件(archive file)。链接器处理归档文件的方法是:扫描归档文件,寻找某些成员,这些成员的符号目前已被引用,不过还没有被定义。但是,如果链接器找到普通的 OBJ 文件,而不是库文件,就把这个 OBJ 文件按平常方式链接进来。指定`-l'选项和指定文件名的唯一区别是,`-l’选项用`lib'和`.a'把 library 包裹起来,而且搜索一些目录。

即使不明显地使用-llibrary 选项,一些默认的库也被链接进去,可以使用-v 选项看到这点:

```
gcc -v -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386 -dynami
c-linker /lib/ld-linux.so.2
-o test
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crt1.o /usr/lib/gcc-lib/i386-redha
t-linux/3.2.2/../../../crti.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtbegin.o
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
main.o
sub.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtend.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crtn.o
```

可以看见，除了 main.o、sub.o 两个文件外，还链接了启动文件 crt1.o、crti.o、crtend.o 、crtn.o，还有一些库文件(-lgcc -lgcc_eh -lc -lgcc -lgcc_eh)。

③ -nostartfiles

不链接系统标准启动文件，而标准库文件仍然正常使用：

```
gcc -v -nostartfiles -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386 -dynami
c-linker
/lib/ld-linux.so.2
-o test
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
main.o
sub.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

可以看见启动文件 crt1.o、crti.o、crtend.o 、crtn.o 没有被链接进去。需要说明的是，对于一般应用程序，这些启动文件是必需的，这里仅是作为例子(这样编译出来的 test 文件无法执行)。在编译 bootloader、内核时，将用到这个选项。

④ -nostdlib

不链接系统标准启动文件和标准库文件，只把指定的文件传递给链接器。这个选项常用于编译内核、bootloader 等程序，它们不需要启动文件、标准库文件。

仍以 options 程序作为例子：

```
gcc -v -nostdlib -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386 -dynami
c-linker /lib/ld-linux.so.2
-o test
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
```

```
main.o
sub.o
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048074
main.o(.text+0x19): In function `main':
: undefined reference to `printf'
sub.o(.text+0xf): In function `sub_fun':
: undefined reference to `printf'
collect2: ld returned 1 exit status
```

出现了一大堆错误，因为 printf 等函数是在库文件中实现的。在编译 bootloader、内核时，用到这个选项——它们用到的很多函数是自包含的。

⑤ -static

在支持动态链接(dynamic linking)的系统上，阻止链接共享库。

仍以 options 程序为例，是否使用-static 选项编译出来的可执行程序大小相差巨大：

```
gcc -c -o main.c
gcc -c -o sub.c
gcc -o test main.o sub.o
gcc -o test_static main.o sub.o -static
ls -l test test_static
-rwxr-xr-x 1 book book   6591 Jan 16 23:51 test
-rwxr-xr-x 1 book book 546479 Jan 16 23:51 test_static
```

其中 test 文件为 6591 字节，test_static 文件为 546479 字节。当不使用-static 编译文件时，程序执行前要链接共享库文件，所以还需要将共享库文件放入文件系统中。

⑥ -shared

生成一个共享 OBJ 文件，它可以和其他 OBJ 文件链接产生可执行文件。只有部分系统支持该选项。

当不想以源代码发布程序时，可以使用-shared 选项生成库文件，比如对于 options 程序，可以如下制作库文件：

```
gcc -c -o sub.o sub.c
gcc -shared -o libsub.so sub.o
```

以后要使用 sub.c 中的函数 sub_fun 时，在链接程序时，指定引脚 libsub.so 即可，比如：

```
gcc -o test main.o  -lsub  -L /libsub.so/所在的目录/
```

可以将多个文件制作为一个库文件，比如：

```
gcc -shared  -o libsub.so  sub.o  sub2.o  sub3.o
```

⑦ -Xlinker option

把选项 option 传递给链接器。可以用来传递系统特定的链接选项，GCC 无法识别这些选项。如果需要传递携带参数的选项，必须使用两次`-Xlinker'，一次传递选项，另一次传递其参数。例如，如果传递`-assert definitions'，要成`-Xlinker -assert -Xlinker definitions'，而不能写成`-Xlinker "-assert definitions"'，因为这样会把整个字符串当做一个参数传递，显然这不是链接器期待的。

⑧ -Wl,option

把选项 option 传递给链接器。如果 option 中含有逗号，就在逗号处分割成多个选项。链接器通常是通过 gcc、arm-linux-gcc 等命令间接启动的，要向它传入参数时，参数前面加上`-Wl,’。

⑨ -u symbol

使链接器认为取消了 symbol 的符号定义，从而链接库模块以取得定义。可以使用多个 `-u'选项，各自跟上不同的符号，使得链接器调入附加的库模块。

2.8 目录选项(Directory Option)

下列选项指定搜索路径，用于查找头文件，库文件，或编译器的某些成员。

① -Idir

在头文件的搜索路径列表中添加 dir 目录。

头文件的搜索方法为：如果以“#include < >”包含文件，则只在标准库目录开始搜索(包括使用-Idir 选项定义的目录)；如果以“#include “ ””包含文件，则先从用户的工作目录开始搜索，再搜索标准库目录。

② -I-

任何在`-I-'前面用`-I'选项指定的搜索路径只适用于`#include "file"'这种情况；它们不能用来搜索`#include <file>'包含的头文件。如果用`-I'选项指定的搜索路径位于`-I-'选项后面，就可以在这些路径中搜索所有的`#include'指令(一般说来-I 选项就是这么用的)。还有，`-I-'选项能够阻止当前目录(存放当前输入文件的地方)成为搜索`#include "file"'的第一选择。

`-I-'不影响使用系统标准目录，因此，`-I-'和`-nostdinc'是不同的选项。

③ -Ldir

在`-l'选项的搜索路径列表中添加 dir 目录。仍使用 options 程序进行说明，先制作库文件 libsub.a：

```
gcc -c -o sub.o sub.c
gcc -shared -o libsub.a sub.o
```

编译 main.c：

```
gcc  -c -o  main.o  main.c
```

链接程序，下面的指令将出错，提示找不到库文件：

```
gcc  -o  test  main.o  -lsub
/usr/bin/ld: cannot find -lsub
collect2: ld returned 1 exit status
```

可以使用-Ldir 选项将当前目录加入搜索路径，如下则链接成功：

```
gcc -L. -o test main.o -lsub
```

④ -Bprefix

这个选项指出在何处寻找可执行文件，库文件，以及编译器自己的数据文件。编译器驱动程序需要使用某些工具，比如：`cpp',`cc1' (或 C++的`cc1plus'),`as'和`ld'。它把 prefix 当作欲执行的工具的前缀，这个前缀可以用来指定目

录，也可以用来修改工具名字。

对于要运行的工具，编译器驱动程序首先试着加上`-B'前缀(如果存在)，如果没有找到文件，或没有指定`-B'选项，编译器接着会试验两个标准前缀`/usr/lib/gcc/'和`/usr/local/lib/gcc-lib/'。如果仍然没能够找到所需文件，编译器就在`PATH'环境变量指定的路径中寻找没加任何前缀的文件名。如果有需要，运行时(run-time)支持文件`libgcc.a'也在`-B'前缀的搜索范围之内。如果这里没有找到，就在上面提到的两个标准前缀中寻找，仅此而已。如果上述方法没有找到这个文件，就不链接它了。多数情况的多数机器上，`libgcc.a'并非必不可少。

可以通过环境变量 GCC_EXEC_PREFIX 获得近似的效果；如果定义了这个变量，其值就和上面说的一样被用作前缀。如果同时指定了`-B'选项和 GCC_EXEC_PREFIX 变量，编译器首先使用`-B'选项，然后才尝试环境变量值。

2.9 ld/objdump/objcopy 选项

我们在开发 APP 时，一般不需要直接调用这 3 个命令；在开发裸机、bootloader 时，或是调试 APP 时会涉及，到时再讲。

第3章 Makefile 的使用

在 Linux 中使用 make 命令来编译程序，特别是大程序；而 make 命令所执行的动作依赖于 Makefile 文件。最简单的 Makefile 文件如下：

```
hello: hello.c
gcc -o hello hello.c
clean:
rm -f  hello
```

将上述 4 行存为 Makefile 文件(注意必须以 Tab 键缩进第 2、4 行，不能以空格键缩进)，放入 01_hello 目录下，然后直接执行 make 命令即可编译程序，执行“make clean”即可清除编译出来的结果。

make 命令根据文件更新的时间戳来决定哪些文件需要重新编译，这使得可以避免编译已经编译过的、没有变化的程序，可以大大提高编译效率。

要想完整地了解 Makefile 的规则，请参考《GNU Make 使用手册》，以下仅粗略介绍。

3.1 配套视频内容大纲

3.1.1 Makefile 规则与示例

参考文档：https://www.gnu.org/software/make/manual/make.html

1 为什么需要 Makefile

怎么高效地编译程序？想达到什么样的效果？请参考 Visual Studio：修改源文件或头文件，只需要重新编译牵涉到的文件，就可以重新生成 APP

2 Makefile 其实挺简单

一个简单的 Makefile 文件包含一系列的“规则”，其样式如下：

```
目标(target)…: 依赖(prerequiries)…
<tab>命令(command)
```

如果“依赖文件”比“目标文件”更加新，那么执行“命令”来重新生成“目标文件”。

命令被执行的 2 个条件：依赖文件比目标文件新，或是 目标文件还没生成。

3 先介绍 Makefile 的 2 个函数

- $(foreach var,list,text)

简单地说，就是 `for each var in list, change it to text`。

对 list 中的每一个元素，取出来赋给 var，然后把 var 改为 text 所描述的形式。

例子：

```
objs := a.o b.o
dep_files := $(foreach  f,  $(objs),  .$(f).d)  // 最终 dep_files := .a.o.d .b.o.d
```

- $(wildcard pattern)

pattern 所列出的文件是否存在，把存在的文件都列出来。

例子：

```
src_files := $( wildcard  *.c)  // 最终 src_files 中列出了当前目录下的所有.c 文件
```

4 一步一步完善 Makefile

① 第 1 个 Makefile，简单粗暴，效率低：

```
test : main.c sub.c sub.h
gcc -o test main.c sub.c
```

② 第 2 个 Makefile，效率高，相似规则太多太啰嗦，不支持检测头文件：

```
test : main.o sub.o
gcc -o test main.o sub.o

main.o : main.c
gcc -c -o main.o  main.c

sub.o : sub.c
gcc -c -o sub.o  sub.c

clean:
rm *.o test -f
```

③ 第 3 个 Makefile，效率高，精炼，不支持检测头文件：

```
test : main.o sub.o
gcc -o test main.o sub.o
%.o : %.c
gcc -c -o $@  $<

clean:
rm *.o test -f
```

④ 第 4 个 Makefile，效率高，精炼，支持检测头文件(但是需要手工添加头文件规则)：

```
test : main.o sub.o
gcc -o test main.o sub.o

%.o : %.c
gcc -c -o $@  $<

sub.o : sub.h

clean:
rm *.o test -f
```

⑤ 第 5 个 Makefile，效率高，精炼，支持自动检测头文件：

```
objs := main.o sub.o

test : $(objs)
    gcc -o test $^


# 需要判断是否存在依赖文件
# .main.o.d .sub.o.d
dep_files := $(foreach f, $(objs), .$(f).d)
dep_files := $(wildcard $(dep_files))
```

```
# 把依赖文件包含进来
ifneq ($(dep_files),)
  include $(dep_files)
endif


%.o : %.c
    gcc -Wp,-MD,.$@.d  -c -o $@  $<


clean:
    rm *.o test -f

distclean:
    rm  $(dep_files) *.o test -f
```

3.1.2 通用 Makefile 的使用

我参考 Linux 内核的 Makefile 编写了一个通用的 Makefile，它可以用来编译应用程序：

① 支持多个目录、多层目录、多个文件；
② 支持给所有文件设置编译选项；
③ 支持给某个目录设置编译选项；
④ 支持给某个文件单独设置编译选项；
⑤ 简单、好用。

使用 git 下载本教程的文档后，下列目录中就有说明和示例：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\05_general_Makefile
```

3.1.3 通用 Makefile 的解析

- make 命令的使用：

执行 make 命令时，它会去当前目录下查找名为“Makefile”的文件，并根据它的指示去执行操作，生成第一个目标。

我们可以使用“-f”选项指定文件，不再使用名为“Makefile”的文件，比如：

```
make  -f  Makefile.build
```

我们可以使用“-C”选项指定目录，切换到其他目录里去，比如：

```
make -C  a/  -f  Makefile.build
```

我们可以指定目标，不再默认生成第一个目标：

```
make -C  a/  -f  Makefile.build   other_target
```

- 即时变量、延时变量：

变量的定义语法形式如下：

```
A  =  xxx    // 延时变量
B  ?= xxx    // 延时变量，只有第一次定义时赋值才成功；如果曾定义过，此赋值无效
C  := xxx    // 立即变量
D  += yyy    // 如果 D 在前面是延时变量，那么现在它还是延时变量；
```

```
// 如果 D 在前面是立即变量，那么现在它还是立即变量
```

在 GNU make 中对变量的赋值有两种方式：延时变量、立即变量。上面语句中，变量 A 是延时变量，它的值在使用时才展开、才确定。比如：

```
A  =  $@
test:
    @echo $A
```

上述 Makefile 中，变量 A 的值在执行时才确定，它等于 test，是延时变量。如果使用“A := $@”，这是立即变量，这时$@为空，所以 A 的值就是空。

- 变量的导出(export)：

在编译程序时，我们会不断地使用“`make -C dir`”切换到其他目录，执行其他目录里的 Makefile。如果想让某个变量的值在所有目录中都可见，要把它 export 出来。

比如“`CC = $(CROSS_COMPILE)gcc`”，这个 CC 变量表示编译器，在整个过程中都是一样的。定义它之后，要使用“export CC”把它导出来。

- Makefile 中可以使用 shell 命令：

比如：

```
TOPDIR := $(shell pwd)
```

这是个立即变量，TOPDIR 等于 shell 命令 pwd 的结果。

- 在 Makefile 中怎么放置第 1 个目标：

执行 make 命令时如果不指定目标，那么它默认是去生成第 1 个目标。

所以“第 1 个目标”，位置很重要。有时候不太方便把第 1 个目标完整地放在文件前面，这时可以在文件的前面直接放置目标，在后面再完善它的依赖与命令。比如：

```
First_target:   // 这句话放在前面
....          // 其他代码，比如 include 其他文件得到后面的 xxx 变量
First_target : $(xxx)   $(yyy)   // 在文件的后面再来完善
    command
```

- 假想目标：

我们的 Makefile 中有这样的目标：

```
clean:
    rm -f $(shell find -name "*.o")
    rm -f $(TARGET)
```

如果当前目录下恰好有名为“clean”的文件，那么执行“`make clean`”时它就不会执行那些删除命令。这时我们需要把“clean”这个目标，设置为“假想目标”，这样可以确保执行“`make clean`”时那些删除命令肯定可以得到执行。

使用下面的语句把“clean”设置为假想目标：

```
.PHONY : clean
```

- 常用的函数：

① $(foreach var,list,text)

简单地说，就是`for each var in list, change it to text`。对 list 中的每一个元素，取出来赋给 var，然后把 var 改为 text 所描述的形式。

例子：

```
objs := a.o b.o
dep_files := $(foreach  f,  $(objs),  .$(f).d)  // 最终 dep_files := .a.o.d .b.o.d
```

② $(wildcard pattern)

pattern 所列出的文件是否存在，把存在的文件都列出来。

例子：

```
src_files := $( wildcard  *.c)  // 最终 src_files 中列出了当前目录下的所有.c 文件
```

③ $(filter pattern...,text)

把 text 中符合 pattern 格式的内容，filter(过滤)出来、留下来。

例子：

```
obj-y := a.o b.o c/ d/
DIR :=  $(filter  %/,  $(obj-y))   //结果为：c/ d/
```

④ $(filter-out pattern...,text)

把 text 中符合 pattern 格式的内容，filter-out(过滤)出来、扔掉。

例子：

```
obj-y := a.o b.o c/ d/
DIR :=  $(filter-out  %/,  $(obj-y))   //结果为：a.o  b.o
```

⑤ $(patsubst pattern,replacement,text)

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。

比如：

```
subdir-y    :=  c/  d/
subdir-y    := $(patsubst  %/,  %,  $(subdir-y))   // 结果为：c  d
```

3.1.4 通用 Makefile 的设计思想

- 在 Makefile 文件中确定要编译的文件、目录，比如：

```
obj-y += main.o
obj-y += a/
```

“Makefile”文件总是被“Makefile.build”包含的。

- 在 Makefile.build 中设置编译规则，有 3 条编译规则：

① 怎么编译子目录？ 进入子目录编译：

```
$(subdir-y):
    make -C $@ -f $(TOPDIR)/Makefile.build
```

② 怎么编译当前目录中的文件？

```
%.o : %.c
    $(CC) $(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$@) -Wp,-MD,$(dep_file) -c -o $@ $<
```

③ 当前目录下的.o 和子目录下的 built-in.o 要打包起来：

```
built-in.o : $(cur_objs) $(subdir_objs)
    $(LD) -r -o $@ $^
```

- 顶层 Makefile 中把顶层目录的 built-in.o 链接成 APP：

```
$(TARGET) : built-in.o
    $(CC) $(LDFLAGS) -o $(TARGET) built-in.o
```

- 情景演绎

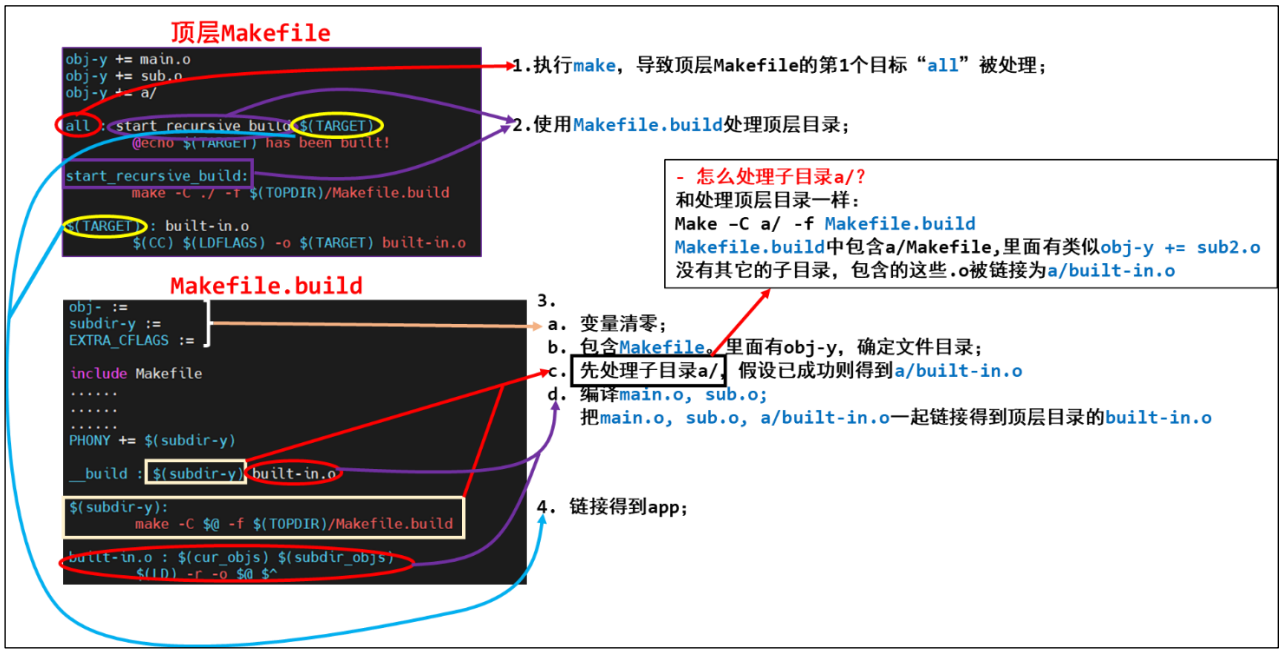


图 3.1 Makefile 情景演绎

3.2 Makefile 规则

一个简单的 Makefile 文件包含一系列的“规则”，其样式如下：

```
目标(target)…: 依赖(prerequiries)…
<tab>命令(command)
```

目标(target)通常是要生成的文件的名称，可以是可执行文件或OBJ文件，也可以是一个执行的动作名称，诸如`clean’。

依赖是用来产生目标的材料(比如源文件)，一个目标经常有几个依赖。

命令是生成目标时执行的动作，一个规则可以含有几个命令，每个命令占一行。

注意：每个命令行前面必须是一个 Tab 字符，即命令行第一个字符是 Tab。这是容易出错的地方。

通常，如果一个依赖发生了变化，就需要规则调用命令以更新或创建目标。但是并非所有的目标都有依赖，例如，目标“clean”的作用是清除文件，它没有依赖。

规则一般是用于解释怎样和何时重建目标。make 首先调用命令处理依赖，进而才能创建或更新目标。当然，一个规则也可以是用于解释怎样和何时执行一个动作，即打印提示信息。

一个Makefile文件可以包含规则以外的其他文本，但一个简单的Makefile文件仅仅需要包含规则。虽然真正的规则比这里展示的例子复杂，但格式是完全一样的。

对于上面的 Makefile，执行“make”命令时，仅当 hello.c 文件比 hello 文件新，才会执行命令“arm-linux-gcc -o hello hello.c”生成可执行文件 hello；

如果还没有 hello 文件，这个命令也会执行。

运行“`make clean`”时，由于目标 clean 没有依赖，它的命令“`rm -f hello`”将被强制执行。

3.3 Makefile 文件里的赋值方法

变量的定义语法形式如下：

```
immediate = deferred
immediate ?= deferred
immediate := immediate
immediate += deferred or immediate
define immediate
deferred
endef
```

在 GNU make 中对变量的赋值有两种方式：延时变量、立即变量。区别在于它们的定义方式和扩展时的方式不同，前者在这个变量使用时才扩展开，意即当真正使用时这个变量的值才确定；后者在定义时它的值就已经确定了。使用`=’，`?=’定义或使用 define 指令定义的变量是延时变量；使用`:=’定义的变量是立即变量。需要注意的一点是，`?=’仅仅在变量还没有定义的情况下有效，即`?=’被用来定义第一次出现的延时变量。

对于附加操作符`+=’，右边变量如果在前面使用（:=）定义为立即变量则它也是立即变量，否则均为延时变量。

3.4 Makefile 常用函数

函数调用的格式如下：

```
$(function arguments)
```

这里`function’是函数名，`arguments’是该函数的参数。参数和函数名之间是用空格或 Tab 隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。内核的 Makefile 中用到大量的函数，现在介绍一些常用的。

3.4.1 字符串替换和分析函数

1 $(subst from,to,text)

在文本`text’中使用`to’替换每一处`from’。

比如：`$(subst ee,EE,feet on the street)`

结果为‘`fEEt on the strEEt`’。

2 $(patsubst pattern,replacement,text)

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。

比如：`$(patsubst %.c,%.o,x.c.c bar.c)`

结果为：`x.c.o bar.o’。

3 $(strip string)

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。

比如：`$(strip a  b c )`

结果为`a b c’。

4 $(findstring find,in)

在字符串`in’中搜寻`find’，如果找到，则返回值是`find’，否则返回值为空。

比如：

```
$(findstring a,a b c)
$(findstring a,b c)
```

将分别产生值`a’和`’(空字符串)。

5 $(filter pattern...,text)

返回在`text’中由空格隔开且匹配格式`pattern...’的字，去除不符合格式`pattern...’的字。

比如：`$(filter %.c %.s,foo.c bar.c baz.s ugh.h)`

结果为`foo.c bar.c baz.s’。

6 $(filter-out pattern...,text)

返回在`text’中由空格隔开且不匹配格式`pattern...’的字，去除符合格式`pattern...’的字。它是函数 filter 的反函数。

比如：`$(filter %.c %.s,foo.c bar.c baz.s ugh.h)`

结果为`ugh.h’。

7 $(sort list)

将‘list’中的字按字母顺序排序，并去掉重复的字。输出由单个空格隔开的字的列表。

比如：`$(sort foo bar lose)`

返回值是‘bar foo lose’。

3.4.2 文件名函数

1 $(dir names...)

抽取‘names...’中每一个文件名的路径部分，文件名的路径部分包括从文件名的首字符到最后一个斜杠(含斜杠)之前的一切字符。

比如：`$(dir src/foo.c hacks)`

结果为‘src/ ./’。

2 $(notdir names...)

抽取‘names...’中每一个文件名中除路径部分外一切字符（真正的文件名）。

比如：`$(notdir src/foo.c hacks)`
结果为‘`foo.c hacks`’。

3 **$(suffix names...)**
抽取‘names...’中每一个文件名的后缀。
比如：`$(suffix src/foo.c src-1.0/bar.c hacks)`
结果为‘`.c .c`’。

4 **$(basename names...)**
抽取‘names...’中每一个文件名中除后缀外一切字符。
比如：`$(basename src/foo.c src-1.0/bar hacks)`
结果为‘`src/foo src-1.0/bar hacks`’。

5 **$(addsuffix suffix,names...)**
参数‘names...’是一系列的文件名，文件名之间用空格隔开；suffix 是一个后缀名。将 suffix(后缀)的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用单个空格隔开。
比如：`$(addsuffix .c,foo bar)`
结果为‘`foo.c bar.c`’。

6 **$(addprefix prefix,names...)**
参数‘names’是一系列的文件名，文件名之间用空格隔开；prefix 是一个前缀名。将 preffix(前缀)的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。
比如：`$(addprefix src/,foo bar)`
结果为‘`src/foo src/bar`’。

7 **$(wildcard pattern)**
参数‘pattern’是一个文件名格式，包含有通配符(通配符和 shell 中的用法一样)。函数 wildcard 的结果是一列和格式匹配的且真实存在的文件的名称，文件名之间用一个空格隔开。
比如若当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(wildcard *.c)
```

结果为‘`1.c 2.c`’。

3.4.3 其他函数

1 $(foreach var,list,text)
前两个参数，‘var’和‘list’将首先扩展，注意最后一个参数‘text’此时不扩展；接着，‘list’扩展所得的每个字，都赋给‘var’变量；然后‘text’引用该变量进行扩展，因此‘text’每次扩展都不相同。
函数的结果是由空格隔开的‘text’ 在‘list’中多次扩展后，得到的新

‘list’，就是说：‘text’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数 foreach 的返回值。

下面是一个简单的例子，将变量‘files’的值设置为‘dirs’中的所有目录下的所有文件的列表：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

这里‘text’是‘$(wildcard $(dir)/*)’，它的扩展过程如下：

① 第一个赋给变量 dir 的值是`a’，扩展结果为‘$(wildcard a/*)’；

② 第二个赋给变量 dir 的值是`b’，扩展结果为‘$(wildcard b/*)’；

③ 第三个赋给变量 dir 的值是`c’，扩展结果为‘$(wildcard c/*)’；

④ 如此继续扩展。

这个例子和下面的例有共同的结果：

```
files := $(wildcard a/* b/* c/* d/*)
```

2 $(if condition,then-part[,else-part])

首先把第一个参数‘condition’的前导空格、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件‘condition’为‘真’；如果扩展为空字符串，则条件‘condition’为‘假’。

如果条件‘condition’为‘真’,那么计算第二个参数‘then-part’的值，并将该值作为整个函数 if 的值。

如果条件‘condition’为‘假’,并且第三个参数存在，则计算第三个参数‘else-part’的值，并将该值作为整个函数 if 的值；如果第三个参数不存在，函数 if 将什么也不计算，返回空值。

注意：仅能计算‘then-part’和‘else-part’二者之一，不能同时计算。这样有可能产生副作用（例如函数 shell 的调用）。

3 $(origin variable)

变量‘variable’是一个查询变量的名称，不是对该变量的引用。所以，不能采用‘$’和圆括号的格式书写该变量，当然，如果需要使用非常量的文件名，可以在文件名中使用变量引用。

函数 origin 的结果是一个字符串，该字符串变量是这样定义的：

```
‘undefined'              ：如果变量‘variable’从没有定义；
‘default'            ：变量‘variable’是缺省定义；
‘environment'        ：变量‘variable’作为环境变量定义，选项‘-e’没有打开；
‘environment override'   ：变量‘variable’作为环境变量定义，选项‘-e’已打开；
‘file'                   ：变量‘variable’在 Makefile 中定义；
‘command line'           ：变量‘variable’在命令行中定义；
‘override'               ：变量‘variable’在 Makefile 中用 override 指令定义；
‘automatic'          ：变量‘variable’是自动变量
```

4 $(shell command arguments)

函数 shell 是 make 与外部环境的通讯工具。函数 shell 的执行结果和在控制台上执行‘command arguments’的结果相似。不过如果‘command arguments’

的结果含有换行符（和回车符），则在函数 shell 的返回结果中将把它们处理为单个空格，若返回结果最后是换行符（和回车符）则被去掉。

比如当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(shell ls *.c)
```

结果为‘1.c 2.c’。

《Makefile 介绍》这小节可以在阅读内核、bootloader、应用程序的 Makefile 文件时，作为手册来查询。下面以 options 程序的 Makefile 作为例子进行演示，Makefile 的内容如下：

```
File: Makefile
01 src  := $(shell ls *.c)
02 objs := $(patsubst %.c,%.o,$(src))
03
04 test: $(objs)
05     gcc -o $@ $^
06
07 %.o:%.c
08     gcc -c -o $@ $<
09
10 clean:
11     rm -f test *.o
```

上述 Makefile 中$@、$^、$<称为自动变量。$@表示规则的目标文件名；$^表示所有依赖的名字，名字之间用空格隔开；$<表示第一个依赖的文件名。‘%’是通配符，它和一个字符串中任意个数的字符相匹配。

options 目录下所有的文件为 main.c，Makefile，sub.c 和 sub.h，下面一行行地分析：

① 第 1 行 src 变量的值为‘main.c sub.c’。

② 第 2 行 objs 变量的值为‘main.o sub.o’，是 src 变量经过 patsubst 函数处理后得到的。

③ 第 4 行实际上就是：

```
test : main.o sub.o
```

目标 test 的依赖有二：main.o 和 sub.o。开始时这两个文件还没有生成，在执行生成 test 的命令之前先将 main.o、sub.o 作为目标查找到合适的规则，以生成 main.o、sub.o。

④ 第 7、8 行就是用来生成 main.o、sub.o 的规则：

对于 main.o 这个规则就是：

```
main.o:main.c
    gcc -c -o main.o main.c
```

对于 sub.o 这个规则就是：

```
sub.o:sub.c
    gcc -c -o sub.o sub.c
```

这样，test 的依赖 main.o 和 sub.o 就生成了。

⑤ 第 5 行的命令在生成 main.o、sub.o 后得以执行。

在 options 目录下第一次执行 make 命令可以看到如下信息：

```
gcc -c -o main.o main.c
```

```
gcc -c -o sub.o sub.c
gcc -o test main.o sub.o
```

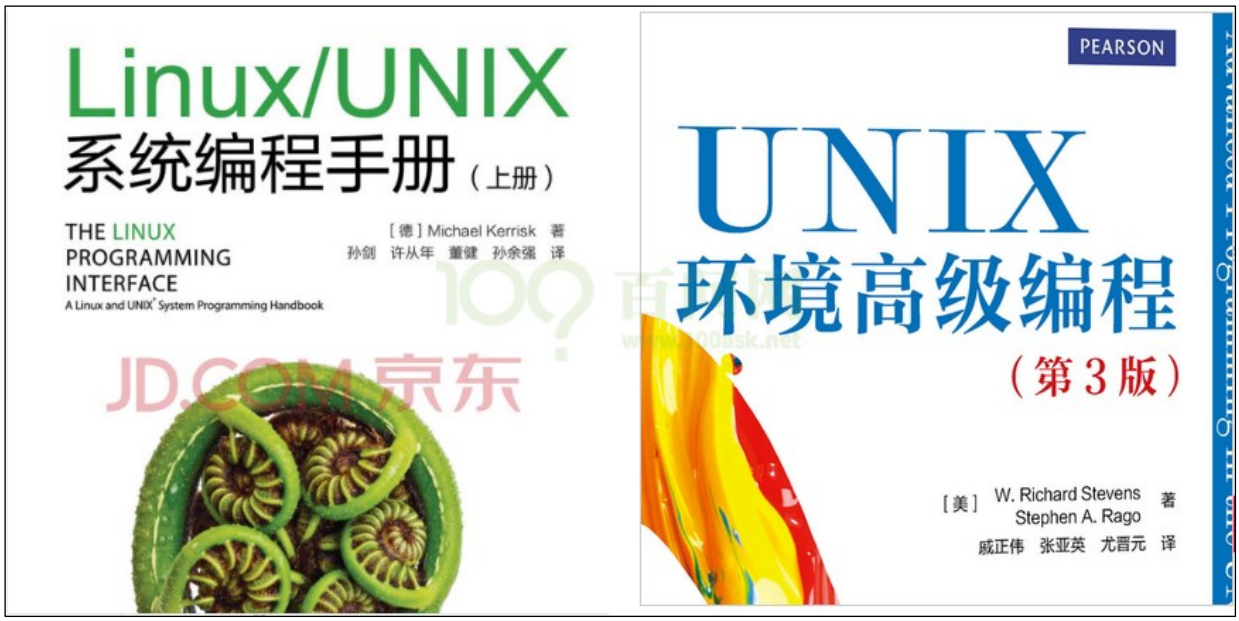
然后修改 sub.c 文件，再次执行 make 命令，可以看到如下信息：

```
gcc -c -o sub.o sub.c
gcc -o test main.o sub.o
```

可见，只编译了更新过的 sub.c 文件，对 main.c 文件不用再次编译，节省了编译的时间。

第4章 文件 IO

参考书：



这 2 本书的内容类似，第一本对知识点有更细致的描述，适合初学者；第二本比较直接，一上来就是各种函数的介绍，适合当作字典，不懂时就去翻看一下。

做纯 Linux 应用的入门，看这 2 本书就可以了，不需要学习我们的视频。我们的侧重于“**嵌入式 Linux**”。

在 Linux 系统中，一切都是“文件”：普通文件、驱动程序、网络通信等等。所有的操作，都是通过“文件 IO”来操作的。所以，很有必要掌握文件操作的常用接口。

4.1 文件从哪来？

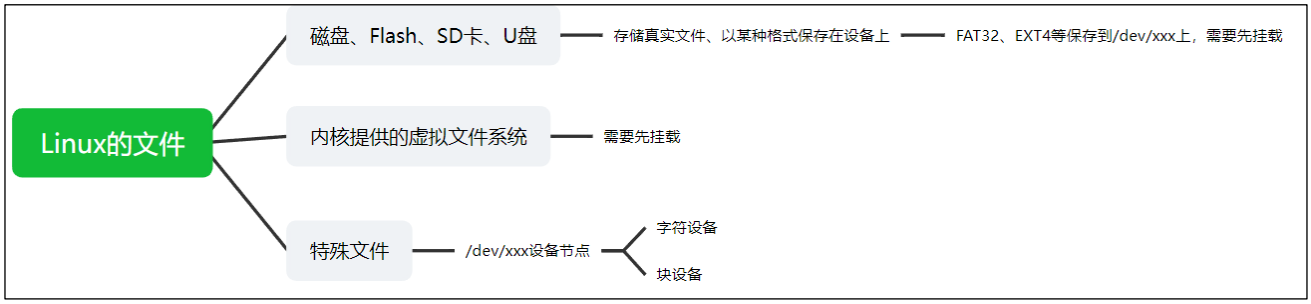


图 4.1 Linux 系统的文件

如图 4.1 所示，Linux 的文件既可以是真实保存到存储介质的文件也可以是自身内核提供的虚拟文件，还可以是设备节点。

4.2 怎么访问文件？

4.2.1 通用的 IO 模型：open/read/write/lseek/close

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\06_fileio\copy.c
```

copy.c 源码如下：

```
02 #include <sys/types.h>
03 #include <sys/stat.h>
04 #include <fcntl.h>
05 #include <unistd.h>
06 #include <stdio.h>
07
08 /*
09  * ./copy 1.txt 2.txt
10  * argc    = 3
11  * argv[0] = "./copy"
12  * argv[1] = "1.txt"
13  * argv[2] = "2.txt"
14  */
15 int main(int argc, char **argv)
16 {
17     int fd_old, fd_new;
18     char buf[1024];
19     int len;
20
21     /* 1. 判断参数 */
22     if (argc != 3)
23     {
24             printf("Usage: %s <old-file> <new-file>\n", argv[0]);
25             return -1;
26     }
27
28     /* 2. 打开老文件 */
29     fd_old = open(argv[1], O_RDONLY);
30     if (fd_old == -1)
31     {
32             printf("can not open file %s\n", argv[1]);
33             return -1;
34     }
35
36     /* 3. 创建新文件 */
37     fd_new = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP |
 S_IWGRP | _IROTH | S_IWOTH);
38     if (fd_new == -1)
39     {
40             printf("can not creat file %s\n", argv[2]);
41             return -1;
42     }
43
44     /* 4. 循环： 读老文件-写新文件 */
45     while ((len = read(fd_old, buf, 1024)) > 0)
46     {
```

```
47              if (write(fd_new, buf, len) != len)
48              {
49                      printf("can not write %s\n", argv[2]);
50                      return -1;
51              }
52      }
53
54      /* 5. 关闭文件 */
55      close(fd_old);
56      close(fd_new);
57
58      return 0;
59 }
60
```

本节源码完全可以在 Ubuntu 上测试，跟在 ARM 板上没什么不同。

执行以下命令编译、运行：

```
gcc -o copy copy.c
./copy  copy.c new.c
```

4.2.2 不是通用的函数：ioctl/mmap

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\06_fileio\copy_mmap.c
```

在 Linux 中，还可以把一个文件的所有内容映射到内存，然后直接读写内存即可读写文件。

copy_mmap.c 源码如下：

```
02 #include <sys/types.h>
03 #include <sys/stat.h>
04 #include <fcntl.h>
05 #include <unistd.h>
06 #include <stdio.h>
07 #include <sys/mman.h>
08
09 /*
10  * ./copy 1.txt 2.txt
11  * argc    = 3
12  * argv[0] = "./copy"
13  * argv[1] = "1.txt"
14  * argv[2] = "2.txt"
15  */
16 int main(int argc, char **argv)
17 {
18      int fd_old, fd_new;
19      struct stat stat;
20      char *buf;
21
22      /* 1. 判断参数 */
23      if (argc != 3)
24      {
25              printf("Usage: %s <old-file> <new-file>\n", argv[0]);
26              return -1;
27      }
```

```
28
29      /* 2. 打开老文件 */
30      fd_old = open(argv[1], O_RDONLY);
31      if (fd_old == -1)
32      {
33              printf("can not open file %s\n", argv[1]);
34              return -1;
35      }
36
37      /* 3. 确定老文件的大小 */
38      if (fstat(fd_old, &stat) == -1)
39      {
40              printf("can not get stat of file %s\n", argv[1]);
41              return -1;
42      }
43
44      /* 4. 映射老文件 */
45      buf = mmap(NULL, stat.st_size, PROT_READ, MAP_SHARED, fd_old, 0);
46      if (buf == MAP_FAILED)
47      {
48              printf("can not mmap file %s\n", argv[1]);
49              return -1;
50      }
51
52      /* 5. 创建新文件 */
53      fd_new = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRG
RP | S_IWGRP | S_IROTH | S_IWOTH);
54      if (fd_new == -1)
55      {
56              printf("can not creat file %s\n", argv[2]);
57              return -1;
58      }
59
60      /* 6. 写新文件 */
61      if (write(fd_new, buf, stat.st_size) != stat.st_size)
62      {
63              printf("can not write %s\n", argv[2]);
64              return -1;
65      }
66
67      /* 5. 关闭文件 */
68      close(fd_old);
69      close(fd_new);
70
71      return 0;
72 }
```

本节源码完全可以在 Ubuntu 上测试，跟在 ARM 板上没什么不同。

执行以下命令编译、运行：

```
gcc -o copy_mmap copy_mmap.c
./copy_mmap  copy_mmap.c  new2.c
```

4.3 怎么知道这些函数的用法？

Linux 下有 3 大帮助方法：help、man、info。

想查看某个命令的用法时，比如查看 ls 命令的用法，可以执行：

```
ls  --help
```

help 只能用于查看某个命令的用法，而 man 手册既可以查看命令的用法，还可以查看函数的详细介绍等等。它含有 9 大分类，如下：

```
1   Executable programs or shell commands       // 命令
2   System calls (functions provided by the kernel)  // 系统调用，比如 man 2 open
3   Library calls (functions within program libraries)  // 函数库调用
4   Special files (usually found in /dev)             // 特殊文件，比如 man 4 tty
5   File formats and conventions eg /etc/passwd  // 文件格式和约定，比如 man 5 passwd
6   Games  // 游戏
7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7) /
/杂项
8   System administration commands (usually only for root) // 系统管理命令
9   Kernel routines [Non standard]  // 内核例程
```

比如想查看 open 函数的用法时，可以直接执行“man open”，发现这不是想要内容时再执行“man 2 open”。

在 man 命令中可以及时按“h”查看帮助信息了解快捷键。常用的快捷键是：

```
f  往前翻一页
b  往后翻一页
/patten 往前搜
?patten 往后搜
```

就内容来说，info 手册比 man 手册编写得要更全面，但 man 手册使用起来更容易些。

以书来形容 info 手册和 man 手册的话，info 手册相当于一章，里面含有若干节，阅读时你需要掌握如果从这一节跳到下一节；而 man 手册只相当于一节，阅读起来当然更容易。

就个人而言，我很少使用 info 命令。

可以直接执行“info”命令后，输入“H”查看它的快捷键，在 info 手册中，某一节被称为“node”，常用的快捷键如下：

```
Up          Move up one line.
Down        Move down one line.
PgUp        Scroll backward one screenful.
PgDn        Scroll forward one screenful.
Home        Go to the beginning of this node.
End         Go to the end of this node.
TAB         Skip to the next hypertext link.
RET         Follow the hypertext link under the cursor.
l           Go back to the last node seen in this window.

[           Go to the previous node in the document.
]           Go to the next node in the document.
p           Go to the previous node on this level.
n           Go to the next node on this level.
u           Go up one level.
t           Go to the top node of this document.
d           Go to the main 'directory' node.
```

4.4 系统调用函数怎么进入内核？

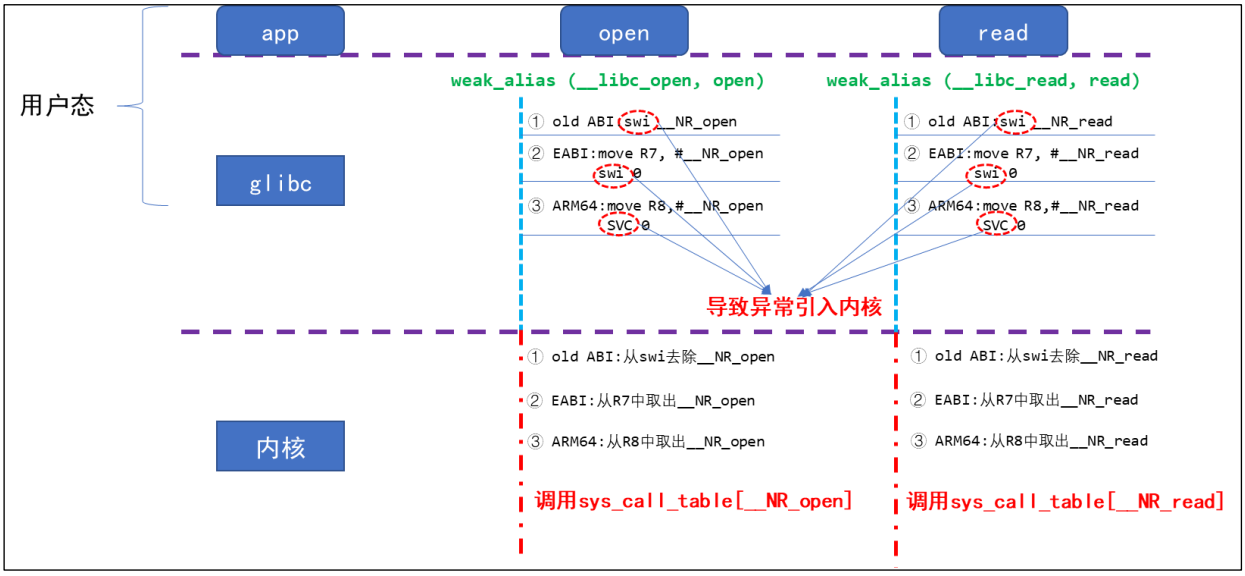


图 4.2 进入内核流程示意

以 open/read 为例，从用户态调用 API 触发异常进入内核的过程如图 4.2 所示，最后调用的 sys_call_table 的函数指针数组如下：

```
/* 0 */ CALL(sys_restart_syscall)
        CALL(sys_exit)
        CALL(sys_fork)
        CALL(sys_read)
        CALL(sys_write)
/* 5 */ CALL(sys_open)
        CALL(sys_close)
        CALL(sys_ni_syscall)
        CALL(sys_creat)
        CALL(sys_link)
/* 10 */CALL(sys_unlink)
        CALL(sys_execve)
        CALL(sys_chdir)
        CALL(OBSULETE(sys_time))
```

4.5 内核的 sys_open、sys_read 会做什么？

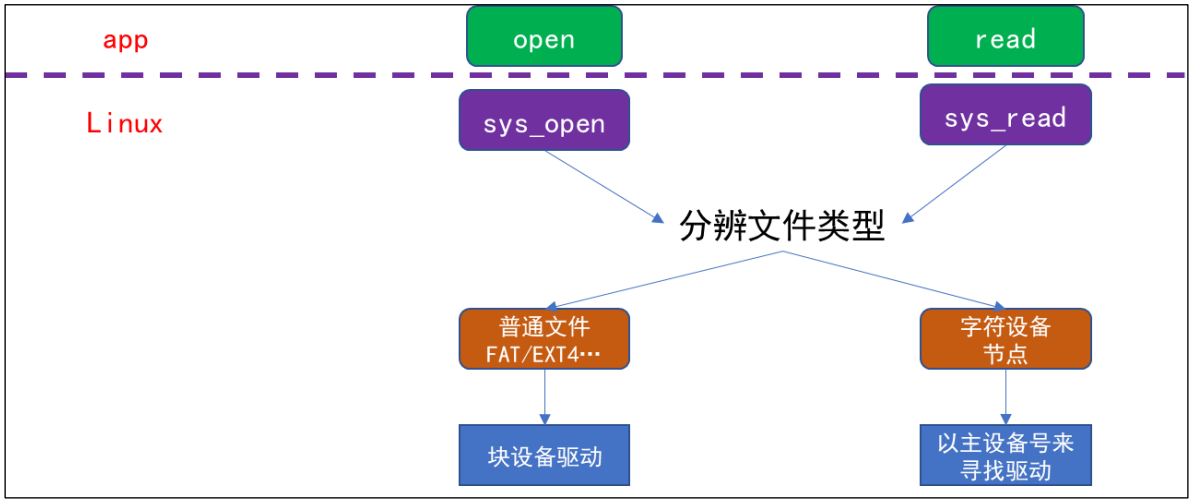


图 4.3 sys_open/read 具体行为

从图 4.3 可以看出，进入内核后，sys_read/open 会首先根据参数判断文件的类型，然后根据不同的文件类型去找不同的设备驱动，继而进行读写或者输入输出控制。

第5章 Framebuffer 应用编程

5.1 LCD 操作原理

在 Linux 系统中通过 Framebuffer 驱动程序来控制 LCD。Frame 是帧的意思，buffer 是缓冲的意思，这意味着 Framebuffer 就是一块内存，里面保存着一帧图像。Framebuffer 中保存着一帧图像的每一个像素颜色值，假设 LCD 的分辨率是 1024x768，每一个像素的颜色用 32 位来表示，那么 Framebuffer 的大小就是：1024x768x32/8=3145728 字节。

简单介绍 LCD 的操作原理：

① 驱动程序设置好 LCD 控制器：

根据 LCD 的参数设置 LCD 控制器的时序、信号极性；

根据 LCD 分辨率、BPP 分配 Framebuffer。

② APP 使用 ioctl 获得 LCD 分辨率、BPP

③ APP 通过 mmap 映射 Framebuffer，在 Framebuffer 中写入数据

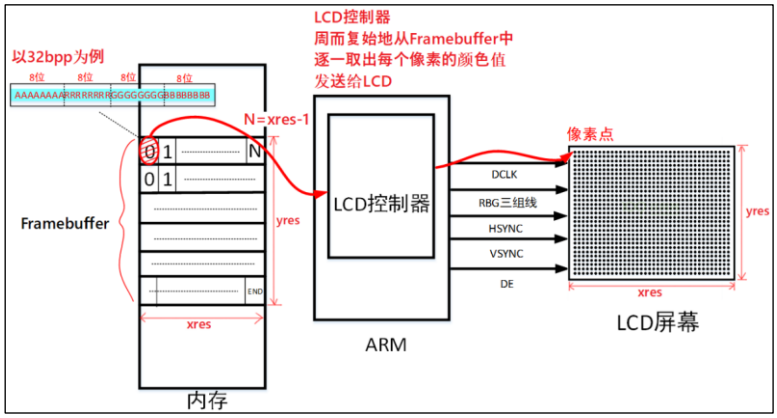


图 5.1 LCD 操作原理示意图

假设需要设置 LCD 中坐标(x,y)处像素的颜色，首要要找到这个像素对应的内存，然后根据它的 BPP 值设置颜色。假设 fb_base 是 APP 执行 mmap 后得到的 Framebuffer 地址，如图 5.2 所示：

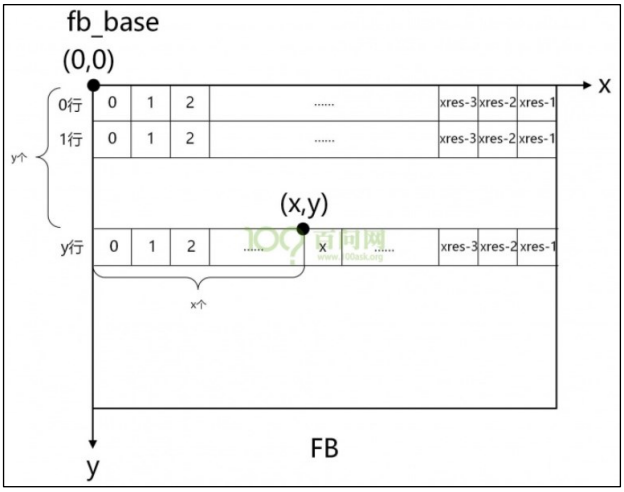


图 5.2 framebuffer 映射地址

可以用以下公式算出(x,y)坐标处像素对应的 Framebuffer 地址：

```
(x，y)像素起始地址=fb_base+(xres*bpp/8)*y + x*bpp/8
```

最后一个要解决的问题就是像素的颜色怎么表示？它是用 RGB 三原色(红、绿、蓝)来表示的，在不同的 BPP 格式中，用不同的位来分别表示 R、G、B，如图 5.3 所示：

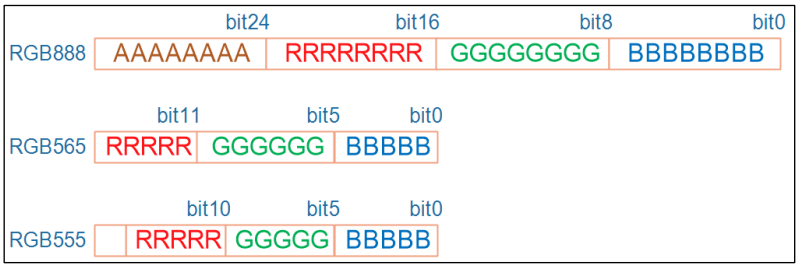


图 5.3 RGB 三种格式

对于 32BPP，一般只设置其中的低 24 位，高 8 位表示透明度，一般的 LCD 都不支持。

对于 24BPP，硬件上为了方便处理，在 Framebuffer 中也是用 32 位来表示，效果跟 32BPP 是一样的。

对于 16BPP，常用的是 RGB565；很少的场合会用到 RGB555，这可以通过 ioctl 读取驱动程序中的 RGB 位偏移来确定使用哪一种格式。

5.2 涉及的 API 函数

本节程序的目的是：打开 LCD 设备节点，获取分辨率等参数，映射 Framebuffer，最后实现描点函数。

5.2.1 open 函数

在 Ubuntu 中执行“man 2 open”，可以看到 open 函数的说明：

```
NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE
```

图 5.4 man 查看 open 函数

头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

函数原型：

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

函数说明：

① pathname 表示打开文件的路径；

② Flags 表示打开文件的方式，常用的有以下 6 种，

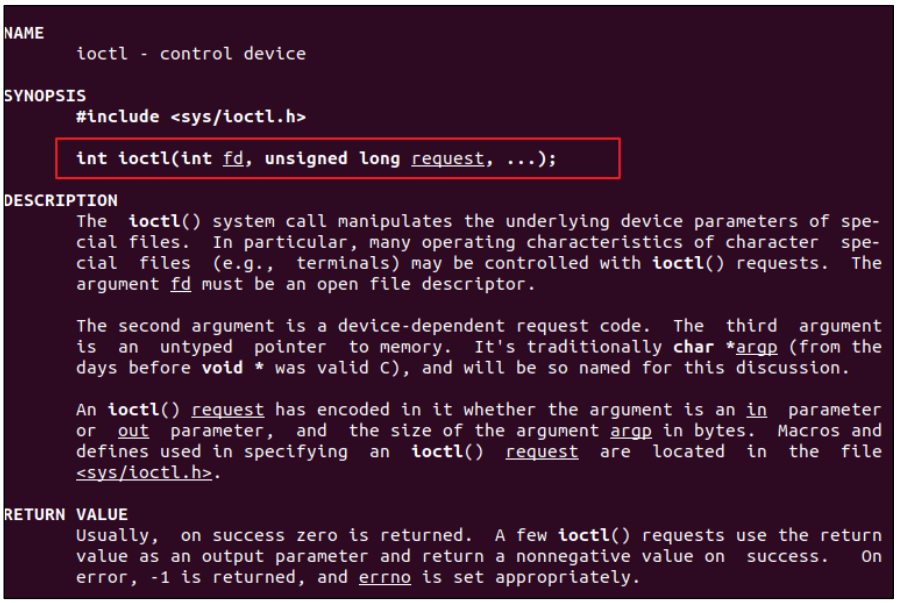
- O_RDWR 表示可读可写方式打开；
- O_RDONLY 表示只读方式打开；
- O_WRONLY 表示只写方式打开；
- O_APPEND 表示如果这个文件中本来是有内容的，则新写入的内容会接续到原来内容的后面；
- O_TRUNC 表示如果这个文件中本来是有内容的，则原来的内容会被丢弃，截断；
- O_CREAT 表示当前打开文件不存在，我们创建它并打开它，通常与 O_EXCL 结合使用，当没有文件时创建文件，有这个文件时会报错提醒我们；

③ Mode 表示创建文件的权限，只有在 flags 中使用了 O_CREAT 时才有效，否则忽略。

④ 返回值：打开成功返回文件描述符，失败将返回-1。

5.2.2 ioctl 函数

在 Ubuntu 中执行“man ioctl”，可以看到 ioctl 函数的说明：



```
NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int fd, unsigned long request, ...);

DESCRIPTION
       The  ioctl() system call manipulates the underlying device parameters of spe-
       cial files.  In particular, many operating characteristics of character  spe-
       cial  files  (e.g.,  terminals) may be controlled with ioctl() requests.  The
       argument fd must be an open file descriptor.

       The second argument is a device-dependent request code.  The  third  argument
       is  an  untyped  pointer  to memory.  It's traditionally char *argp (from the
       days before void * was valid C), and will be so named for this discussion.

       An ioctl() request has encoded in it whether the argument is an in  parameter
       or  out  parameter,  and  the size of the argument argp in bytes.  Macros and
       defines used in specifying  an  ioctl()  request  are  located  in  the  file
       <sys/ioctl.h>.

RETURN VALUE
       Usually,  on success zero is returned.  A few ioctl() requests use the return
       value as an output parameter and return a nonnegative value on  success.   On
       error, -1 is returned, and errno is set appropriately.
```

图 5.5 man 查看 ioctl 函数

头文件：

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

函数原型：

```
int ioctl(int fd, unsigned long request, ...);
```

函数说明：

① fd 表示文件描述符；

② request 表示与驱动程序交互的命令，用不同的命令控制驱动程序输出我们需要的数据；

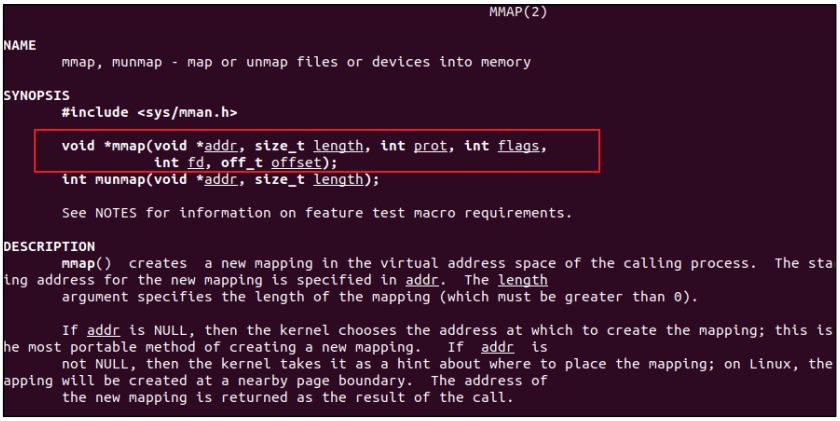
③ … 表示可变参数 arg，根据 request 命令，设备驱动程序返回输出的数据。

④ 返回值：打开成功返回文件描述符，失败将返回-1。

ioctl 的作用非常强大、灵活。不同的驱动程序内部会实现不同的 ioctl，APP 可以使用各种 ioctl 跟驱动程序交互：可以传数据给驱动程序，也可以从驱动程序中读出数据。

5.2.3 mmap 函数(待后续修正)

在 Ubuntu 中执行“man mmap”，可以看到 mmap 函数的说明：



```
                                                MMAP(2)

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>

       void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
       int munmap(void *addr, size_t length);

       See NOTES for information on feature test macro requirements.

DESCRIPTION
       mmap()  creates  a new mapping in the virtual address space of the calling process.  The sta
ing address for the new mapping is specified in addr.  The length
       argument specifies the length of the mapping (which must be greater than 0).

       If addr is NULL, then the kernel chooses the address at which to create the mapping; this is
he most portable method of creating a new mapping.   If  addr  is
       not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the
apping will be created at a nearby page boundary.  The address of
       the new mapping is returned as the result of the call.
```

图 5.6 man 查看 mmap 函数

想更深刻地理解 mmap 的内部机制，可以看《嵌入式 Linux 驱动开发基础知识》中关于 mmap 的介绍。作为 APP 开发，只需要知道它的用法就可以了。

头文件：

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

函数原型：

```
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

函数说明：

① addr 表示指定映射的内存起始地址，通常设为 NULL 表示让系统自动选定地址，并在成功映射后返回该地址；

② length 表示将文件中多大的内容映射到内存中；

③ prot 表示映射区域的保护方式，可以为以下 4 种方式的组合

- PROT_EXEC 映射区域可被执行
- PROT_READ 映射区域可被读出
- PROT_WRITE 映射区域可被写入

- PROT_NONE 映射区域不能存取

④ Flags 表示影响映射区域的不同特性，常用的有以下两种

- MAP_SHARED 表示对映射区域写入的数据会复制回文件内，原来的文件会改变。
- MAP_PRIVATE 表示对映射区域的操作会产生一个映射文件的复制，对此区域的任何修改都不会写回原来的文件内容中。

⑤ 返回值：若成功映射，将返回指向映射的区域的指针，失败将返回-1。

5.3 Framebuffer 程序分析

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\07_framebuffer\show_pixel.c
```

5.3.1 打开设备

首先打开设备节点：

```
73      fd_fb = open("/dev/fb0", O_RDWR);
74      if (fd_fb < 0)
75      {
76              printf("can't open /dev/fb0\n");
77              return -1;
78      }
```

5.3.2 获取 LCD 参数

LCD 驱动程序给 APP 提供 2 类参数：可变的参数 fb_var_screeninfo、固定的参数 fb_fix_screeninfo。编写应用程序时主要关心可变参数，它的结构体定义如下(#include <linux/fb.h>)：

```
struct fb_var_screeninfo {
    __u32 xres;             /* visible resolution       */
    __u32 yres;             分辨率
    __u32 xres_virtual;     /* virtual resolution       */
    __u32 yres_virtual;
    __u32 xoffset;          /* offset from virtual to visible */
    __u32 yoffset;          /* resolution               */

    __u32 bits_per_pixel;   bpp  /* guess what          */
    __u32 grayscale;        /* 0 = color, 1 = grayscale,    */
                            /* >1 = FOURCC              */
    struct fb_bitfield red;     /* bitfield in fb mem if true color, */
    struct fb_bitfield green;   /* else only length is significant */
    struct fb_bitfield blue;    RGB分别用多少位来表示,从哪位开始
    struct fb_bitfield transp;  /* transparency         */

    __u32 nonstd;           /* != 0 Non standard pixel format */

    __u32 activate;         /* see FB_ACTIVATE_*        */

    __u32 height;           /* height of picture in mm    */
    __u32 width;            /* width of picture in mm     */

    __u32 accel_flags;      /* (OBSOLETE) see fb_info.flags */

    /* Timing: All values in pixclocks, except pixclock (of course) */
    __u32 pixclock;         /* pixel clock in ps (pico seconds) */
    __u32 left_margin;      /* time from sync to picture    */
    __u32 right_margin;     /* time from picture to sync    */
    __u32 upper_margin;     /* time from sync to picture    */
    __u32 lower_margin;
    __u32 hsync_len;        /* length of horizontal sync    */
    __u32 vsync_len;        /* length of vertical sync  */
    __u32 sync;         /* see FB_SYNC_*        */
    __u32 vmode;            /* see FB_VMODE_*       */
    __u32 rotate;           /* angle we rotate counter clockwise */
    __u32 colorspace;       /* colorspace for FOURCC-based modes */
    __u32 reserved[4];      /* Reserved for future compatibility */
} « end fb_var_screeninfo » ;
```

图 5.7 fb_var_screeninfo

可以使用以下代码获取 fb_var_screeninfo：

```
12 static struct fb_var_screeninfo var; /* Current var */
……
79      if (ioctl(fd_fb, FBIOGET_VSCREENINFO, &var))
80      {
81              printf("can't get var\n");
82              return -1;
83      }
```

注意到 ioctl 里用的参数是：FBIOGET_VSCREENINFO，它表示 get var screen info，获得屏幕的可变信息；当然也可以使用 FBIOPUT_VSCREENINFO 来调整这些参数，但是很少用到。

对于固定的参数 fb_fix_screeninfo，在应用编程中很少用到。它的结构体定义如下：

```
struct fb_fix_screeninfo {
    char id[16];            /* identification string eg "TT Builtin" */
    unsigned long smem_start;   /* Start of frame buffer mem */
                    /* (physical address) */
    __u32 smem_len;         /* Length of frame buffer mem */
    __u32 type;          /* see FB_TYPE_*        */
    __u32 type_aux;         /* Interleave for interleaved Planes */
    __u32 visual;           /* see FB_VISUAL_*      */
    __u16 xpanstep;         /* zero if no hardware panning  */
    __u16 ypanstep;         /* zero if no hardware panning  */
    __u16 ywrapstep;        /* zero if no hardware ywrap    */
    __u32 line_length;      /* length of a line in bytes    */
    unsigned long mmio_start;   /* Start of Memory Mapped I/O   */
                    /* (physical address) */
    __u32 mmio_len;         /* Length of Memory Mapped I/O  */
    __u32 accel;            /* Indicate to driver which */
                    /*  specific chip/card we have  */
    __u16 capabilities;     /* see FB_CAP_*         */
    __u16 reserved[2];      /* Reserved for future compatibility */
} « end fb_fix_screeninfo » ;
```

Framebuffer的物理地址

图 5.8 fb_fix_screeninfo

可以使用 ioctl FBIOGET_FSCREENINFO 来读出这些信息，但是很少用到。

5.3.3 映射 Framebuffer

要映射一块内存，需要知道它的地址——这由驱动程序来设置，需要知道它的大小——这由应用程序决定。代码如下：

```
85      line_width  = var.xres * var.bits_per_pixel / 8;
86      pixel_width = var.bits_per_pixel / 8;
87      screen_size = var.xres * var.yres * var.bits_per_pixel / 8;
88      fb_base = (unsigned char *)mmap(NULL , screen_size, PROT_READ | PROT_WRITE, M
                  AP_SHARED, fd_fb, 0);
89      if (fb_base == (unsigned char *)-1)
90      {
91              printf("can't mmap\n");
92              return -1;
93      }
```

第 88 行中，screen_size 是整个 Framebuffer 的大小；PROT_READ | PROT_WRITE 表示该区域可读、可写；MAP_SHARED 表示该区域是共享的，APP 写入数据时，会直达驱动程序，这个参数的更深刻理解可以参考后面驱动基础中讲

到的 mmap 知识。

5.3.4 描点函数

能够在 LCD 上描绘指定像素后，就可以写字、画图，描点函数是基础。代码如下：

```
28 void lcd_put_pixel(int x, int y, unsigned int color)
29 {
30     unsigned char *pen_8 = fb_base+y*line_width+x*pixel_width;
31     unsigned short *pen_16;
32     unsigned int *pen_32;
33
34     unsigned int red, green, blue;
35
36     pen_16 = (unsigned short *)pen_8;
37     pen_32 = (unsigned int *)pen_8;
38
39     switch (var.bits_per_pixel)
40     {
41             case 8:
42             {
43                     *pen_8 = color;
44                     break;
45             }
46             case 16:
47             {
48                     /* 565 */
49                     red   = (color >> 16) & 0xff;
50                     green = (color >> 8) & 0xff;
51                     blue  = (color >> 0) & 0xff;
52                     color = ((red >> 3) << 11) | ((green >> 2) << 5) | (blue >> 3);
53                     *pen_16 = color;
54                     break;
55             }
56             case 32:
57             {
58                     *pen_32 = color;
59                     break;
60             }
61             default:
62             {
63                     printf("can't surport %dbpp\n",var.bits_per_pixel);
64                     break;
65             }
66     }
67 }
```

第 28 行中传入的 color 表示颜色，它的格式永远是 0x00RRGGBB，即 RGB888。当 LCD 是 16bpp 时，要把 color 变量中的 R、G、B 抽出来再合并成 RGB565 格式。

第 30 行计算(x,y)坐标上像素对应的 Framebuffer 地址。

第 43 行，对于 8bpp，color 就不再表示 RBG 三原色了，这涉及调色板的概念，color 是调色板的值。

第 49～51 行，先从 color 变量中把 R、G、B 抽出来。

第 52 行，把 red、green、blue 这三种 8 位颜色值，根据 RGB565 的格式，只保留 red 中的高 5 位、green 中的高 6 位、blue 中的高 5 位，组合成一个新的 16 位颜色值。

第 53 行，把新的 16 位颜色值写入 Framebuffer。

第 58 行，对于 32bpp，颜色格式跟 color 参数一致，可以直接写入 Framebuffer。

5.3.5 随便画几个点

本程序的 main 函数，在最后只是简单地画了几个点：

```
95      /* 清屏：全部设为白色 */
96      memset(fbmem, 0xff, screen_size);
97
98      /* 随便设置出 100 个为红色 */
99      for (i = 0; i < 100; i++)
100             lcd_put_pixel(var.xres/2+i, var.yres/2, 0xFF0000);
```

5.4 上机实验

在 Ubuntu 中编译程序，先设置交叉编译工具链，再执行以下命令：

```
arm-buildroot-linux-gnueabihf-gcc -o show_pixel show_pixel.c
```

然后在开发板上执行 show_pixel 程序。

注意：板子的出厂程序中一般都有 GUI，所以可能需要把 GUI 程序禁止掉。具体方法请看本文档，以后会补充禁止 GUI 的方法。你可以先不禁止 GUI，直接执行 show_pixel 看看 LCD 有无现象。

第6章 文字显示

6.1 字符的编码方式

6.1.1 编码与字体

在计算机上，我们看到的字符“A”可能长这样：

A

也可能长这样：

$\mathcal{A}$

对于同一个 TXT 文件中的内容，你在 Notepad 上选择不同字体时，字符显示的形状不一样。

所以 TXT 文件中保存的是字符的核心：它的编码值。而 Notepad 上显示时，这些字符对应什么样的形状态，这是由字符文件决定的。编码值，字体是两个不一样的东西，比如 A 的编码值是 0x41，但是在屏幕上显示出来时可以使用不同的形状。

什么叫编码？就是一个字符用什么数字来表示。在计算机里一切都是用数字来表示，比如字符 A，用 0x01 还是 0x02 来表示它？我们使用 0x41 来表示它。当你去打开一个 TXT 文件时，发现里面含有数值 0x41，你就知道了：哦，这里有一个字符 A。

一个字符用哪个数字来表示？有很多标准，举例讲解。

1 ASCII

是“American Standard Code for Information Interchange”的缩写，美国信息交换标准代码。

电脑毕竟是西方人发明的，他们常用字母就 26 个，区分大小写、加上标点符号也没超过 127 个，每个字符用一个字节来表示就足够了。一个字节的 7 位就可以表示 128 个数值，在 ASCII 码中最高位永远是 0。

字符和数值的对应关系可以参考：

https://baike.baidu.com/item/ASCII

下面摘录部分给大家一个印象：

Bin (二进制)	Oct (八进制)	Dec (十进制)	Hex (十六进制)	缩写/字符	解释
0100 0000	0100	64	0x40	@	电子邮件符号
0100 0001	0101	65	0x41	A	大写字母A
0100 0010	0102	66	0x42	B	大写字母B
0100 0011	0103	67	0x43	C	大写字母C
0100 0100	0104	68	0x44	D	大写字母D
0100 0101	0105	69	0x45	E	大写字母E
0100 0110	0106	70	0x46	F	大写字母F
0100 0111	0107	71	0x47	G	大写字母G
0100 1000	0110	72	0x48	H	大写字母H
0100 1001	0111	73	0x49	I	大写字母I

图 6.1 ASCII 表

2 ANSI

强烈建议阅读：https://www.cnblogs.com/malecrab/p/5300486.html

使用记事本保存文件时，可以选择“ANSI”编码，却没有“ASCII”，如图 6.2 所示，怎么回事？

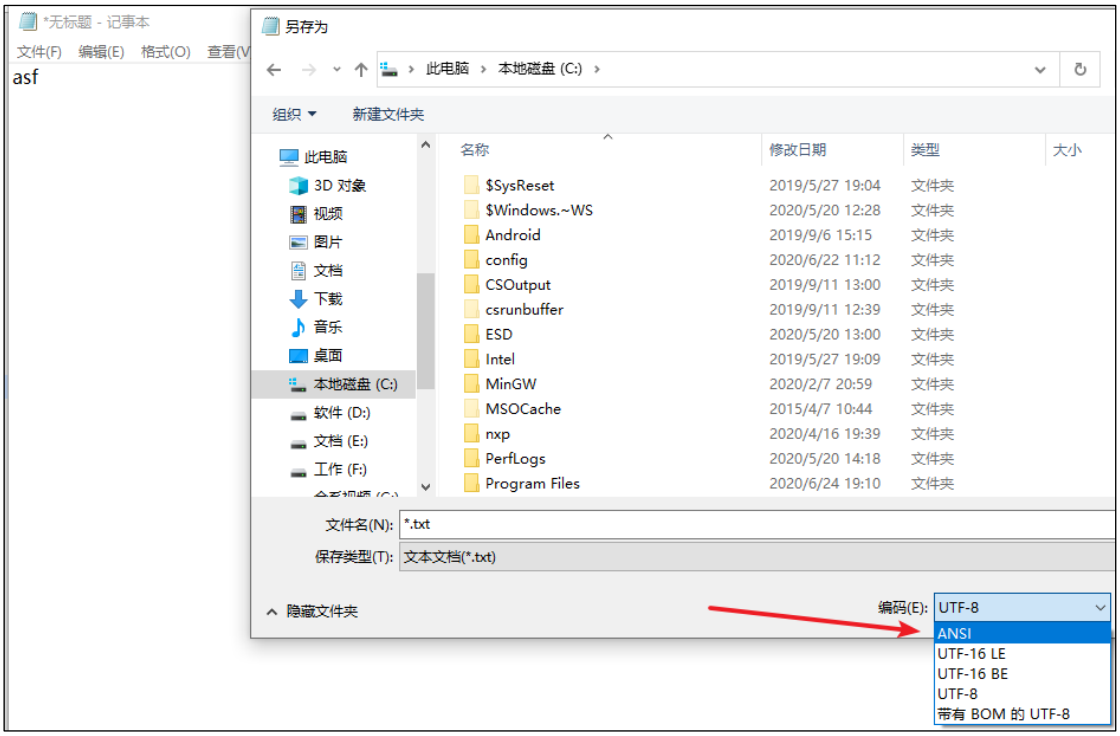


图 6.2 不能保存为 ASCII

ASNI 是 ASCII 的扩展，向下包含 ASCII。对于 ASCII 字符仍以一个字节来表示，对于非 ASCII 字符则使用 2 字节来表示。并没有固定的 ASNI 编码，它跟“本地化”(locale)密切相关。比如在中国大陆地区，ANSI 的默认编码是 GB2312；在港澳台地区默认编码是 BIG5。以数值“0xd0d6”为例，对于 GB2312 编码它表示“中”；对于 BIG5 编码它表示“筐”。所以对于 ANSI 编码的 TXT 文件，如果你打开它发现乱码，那么还得再次细分它的具体编码。

比如对于一个 TXT 文件，里面的数值如下：

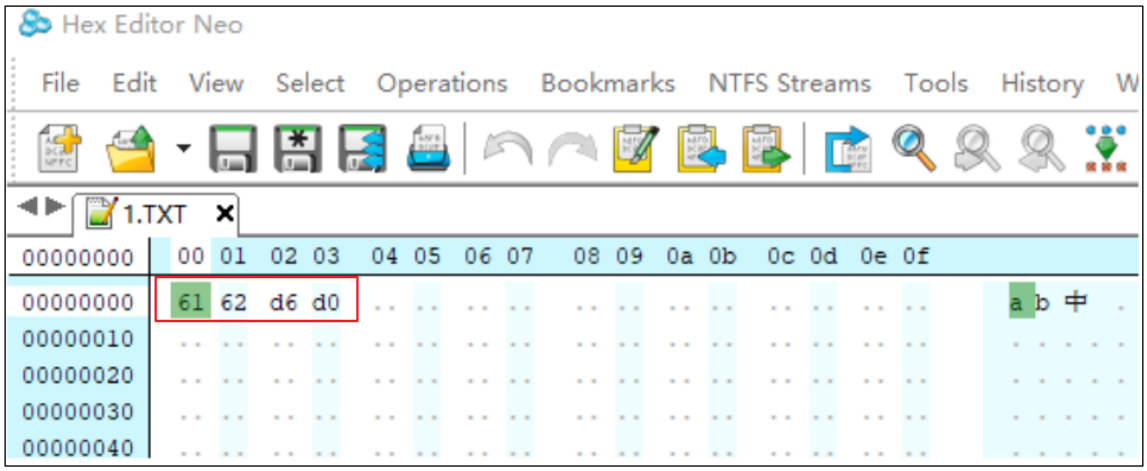


图 6.3 txt 编码

使用 Notepad 打开后，选择不同的编码(或称为字符集)，有不一样的显示，如下：

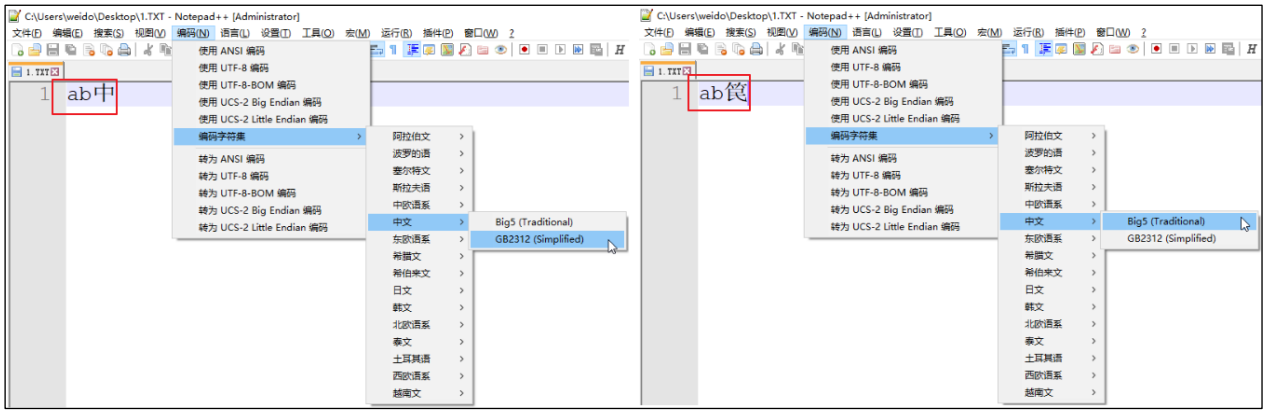


图 6.4 选择不同编码打开 txt

这仅仅是在中国地区就出现这些不兼容的问题。对于不同国家，它们默认的 ANSI 编码各不相同，所以同一个 TXT 文件在不同国家就很有可能出现乱码。

根本的原理在于没有“统一的编码”，那解决方法自然就是使用“统一的编码”：UNICODE。

3 UNICODE

在 ANSI 标准中，很多种文字都有自己的编码标准，汉字简体字有 GB2312、繁体字有 BIG5，这难免同一个数值对应不同字符。比如数值“0xd0d6”，对于 GB2312 编码它表示“中”；对于 BIG5 编码它表示“筧”。这造成了使用 ANSI 编码保存的文件，不适合跨地区交流。

UNICODE 编码就是解决这类问题：对于地球上任意一个字符，都给它一个唯一的数值。

UNICODE 仍然向下兼容 ASCII，但是对于其他字符会有对应的数值，比如对于“中”、“筧”，它们的数值分别是：0x4e2d、0x7b22

UNICODE 中的数值范围是 0x0000 至 0x10FFFF，有 1,114,111 即 100 多万个数值，可以表示 100 多万个字符，足够地球人使用了。

6.1.2 UNICODE 编码实现

所谓编码实现，就是对于一个数值，怎么表示它。这很奇怪，数值还能怎么表示？比如“中”的 UNICODE 值是 0x4e2d，在 TXT 文件中怎么表示 0x4e2d？直接写入 0x4e2d？不行！

比如在 TXT 文件中写入 2 字节数据“0x2d 0x4e”，它可以用来表示“中”字吗？不能！它们对应 ASCII 字符“-N”。

问题的关键在于：怎么断字。在 TXT 文件中，2 字节数据“0x2d 0x4e”是作为一个整体看待，还是拆成 2 部分看待？

所以，需要用一定的技巧来表示数值，这就对应不同的编码实现。

现在我们知道：

① ASCII 编码中使用一个字节来表示一个字符，只用到其中的 7 位，最高位恒为 0；
② ANSI 编码中，对于 ASCII 字符仍使用一个字节来表示(BIT7 是 0)，对于非 ASCII 字符一般使用 2 个字节来表示，非 ASCII 字符的数值 BIT7 都是 1。
③ UNICODE：这就有点复杂了，下面一一讲解。

先用记事本新建 3 个文件：utf-16_le.txt、utf-16_be.txt、utf-8.txt、bom_utf-8.txt，里面的内容都是“ab 中”，保存时编码分别选择“UTF-16 LE”、“UTF-16 BE”、“UTF-8”、“带有 BOM 的 UTF-8”，图 6.5 是其中一个例子：

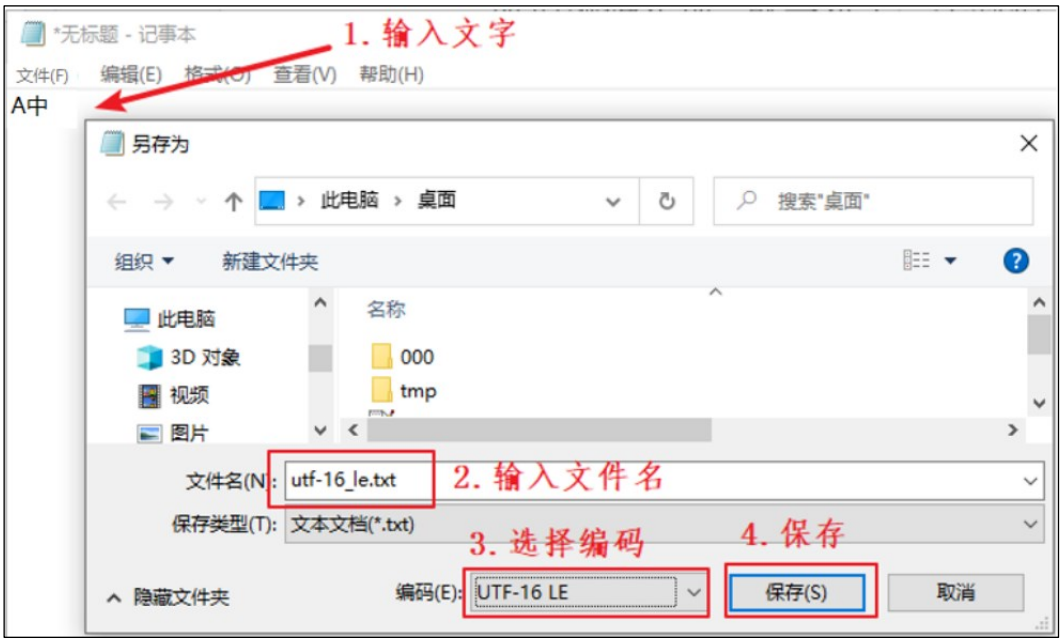


图 6.5 选择编码保存文件

怎么表示一个 UNICODE 数值？

1 使用 3 个字节表示一个 UNICODE

不，太浪费。

UNICODE 的最大值是 0x10FFFF，那使用 3 个字节来表示一个 UNICODE 数值？这当然是很省事的方法，但是会造成浪费，比如字符 A 的 UNICOCDE 值是 0x41，难道也用“0x41 0x00 0x00”这 3 个字节来表示？

2 UCS-2 Little endian/UTF-16 LE

每个 UNICODE 值用 3 字节来表示有点浪费，那只用 2 字节呢？它可以表示 2^16=65536 个字符，全世界常用的字符都可以表示了。

Little endian 表示小字节序，数值中权重低的字节放在前面，比如字符“A 中”在 TXT 文件中的数值如下，其中的“A”使用“0x41 0x00”两字节表示；“中”使用“0x2d 0x4e”两字节表示。文件开头的“0xff 0xfe”表示“UTF-16 LE”。

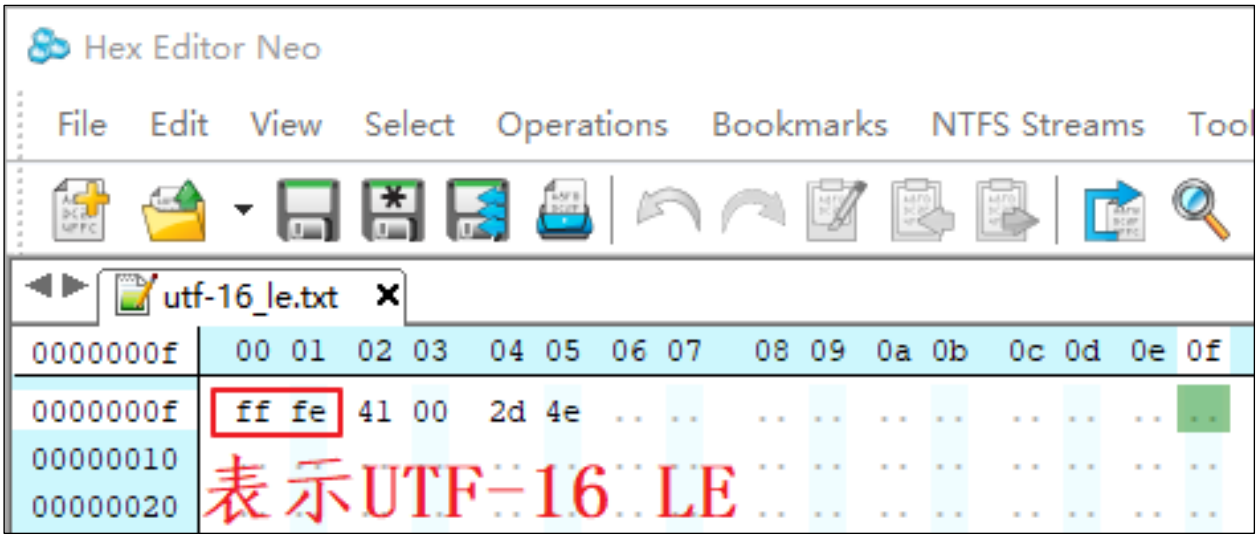


图 6.6 UTF-16 LE

3 UCS-2 Big endian/UTF-16 BE

Big endian 表示大字节序，数值中权重低的字节放在后面，比如字符“ab中”在 TXT 文件中的数值如下，其中的“A”使用“0x00 0x41”两字节表示；“中”使用“0x4e 0x2d”两字节表示。文件开头的“0xfe 0xff”表示“UTF-16 BE”。

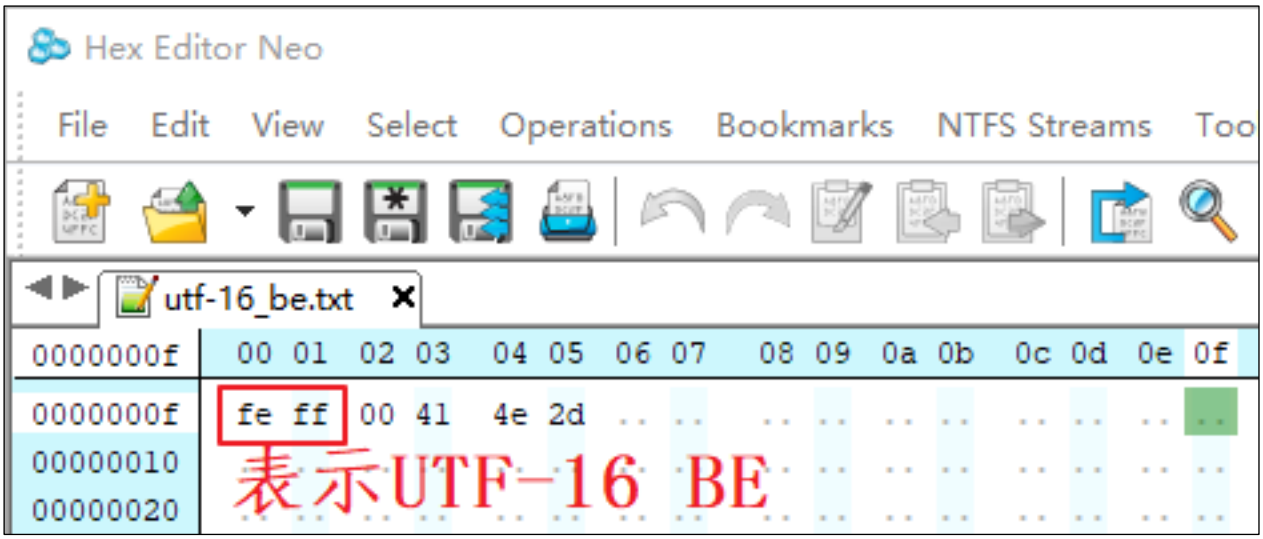


图 6.7 UTF-16 BE

4 UTF8

在上面 2 种方法中，每一个 UNICODE 使用 2 字节来表示，这有 3 个缺点：表示的字符数量有限、对于 ASCII 字符有空间浪费、如果文件中有某个字节丢失，这会使得后面所有字符都因为错位而无法显示。

使用 UTF8 可以解决上述所有问题。UTF8 是变长的编码方法，有 2 种 UTF8 格式的文件：带有头部、不带头部。先举例，看图 6.8：

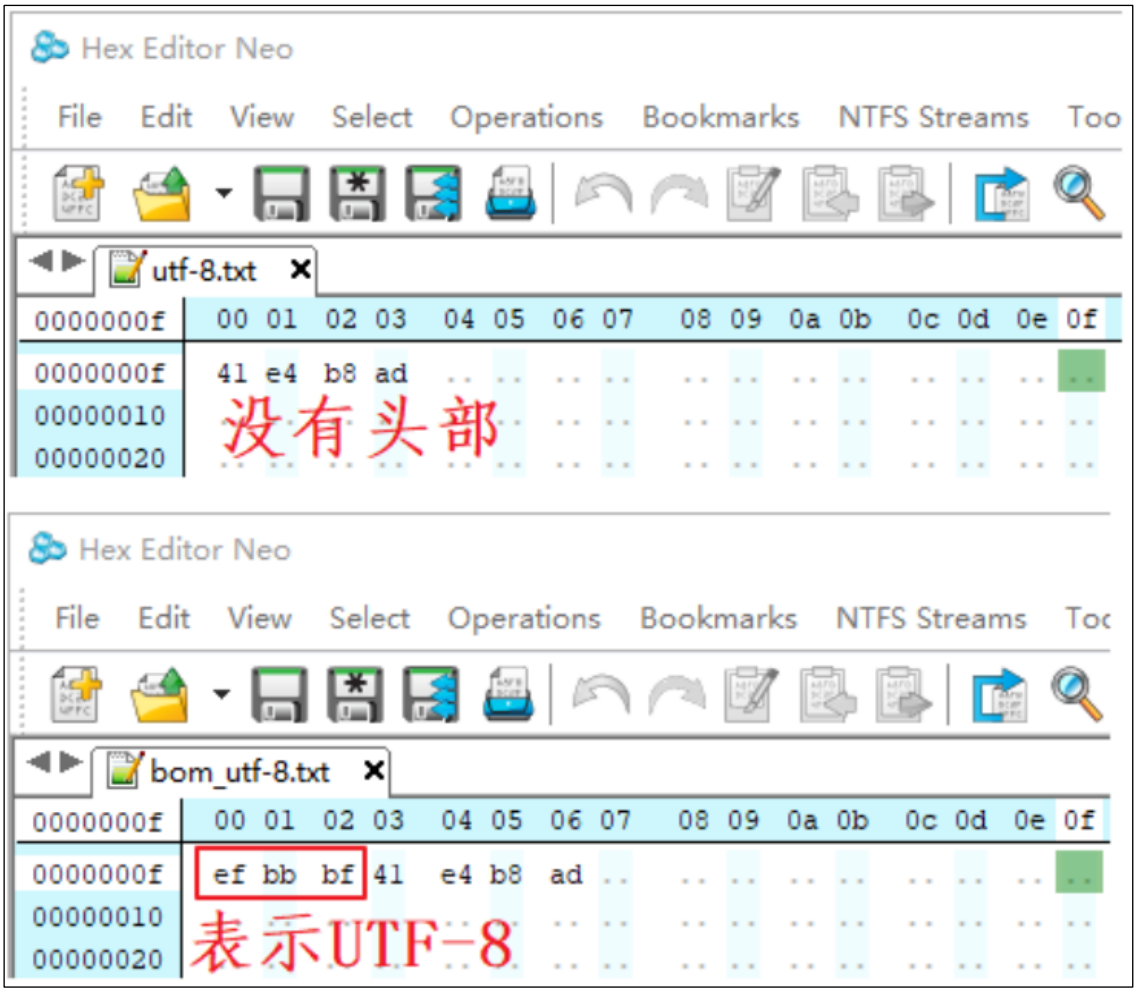


图 6.8 UTF-8

对于其中的 ASCII 字符，在 UTF8 文件中直接用其 ASCII 码来表示，比如上图中的 0x61 表示字符 a、0x62 表示字符 b。上图中的 3 个字节“0xe4 0xb8 0xad”表示的数值是 0x4e2d，对应“中”的 UNICODE 码。

对于非 ASCII 字符，使用变长的编码：每一个字节的高位都自带长度信息。请看图 6.9：

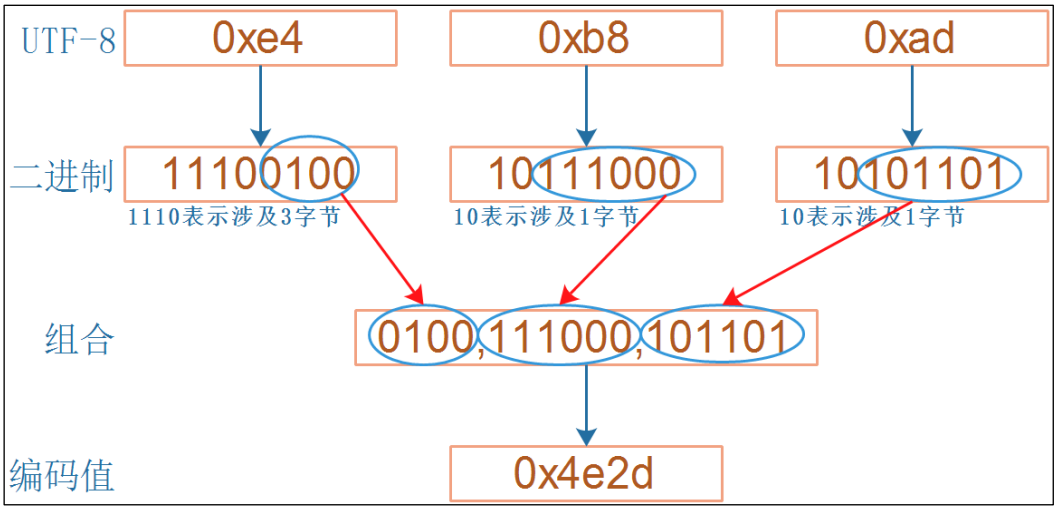


图 6.9 变长编码

上图中，0xe4 的二进制是“11100100”，高位有 3 个 1，表示从当前字节起有 3 字节参与表示 UNICODE；

0xb8 的二进制是“10111000”，高位有 1 个 1，表示从当前字节起有 1 字节参与表示 UNICODE；

0xad 的二进制是“10101101”，高位有 1 个 1，表示从当前字节起有 1 字节参与表示 UNICODE；

除去高位的“1110”、“10”、“10”后，剩下的二进制数组合起来得到“01001110001101”，它就是 0x4e2d，即“中”的 UNICODE 值。

使用 UTF8 编码时，即使 TXT 文件中丢失了某些数据，也只会影响到当前字符的显示，后面的字符不受影响。

6.2 ASCII 字符的点阵显示

要在 LCD 中显示一个 ASCII 字符，即英文字母这些字符，首先是要找到字符对应的点阵。在 Linux 内核源码中有这个文件：lib\fonts\font_8x16.c，里面以数组形式保存各个字符的点阵，比如：

```
00013: static const unsigned char fontdata_8x16[FONTDATAMAX] = {
00014:
00015:     /* 0 0x00 '^@' */
00016:     0x00, /* 00000000 */
00017:     0x00, /* 00000000 */
00018:     0x00, /* 00000000 */
00019:     0x00, /* 00000000 */
00020:     0x00, /* 00000000 */
00021:     0x00, /* 00000000 */
00022:     0x00, /* 00000000 */
00023:     0x00, /* 00000000 */
00024:     0x00, /* 00000000 */
00025:     0x00, /* 00000000 */
00026:     0x00, /* 00000000 */
00027:     0x00, /* 00000000 */
00028:     0x00, /* 00000000 */
00029:     0x00, /* 00000000 */
00030:     0x00, /* 00000000 */
00031:     0x00, /* 00000000 */
00032:
00033:     /* 1 0x01 '^A' */
00034:     0x00, /* 00000000 */
00035:     0x00, /* 00000000 */
00036:     0x7e, /* 01111110 */
00037:     0x81, /* 10000001 */
00038:     0xa5, /* 10100101 */
00039:     0x81, /* 10000001 */
00040:     0x81, /* 10000001 */
00041:     0xbd, /* 10111101 */
00042:     0x99, /* 10011001 */
00043:     0x81, /* 10000001 */
00044:     0x81, /* 10000001 */
00045:     0x7e, /* 01111110 */
00046:     0x00, /* 00000000 */
00047:     0x00, /* 00000000 */
00048:     0x00, /* 00000000 */
00049:     0x00, /* 00000000 */
```

ASCII值0的字符的点阵，占据16字节

ASCII值1的字符的点阵，占据16字节

图 6.10 LCD 的 ASCII 点阵

数组里的数字是如何表示点阵的？以字符 A 为例，如图 6.11 所示：

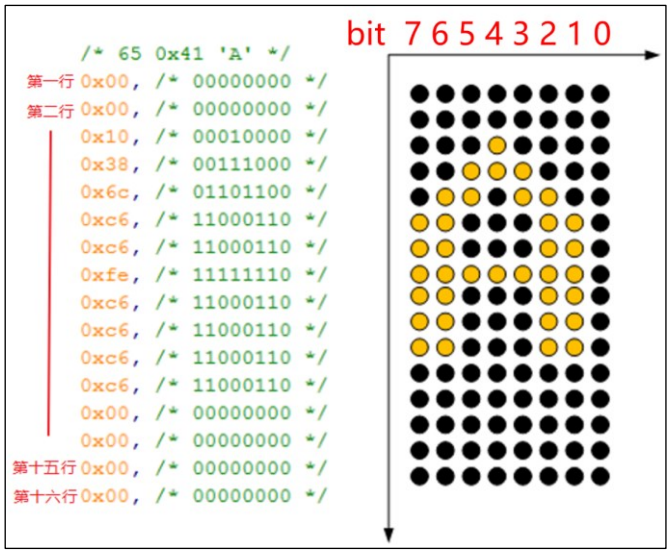


图 6.11 字符 A 的点阵

上图左侧有 16 行数值，每行 1 个字节。每一个节对应右侧一行中 8 个像素：像素从右边数起，bit0 对应第 0 个像素，bit1 对应第 1 个像素，……，bit7 对应第 7 个像素。某位的值为 1 时，表示对应的像素要被点亮；值为 0 时表示对应的像素要熄灭。

所以要显示某个字符时，根据它的 ASCII 码在 fontdata_8x16 数组中找到它的点阵，然后取出这 16 个字节去描画 16 行像素。

比如字符 A 的 ASCII 值是 0x41，那么从 fontdata_8x16[0x41*16]开始取其点阵数据。

使用 GIT 下载所有源码后，本节源码位于如下目录：

01_all_series_quickstart
04_嵌入式 Linux 应用开发基础知识\source\08_show_ascii\show_ascii.c

核心函数是 void lcd_put_ascii(int x, int y, unsigned char c)，它在 LCD 的(x,y)位置处显示字符 c，代码如下图所示：

```
04691: void lcd_put_ascii(int x, int y, unsigned char c)
04692: {
04693:     unsigned char *dots = (unsigned char *)&fontdata_8x16[c*16];
04694:     int i, b;
04695:     unsigned char byte;
04696:
04697:     for (i = 0; i < 16; i++)
04698:     {
04699:         byte = dots[i];
04700:         for (b = 7; b >= 0; b--)
04701:         {
04702:             if (byte & (1<<b))
04703:             {
04704:                 /* show */
04705:                 lcd_put_pixel(x+7-b, y+i, 0xffffff); /* 白 */
04706:             }
04707:             else
04708:             {
04709:                 /* hide */
04710:                 lcd_put_pixel(x+7-b, y+i, 0); /* 黑 */
04711:             }
04712:         }
04713:     }
04714: } ? end lcd_put_ascii ?
04715:
```

图 6.12 显示字符函数

6.2.1 获取点阵

对于字符 c，char c，它的点阵获取方法如下：

```
4693        unsigned char *dots = (unsigned char *)&fontdata_8x16[c*16];
```

6.2.2 描点

根据“图 6.11 字符 A 的点阵”，我们分析下如何利用点阵在 LCD 上显示一个英文字母。

因为有十六行，所以首先要有一个循环 16 次的大循环，然后每一行里有 8 位，那么在每一个大循环里也需要一个循环 8 次的小循环。小循环里的判断单行的描点情况，如果是 1，就填充白色，如果是 0 就填充黑色，如此一来，就可以显示出黑色底，白色轮廓的英文字母。

```
4697        for (i = 0; i < 16; i++)
4698        {
4699            byte = dots[i];
4700            for (b = 7; b >= 0; b--)
4701            {
4702                if (byte & (1<<b))
4703                {
4704                    /* show */
4705                    lcd_put_pixel(x+7-b, y+i, 0xffffff); /* 白 */
4706                }
4707                else
4708                {
4709                    /* hide */
4710                    lcd_put_pixel(x+7-b, y+i, 0); /* 黑 */
4711                }
4712            }
4713 }
```

6.2.3 main 函数

main 函数中首先要打开 LCD 设备，获取 Framebuffer 参数，实现 lcd_put_pixel 函数；然后调用 lcd_put_ascii 即可绘制字符。

代码如下：

```
4716 int main(int argc, char **argv)
4717 {
4718    fd_fb = open("/dev/fb0", O_RDWR);
4719    if (fd_fb < 0)
4720    {
4721            printf("can't open /dev/fb0\n");
4722            return -1;
4723    }
4724    if (ioctl(fd_fb, FBIOGET_VSCREENINFO, &var))
4725    {
4726            printf("can't get var\n");
4727            return -1;
4728    }
4729
```

```
4730    line_width  = var.xres * var.bits_per_pixel / 8;
4731    pixel_width = var.bits_per_pixel / 8;
4732    screen_size = var.xres * var.yres * var.bits_per_pixel / 8;
4733    fbmem = (unsigned char *)mmap(NULL , screen_size, PROT_READ | PROT_WRITE, MAP
                _SHARED, fd_fb, 0);
4734    if (fbmem == (unsigned char *)-1)
4735    {
4736            printf("can't mmap\n");
4737            return -1;
4738    }
4739
4740    /* 清屏: 全部设为黑色 */
4741    memset(fbmem, 0, screen_size);
4742
4743    lcd_put_ascii(var.xres/2, var.yres/2, 'A'); /*在屏幕中间显示 8*16 的字母 A*/
4744
4745    munmap(fbmem , screen_size);
4746    close(fd_fb);
4747
4748    return 0;
4749 }
4750
```

6.2.4 编译 c 文件 show_ascii.c

编译命令：`arm-buildroot-linux-gnueabihf-gcc -o show_ascii show_ascii.c`

注意：不同的板子，编译工具的前缀可能不一样。

6.2.5 机实验

把 show_ascii 程序放到板子上，执行命令：./show_ascii。如果实验成功，我们将看到屏幕中间会显示出一个白色的字母‘A’。

6.2.6 课后作业

修改 lcd_put_ascii 函数，可以指定字符颜色。实现 lcd_put_str 函数，输出字符串，可以换行。

在 show_ascii.c 的基础上实现汉字的显示：要找到汉字字库、了解像素排列顺序、得到汉字编码。

6.3 中文字符的点阵显示

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\09_show_chinese\
      test_charset_ansi.c
      test_charset_utf8.c
      show_chinese.c
```

6.3.1 指定编码格式

使用点阵字库时，中文字符的显示原理跟 ASCII 字符是一样的。要注意的地方在于中文的编码：在 C 源文件中它的编码方式是 GB2312 还是 UTF-8？编译出的可执行程序，其中的汉字编码方式是 GB2312 还是 UTF-8？

注意：一般不会使用 UTF-16 的编码方式，在这种方式下 ASCII 字符也是用 2 字节来表示，而其中一个字节是 0，但是在 C 语言中 0 表示字符串的结束符，会引起误会。

我们编写 C 程序时，可以使用 ANSI 编码，或是 UTF-8 编码；在编译程序时，可以使用以下的选项告诉编译器：

```
-finput-charset=GB2312
-finput-charset=UTF-8
```

如果不指定“-finput-charset”，GCC 就会默认 C 程序的编码方式为 UTF-8，即使你是以 ANSI 格式保存，也会被当作 UTF-8 来对待。

对于编译出来的可执行程序，可以指定它里面的字符是以什么方式编码，可以使用以下的选项编译器：

```
-fexec-charset=GB2312
-fexec-charset=UTF-8
```

如果不指定“-fexec-charset”，GCC 就会默认编译出的可执行程序中字符的编码方式为 UTF-8。

如果“-finput-charset”与“-fexec-charset”不一样，编译器会进行格式转换。

6.3.2 编码格式实验

下面做实验。

test_charset_ansi.c、test_charset_utf8.c 的编码格式分别为 ANSI、UTF-8，它们的程序代码是一样的，如下：

```
01 #include <stdio.h>
02 #include <string.h>
03
04 int main(int argc, char **argv)
05 {
06      char *str = "A 中";
07      int i;
08
09      printf("str's len = %d\n", (int)strlen(str));
10      printf("Hex code: ");
11      for (i = 0; i < strlen(str); i++)
12      {
13              printf("%02x ", (unsigned char)str[i]);
14      }
15      printf("\n");
16      return 0;
17 }
```

1 默认编码

实验如下：

```
book@100ask:~/09_show_chinese$ gcc -o test_charset_ansi test_charset_ansi.c
book@100ask:~/09_show_chinese$ ./test_charset_ansi
str's len = 3
Hex code: 41 d6 d0
book@100ask:~/09_show_chinese$ gcc -o test_charset_utf8 test_charset_utf8.c
book@100ask:~/09_show_chinese$ ./test_charset_utf8
str's len = 4
Hex code: 41 e4 b8 ad
```

不指定“-finput-charset”与“-fexec-charset”时，input-charset 和 exec-charset 默认都是 UTF-8，不会进行编码转换。即使 C 文件是 ANSI，也会被认为是 UTF-8，所以不会导致编码转换。

2 GB2312 转为 UTF-8

实验如下：

```
book@100ask:~/09_show_chinese$ gcc -finput-charset=GB2312 -fexec-charset=UTF-8 -o te
st_charset_ansi test_charset_ansi.c
book@100ask:~/09_show_chinese$ ./test_charset_ansi str's len = 4 Hex code: 41 e4 b8 a
d
book@100ask:~/09_show_chinese$ gcc -finput-charset=GB2312 -fexec-charset=UTF-8 -o te
st_charset_utf8 test_charset_utf8.c cc1: error: failure to convert GB2312 to UTF-8
```

从上面的输出信息可以看出来，GB2312 的“0xd6 0xd0”可以转换为 UTF-8 的“0xe4 0xb8 0xad”。而如果把原本就是 UTF-8 格式的 test_charset_utf8.c 当作 GB2312 格式，会引起错误。

3 UTF-8 转为 GB2312

实验如下：

```
book@100ask:~/09_show_chinese$ gcc -finput-charset=UTF-8 -fexec-charset=GB2312 -o te
st_charset_ansi test_charset_ansi.c
test_charset_ansi.c: In function ‘main’:
test_charset_ansi.c:6:14: error: converting to execution character set: Invalid or in
complete multibyte or wide character
  char *str = "A▒▒";
              ^~~~
book@100ask:~/09_show_chinese$ gcc -finput-charset=UTF-8 -fexec-charset=GB2312 -o te
st_charset_utf8 test_charset_utf8.c
book@100ask:~/09_show_chinese$ ./test_charset_utf8
str's len = 3
Hex code: 41 d6 d0
```

从上面的输出信息可以看出来，如果把原本就是 GB2312 格式的 test_charset_ansi.c 当作 UTF-8 格式，会引起错误。而 UTF-8 格式的“中”编码值为“0xe4 0xb8 0xad”，可以转换为 GB2312 的“0xd6 0xd0”。

在代码中使用汉字这类非 ASCII 码时，要特别留意编码格式。

6.3.3 汉字区位码

我们从网上搜到 HZK16 这个文件，它是常用汉字的 16*16 点阵字库。HZK16

里每个汉字使用 32 字节来描述，如图 6.13 所示：

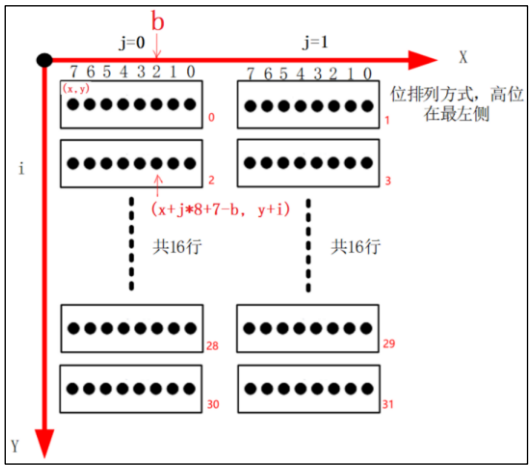


图 6.13 汉字区位码

跟 ASCII 字库一样，每个字节中每一位用来表示一个像素，位值等于 1 时表示对应像素被点亮，位值等于 0 时表示对应像素被熄灭。

HZK16 中是以 GB2312 编码值来查找点阵的，以“中”字为例，它的编码值是“0xd6 0xd0”，其中的 0xd6 表示“区码”，表示在哪一个区：第“0xd6 - 0xa1”区；其中的 0xd0 表示“位码”，表示它是这个区里的哪一个字符：第“0xd0 - 0xa1”个。每一个区有 94 个汉字。区位码从 0xa1 而不是从 0 开始，是为了兼容 ASCII 码。

所以，我们要显示的“中”字，它的 GB2312 编码是 d6d0，它是 HZK16 里第“(0xd6-0xa1)*94+(0xd0-0xa1)”个字符。

6.3.4 汉字点阵显示实验

1 打开汉字库文件

```
4787    fd_hzk16 = open("HZK16", O_RDONLY);
4788    if (fd_hzk16 < 0)
4789    {
4790            printf("can't open HZK16\n");
4791            return -1;
4792    }
4793    if(fstat(fd_hzk16, &hzk_stat))
4794    {
4795            printf("can't get fstat\n");
4796            return -1;
4797    }
4798    hzkmem = (unsigned char *)mmap(NULL , hzk_stat.st_size, PROT_READ, MAP_SHARE
D, fd_hzk16, 0);
4799    if (hzkmem == (unsigned char *)-1)
4800    {
4801            printf("can't mmap for hzk16\n");
4802            return -1;
4803    }
```

第 4787 行打开当前目录的字库文件：HZK16。

第 4793 行获得文件的状态信息，里面含有文件长度，这在后面的 mmap 中

用到。

第 4798 行使用 mmap 映射文件，以后就可以像访问内存一样读取文件内容；mmap 的返回结果保存在 hzkmem 中，它将作为字库的基地址。

2 编写显示汉字的函数

核心函数是 void lcd_put_chinese(int x, int y, unsigned char *str)，它在 LCD 的(x,y)位置处显示汉字字符 str，str[0]中保存区码、str[1]中保存位码。

代码如下图所示：

```
4732: void lcd_put_chinese(int x, int y, unsigned char *str)
4733: {
4734:     unsigned int area  = str[0] - 0xA1;
4735:     unsigned int where = str[1] - 0xA1;
4736:     unsigned char *dots = hzkmem + (area * 94 + where)*32;
4737:     unsigned char byte;
4738:
4739:     int i, j, b;
4740:     for (i = 0; i < 16; i++)
4741:         for (j = 0; j < 2; j++)
4742:         {
4743:             byte = dots[i*2 + j];
4744:             for (b = 7; b >=0; b--)
4745:             {
4746:                 if (byte & (1<<b))
4747:                 {
4748:                     /* show */
4749:                     lcd_put_pixel(x+j*8+7-b, y+i, 0xffffff); /* 白 */
4750:                 }
4751:                 else
4752:                 {
4753:                     /* hide */
4754:                     lcd_put_pixel(x+j*8+7-b, y+i, 0); /* 黑 */
4755:                 }
4756:             }
4757:         }
4758: } « end lcd_put_chinese »
4759:
```

图 6.14 显示汉字函数

代码分解如下：

第 4734 行确定该汉字属于哪个区；第 4735 行确实它是该区中哪一个汉字。

第 4736 行确实它的字库地址：每个区中有 94 个汉字，每个汉字在字库中占据 32 字节。

需要根据图 6.15 来理解第 4740 行开始的循环：

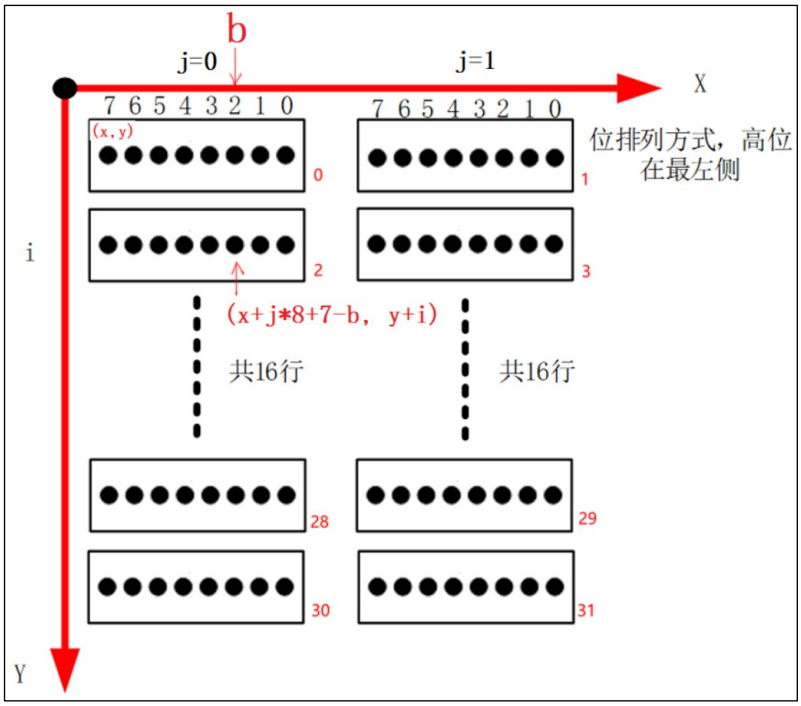


图 6.15 汉字点阵排布

图 6.15 是汉字点阵排布的示意图，总共有十六行，因此需要一个循环 16 次的大循环(第 4740 行)。

考虑到一行有两个字节，在大循环中加入一个 2 次的循环用于区分是哪个字节(第 4741 行)。

最后使用第 3 个循环来处理一个字节中的 8 位(第 4744 行)。对于每一位，它等于 1 时对应的像素被设置为白色，它等于 0 时对应的像素被设置为黑色。需要注意的是根据 x、y、i、j、b 来计算像素坐标。

3 使用 lcd_put_chinese 函数

程序文件：show_font.c

```
4762    unsigned char str[] = "中";
......
4810    printf("chinese code: %02x %02x\n", str[0], str[1]);
4811    lcd_put_chinese(var.xres/2 + 8,  var.yres/2, str);
```

4 编译程序

编译命令：

```
arm-buildroot-linux-gnueabihf-gcc -o  show_chinese  show_chinese.c
```

注意：不同的板子，编译工具的前缀可能不一样。

注意：使用上述命令时 show_chinese.c 的编码格式必须是 ANSI(GB2312)，否则编译时需要指定“-fexec-charset=GB2312”。

5 上机实验

把 show_chinese 程序放到板子上，执行命令：./show_chinese。如果实验成功，我们将看到屏幕中间会显示出一个白色的字母“A”和“中”。

6 课后作业

修改 lcd_put_chinese 函数，可以指定字符颜色。

实现 lcd_put_str 函数，可以输出混合的中英文字符，比如“中国 china”，支持自动换行。

6.4 交叉编译程序：以 freetype 为例

使用 buildroot 来给 ARM 板编译程序、编译库会很简单，以后系统讲解 buildroot 时再使用 buildroot。

现在我们还是手工交叉编译 freetype，这种方法在编译、安装一些小程序时很有用。

6.4.1 程序运行的一些基础知识

1 编译程序时去哪找头文件？

系统目录：就是交叉编译工具链里的某个 include 目录；也可以自己指定：编译时用“ -I dir ”选项指定。

2 链接时去哪找库文件？

系统目录：就是交叉编译工具链里的某个 lib 目录；也可以自己指定：链接时用“ -L dir ”选项指定。

3 运行时去哪找库文件？

系统目录：就是板子上的/lib、/usr/lib 目录；也可以自己指定：运行程序用环境变量 LD_LIBRARY_PATH 指定。

4 运行时不需要头文件，所以头文件不用放到板子上

6.4.2 常见错误的解决方法

1 头文件问题

编译时找不到头文件。在程序中这样包含头文件：#include <xxx.h>

对于尖括号里的头文件，去哪里找它？

系统目录：就是交叉编译工具链里的某个 include 目录；

也可以自己指定：编译时用“ -I dir ”选项指定。

怎么确定“系统目录”？

执行下面命令确定目录：

```
echo 'main(){}'| arm-buildroot-linux-gnueabihf-gcc -E -v -
```

它会列出头文件目录、库目录(LIBRARY_PATH)。

你需要在头文件目录中确定有没有这个文件，或是自己指定头文件目录。

2 库文件问题

链接程序时如果有这样的提示：undefined reference to `xxx'，它表示 xxx 函数未定义。

那么解决方法有 2：

① 去写出这个函数

② 或是使用库函数，那需要在链接时指定库

怎么指定库？想链接 libabc.so，那链接时加上：-labc。

库在哪里？

- 系统目录：就是交叉编译工具链里的某个 lib 目录
- 也可以自己指定：链接时用“ -L dir ”选项指定

怎么确定“系统目录”？执行下面命令确定目录：

```
echo 'main(){}'| arm-buildroot-linux-gnueabihf-gcc -E -v -
```

它会列出头文件目录、库目录(LIBRARY_PATH)，你编译出库文件时，可以把它放入系统库目录。

3 运行问题

运行程序时找不到库：

```
error while loading shared libraries: libxxx.so:
cannot open shared object file: No such file or directory
```

找不到库，库在哪？

- 系统目录：就是板子上的/lib、/usr/lib 目录
- 也可以自己指定：

运行程序用环境变量 LD_LIBRARY_PATH 指定，执行以下的命令：

```
export  LD_LIBRARY_PATH=/xxx_dir  ;  ./test
```

或

```
LD_LIBRARY_PATH=/xxx_dir   ./test
```

6.4.3 交叉编译程序的万能命令

如果交叉编辑工具链的前缀是 `arm-buildroot-linux-gnueabihf-`，比如 `arm-buildroot-linux-gnueabihf-gcc`，交叉编译开源软件时，如果它里面有 configure，万能命令如下：

```
./configure  --host=arm-buildroot-linux-gnueabihf   --prefix=$PWD/tmp
make
make install
```

就可以在当前目录的 tmp 目录下看见 bin, lib, include 等目录，里面存有可执行程序、库、头文件。

1 把头文件、库文件放到工具链目录里

如果你编译的是一个库，请把得到的头文件、库文件放入工具链的 include、lib 目录里。别的程序要使用这些头文件、库时，会很方便。

工具链里可能有多个 include、lib 目录，放到哪里去？

执行下面命令来确定目录：

```
echo 'main(){}'| arm-buildroot-linux-gnueabihf-gcc -E -v -
```

它会列出头文件目录、库目录(LIBRARY_PATH)。

2 把库文件放到板子上的/lib 或/usr/lib 目录里

程序在板子上运行时，需要用到板子上/lib 或/usr/lib 下的库文件；程序运行时不需要头文件。

6.4.4 给 IMX6ULL 交叉编译 freetype

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\10_freetype\
      freetype-2.10.2.tar.xz
      libpng-1.6.37.tar.xz
      zlib-1.2.11.tar.gz
```

freetype 依赖于 libpng，libpng 又依赖于 zlib，所以我们应该：先编译安装 zlib，再编译安装 libpng，最后编译安装 freetype。但是，有些工具链里有 zlib，那就不用编译安装 zlib，比如 STM32MP157。

本节文档以 IMX6ULL 开发板中 arm-buildroot-linux-gnueabihf-gcc 工具链为例，对于其他开发板：工具链可能不一样，请灵活变通。

第1步 确定头文件、库文件在工具链中的目录

先设置交叉编译工具链：

```
export ARCH=arm
export CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueab
ihf_sdk-buildroot/bin
```

它里面有 zlib，跟着视频操作即可

以 IMX6ULL 开发板为例，它的工具链是 arm-buildroot-linux-gnueabihf-gcc，可以执行以下命令：

```
echo 'main(){}'| arm-buildroot-linux-gnueabihf-gcc -E -v -
```

可以确定头文件的系统目录为：

```
/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/
bin/../lib/gcc/arm-buildroot-linux-gnueabihf/7.5.0/include
```

库文件的系统目录为：

```
/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/
bin/../lib/gcc/arm-buildroot-linux-gnueabihf/7.5.0/../../../../arm-buildroot-linux-g
nueabihf/lib
```

第2步 交叉编译、安装 libpng

freetype 依赖于 libpng，所以需要先编译、安装 libpng。命令如下：

```
book@100ask$ cp /home/book/01_all_series_quickstart/04_嵌入式 Linux 应用开发基础知识/so
urce/10_freetype/libpng-1.6.37.tar.xz ./
book@100ask$ tar  xJf  libpng-1.6.37.tar.xz
book@100ask$ cd  libpng-1.6.37
book@100ask:~/libpng-1.6.37$ ./configure --host=arm-buildroot-linux-gnueabihf --pref
ix=$PWD/tmp
book@100ask:~/libpng-1.6.37$ make
```

```
book@100ask:~/libpng-1.6.37$ make install
book@100ask:~/libpng-1.6.37$ cd  tmp
book@100ask:~/libpng-1.6.37/tmp$ cp  include/*  -rf  /home/book/100ask_imx6ull-sdk/To
olChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin/../lib/gcc/arm-buildroot-lin
ux-gnueabihf/7.5.0/include
book@100ask:~/libpng-1.6.37/tmp$ cp  lib/*  -rfd   /home/book/100ask_imx6ull-sdk/Tool
Chain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin/../lib/gcc/arm-buildroot-linux
-gnueabihf/7.5.0/../../../../arm-buildroot-linux-gnueabihf/lib
```

第3步 交叉编译、安装 freetype

命令如下：

```
book@100ask$ cp /home/book/01_all_series_quickstart/04_嵌入式 Linux 应用开发基础知识/so
urce/10_freetype/freetype-2.10.2.tar.xz ./
book@100ask$ tar  xJf  freetype-2.10.2.tar.xz
book@100ask$ cd  freetype-2.10.2
book@100ask:~/freetype-2.10.2$  ./configure  --host=arm-buildroot-linux-gnueabihf   -
-prefix=$PWD/tmp
book@100ask:~/freetype-2.10.2$ make
book@100ask:~/freetype-2.10.2$ make install
book@100ask:~/freetype-2.10.2$ cd  tmp
book@100ask:~/freetype-2.10.2/tmp$ cp  include/*  -rf  /home/book/100ask_imx6ull-sdk/
ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/../arm-linux-gnuea
bihf/libc/usr/include
book@100ask:~/freetype-2.10.2/tmp$ cp  lib/*  -rfd   /home/book/100ask_imx6ull-sdk/To
olChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/../arm-linux-gnueabi
hf/libc/usr/lib/
```

6.5 使用 freetype 显示单个文字

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\10_freetype\
      01_wchar\test_wchar.c
      02_freetype_show_font\freetype_show_font.c
      03_freetype_show_font_angle\freetype_show_font_angle.c
```

6.5.1 矢量字体引入

使用点阵字库显示英文字母、汉字时，大小固定，如果放大缩小则会模糊甚至有锯齿出现，为了解决这个问题，引用矢量字体。

矢量字体形成分三步：

第1步 确定关键点，

第2步 使用数学曲线（贝塞尔曲线）连接头键点，

第3步 填充闭合区线内部空间。

什么是关键点？以字母“A”为例，它的的关键点如图 6.16 中的黄色所示。

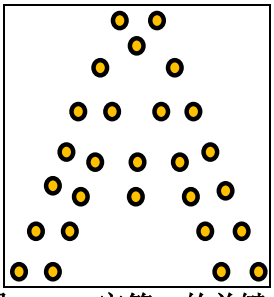


图 6.16 字符 A 的关键点

再用数学曲线(比如贝塞尔曲线)将关键点都连接起来，得到一系列的封闭的曲线，如图 6.17 所示：

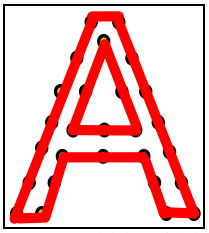


图 6.17 封闭关键点

最后把封闭空间填满颜色，就显示出一个 A 字母，如图 6.18 所示：

图 6.18 填充封闭曲线

如果需要放大或者缩小字体，关键点的相对位置是不变的，只要数学曲线平滑，字体就不会变形。

6.5.2 Freetype 介绍

Freetype 是开源的字体引擎库，它提供统一的接口来访问多种字体格式文件，从而实现矢量字体显示。我们只需要移植这个字体引擎，调用对应的 API 接口，提供字体文件，就可以让 freetype 库帮我们取出关键点、实现闭合曲线，填充颜色，达到显示矢量字体的目的。

关键点(glyph)存在字体文件中，Windows 使用的字体文件在 c:\Windows\Fonts 目录下，扩展名为 TTF 的都是矢量字库，本次使用实验使用的是新宋字体 simsun.ttc。

给定一个字符，怎么在字体文件中找到它的关键点？

首先要确定该字符的编码值：比如 ASCII 码、GB2312 码、UNICODE 码。如果字体文件支持某种编码格式(charset)，就可以使用这类编码值去找到该字符的关键点(glyph)。有些字体文件支持多种编码格式(charset)，这在文件中被

称为 charmaps(注意：这个单词是复数，意味着可能支持多种 charset)。

以 simsun.ttc 为例，该字体文件的格如下：头部含有 charmaps，可以使用某种编码值去 charmaps 中找到它对应的关键点。下图中的“A、B、中、国、韦”等只是 glyph 的示意图，表示关键点。

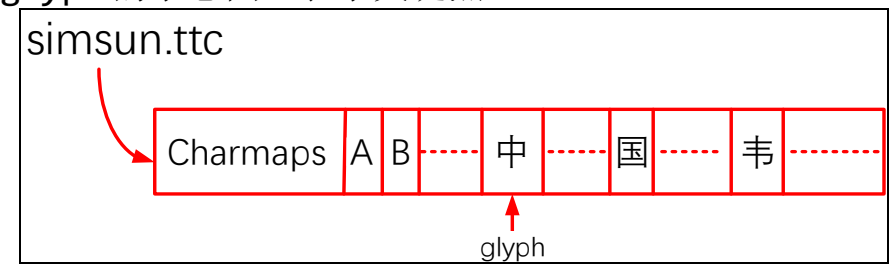


图 6.19 glyph

Charmaps 表示字符映射表，字体文件可能支持哪一些编码，GB2312、UNICODE、BIG5 或其他。如果字体文件支持该编码，使用编码值通过 charmap 就可以找到对应的 glyph，一般而言都支持 UNICODE 码。

有了以上基础，一个文字的显示过程可以概括如下：

- 给定一个字符可以确定它的编码值(ASCII、UNICODE、GB2312)；
- 设置字体大小；
- 根据编码值，从文件头部中通过 charmap 找到对应的关键点(glyph)，它会根据字体大小调整关键点；
- 把关键点转换为位图点阵；
- 在 LCD 上显示出来

从 https://www.freetype.org/ 可以下载到“freetype-doc-2.10.2.tar.xz”，下图中的文件就是官方文档：

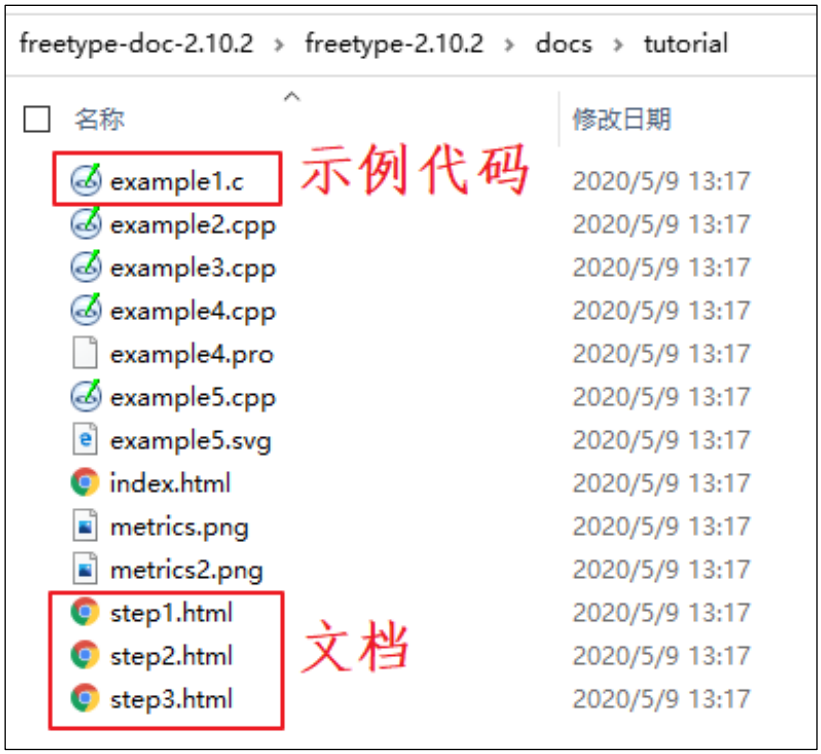


图 6.20 freetype 文件目录

参照上图中 step1，step2，step3 里的内容，可以学习如何使用 freetype 库，总结出下列步骤：

① 初始化：FT_InitFreetype
② 加载(打开)字体 Face：FT_New_Face
③ 设置字体大小：FT_Set_Char_Sizes 或 FT_Set_Pixel_Sizes
④ 选择 charmap：FT_Select_Charmap
⑤ 根据编码值 charcode 找到 glyph_index：glyph_index = FT_Get_Char_Index（face，charcode）
⑥ 根据 glyph_index 取出 glyph：FT_Load_Glyph（face，glyph_index）
⑦ 转为位图：FT_Render_Glyph
⑧ 移动或旋转:FT_Set_Transform
⑨ 最后显示出来。

上面的⑤⑥⑦可以使用一个函数代替：FT_Load_Char(face, charcode, FT_LOAD_RENDER)，它就可以得到位图。

6.5.3 在 LCD 上显示一个矢量字体

1 使用 wchar_t 获得字符的 UNICODE 值

要显示一个字符，首先要确定它的编码值。常用的是 UNICODE 编码，在程序里使用这样的语句定义字符串时，str 中保存的要么是 GB2312 编码值，要么是 UTF-8 格式的编码值，即使编译时使用“-fexec-charset=UTF-8”，str 中保存的也不是直接能使用的 UNICODE 值：

```
char  *str = “中”;
```

如果想在代码中能直接使用 UNICODE 值，需要使用 wchar_t，宽字符，示例代码如下：

```
01 #include <stdio.h>
02 #include <string.h>
03 #include <wchar.h>
04
05 int main( int argc, char** argv)
06 {
07      wchar_t *chinese_str = L"中 gif";
08      unsigned int *p = (wchar_t *)chinese_str;
09      int i;
10
11      printf("sizeof(wchar_t) = %d, str's Uniocde: \n", (int)sizeof(wchar_t));
12      for (i = 0; i < wcslen(chinese_str); i++)
13      {
14              printf("0x%x ", p[i]);
15      }
16      printf("\n");
17
18      return 0;
19 }}
```

UTF-8 格式保存 test_wchar.c，编译、测试命令如下：

```
book@100ask:~/10_freetype/01_wchar$ gcc -o test_wchar test_wchar.c
book@100ask:~/10_freetype/01_wchar$ ./test_wchar
sizeof(wchar_t) = 4, str's Uniocde:
0x4e2d 0x67 0x69 0x66
```

每个 wchar_t 占据 4 字节，可执行程序里 wchar_t 中保存的就是字符的 UNICODE 值。

注意：如果 test_wchar.c 是以 ANSI(GB2312)格式保存，那么需要使用以下命令来编译：

```
gcc -finput-charset=GB2312  -fexec-charset=UTF-8  -o test_wchar test_wchar.c
```

2 使用 freetype 得到位图

参考

```
freetype-doc-2.10.2\freetype-2.10.2\docs\tutorial\image.c
```

使用 freetype 显示一个字符并不难。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\10_freetype\02_freetype_show_font\freetype_show_font.c
```

要使用 freetype 得到一个字符的位图，只需要 4 个步骤，代码先贴出来再分析：

```
157:    /* 显示矢量字体 */
158:    error = FT_Init_FreeType( &library );              /* initialize library */
159:    /* error handling omitted */
160:
161:    error = FT_New_Face( library, argv[1], 0, &face ); /* create face object */
162:    /* error handling omitted */
163:    slot = face->glyph;
164:
165:    FT_Set_Pixel_Sizes(face, font_size, 0);
166:
167:    /* 确定座标:
168:     */
169:    //pen.x = 0;
170:    //pen.y = 0;
171:
172:    /* set transformation */
173:    //FT_Set_Transform( face, 0, &pen);
174:
175:    /* load glyph image into the slot (erase previous one) */
176:    error = FT_Load_Char( face, chinese_str[0], FT_LOAD_RENDER );
177:    if (error)
178:    {
179:        printf("FT_Load_Char error\n");
180:        return -1;
181:    }
```

因为不涉及旋转，
不涉及多个文字一起显示，
所以：
不需要调用FT_Set_Transform

图 6.21 使用 freetype 获取字符位图

① 初始化 freetype 库

```
158  error = FT_Init_FreeType( &library ); /* initialize library */
```

② 加载字体文件，保存在&face 中：

```
161 error = FT_New_Face( library, argv[1], 0, &face ); /* create face object */
162 /* error handling omitted */
163 slot = face->glyph;
```

第 163 行是从 face 中获得 FT_GlyphSlot，后面的代码中文字的位图就是保存

在 FT_GlyphSlot 里。

③ 设置字体大小

```
165     FT_Set_Pixel_Sizes(face, font_size, 0);
```

④ 根据编码值得到位图

使用 FT_Load_Char 函数，就可以实现这 3 个功能：

- 根据编码值获得 glyph_index：FT_Get_Char_Index
- 根据 glyph_idex 取出 glyph：FT_Load_Glyph
- 渲染出位图：FT_Render_Glyph

代码如下：

```
175     /* load glyph image into the slot (erase previous one) */
176     error = FT_Load_Char( face, chinese_str[0], FT_LOAD_RENDER );
```

执行 FT_Load_Char 之后，字符的位图被存在 slot->bitmap 里，即 face->glyph->bitmap。

3 在屏幕上显示位图

位图里的数据格式是怎样的？参考 example1.c 的代码，可以得到图 6.22：

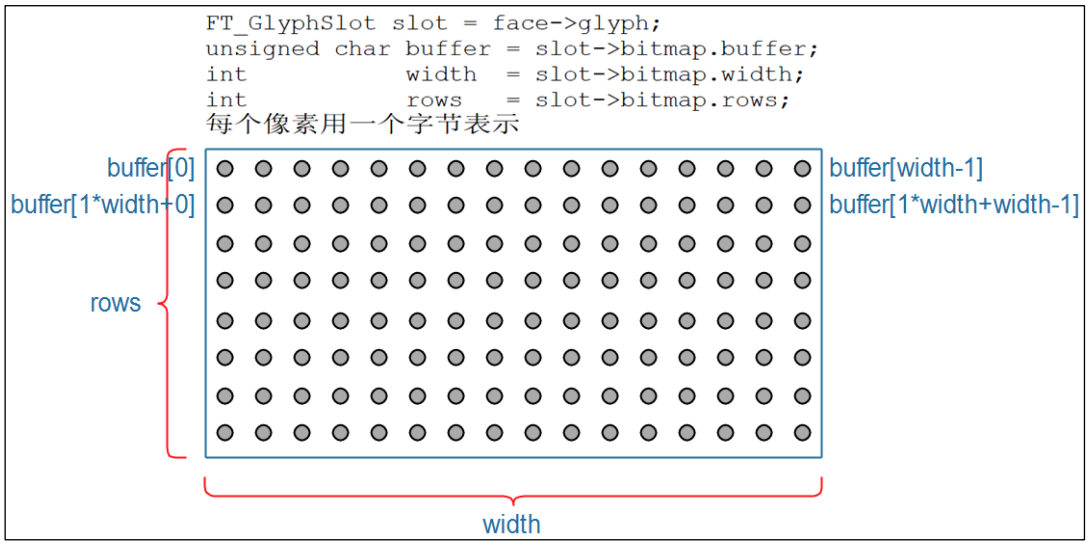


图 6.22 位图数据格式

要在屏幕上显示出这些位图，并不复杂：

```
183     draw_bitmap( &slot->bitmap,
184                  var.xres/2,
185                  var.yres/2);
```

draw_bitmap 函数代码如下，由于位图中每一个像素用一个字节来表示，在 0x00RRGGBB 的颜色格式中它只能表示蓝色，所以在 LCD 上显示出来的文字是蓝色的：

```
 85: void
 86: draw_bitmap( FT_Bitmap*bitmap,
 87:               FT_Int      x,
 88:               FT_Int      y)
 89: {
 90:     FT_Int  i, j, p, q;
 91:     FT_Int  x_max = x + bitmap->width;
 92:     FT_Int  y_max = y + bitmap->rows;
 93:
 94:     //printf("x = %d, y = %d\n", x, y);
 95:
 96:     for ( j = y, q = 0; j < y_max; j++, q++ )
 97:     {
 98:         for ( i = x, p = 0; i < x_max; i++, p++ )
 99:         {
100:             if ( i < 0      || j < 0       ||
101:                  i >= var.xres || j >= var.yres )
102:             continue;
103:
104:             //image[j][i] |= bitmap->buffer[q * bitmap->width + p];
105:             lcd_put_pixel(i, j, bitmap->buffer[q * bitmap->width + p]);
106:         }
107:     }
108: } « end draw_bitmap »
```

图 6.23 LCD显示位图函数

4 编译

编译命令(如果你使用的交叉编译链前缀不是 `arm-buildroot-linux-gnueabihf`，请自行修改命令)：

```
arm-buildroot-linux-gnueabihf-gcc -o freetype_show_font freetype_show_font.c  -lfree
type
```

它会提示如下错误：

```
freetype_show_font.c:12:10: fatal error: ft2build.h: No such file or directory
 #include <ft2build.h>
          ^~~~~~~~~~~~
compilation terminated.
```

我们不是已经编译过 freetype 并且把头文件复制进工具链里了吗？怎么还有这个错误？

我们编译出 freetype 后，得到的 ft2build.h 是位于 freetype2 目录里，我们把整个 freetype2 目录复制进了工具链里。

但是包括头文件时，用的是“#include <ft2build.h>”，要么改成：

```
#include <freetype2/ft2build.h>
```

要么把工具链里 incldue/freetype2/*.h 复制到上一级目录，我们使用这种方法：跟 freetype 文档保持一致。执行以下命令：

```
book@100ask$ cd   /home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueab
ihf_sdk-buildroot/arm-buildroot-linux-gnueabihf/sysroot/usr/include
book@100ask$ mv  freetype2/*   ./
```

然后再次执行以下命令：

```
arm-buildroot-linux-gnueabihf-gcc -o freetype_show_font freetype_show_font.c  -lfree
type
```

5 上机

将编译好的 freetype_show_font 文件与 simsun.ttc 字体文件拷贝至开发板，这 2 个文件放在同一个目录下，然后执行以下命令。

```
[root@100ask:~]# ./freetype_show_font  ./simsun.ttc
```

或

```
[root@100ask:~]# ./freetype_show_font  ./simsun.ttc   300
```

如果实验成功，我们将在屏幕中间看到一个蓝色的“繁”字。

6.5.4 在 LCD 上令矢量字体旋转某个角度

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\10_freetype\03_freetype_show_font_angle\freetype_show_font_angle.c
```

在实现显示一个矢量字体后，我们可以添加让该字旋转某个角度的功能，主要代码还是参照 example1.c。

1 关键代码

① 定义 2 个变量：角度、矩阵，如图 6.24：

```
121:    FT_Matrix     matrix;                 /* transformation matrix */
122:    double        angle;
123:
```

图 6.24 定义角度、矩阵变量

② 设置角度值：

```
130:    angle   = ( 1.0* strtoul(argv[2], NULL, 0) / 360 ) * 3.14159 * 2;       /* use 25 degrees     */
```

图 6.25 设置角度值

③ 设置矩阵、变形、加载位图：

```
176:    /* set up matrix */
177:    matrix.xx = (FT_Fixed)( cos( angle ) * 0x10000L );
178:    matrix.xy = (FT_Fixed)(-sin( angle ) * 0x10000L );
179:    matrix.yx = (FT_Fixed)( sin( angle ) * 0x10000L );
180:    matrix.yy = (FT_Fixed)( cos( angle ) * 0x10000L );
181:
182:    /* set transformation */
183:    FT_Set_Transform( face, &matrix, &pen);
184:
185:    /* load glyph image into the slot (erase previous one) */
186:    error = FT_Load_Char( face, chinese_str[0], FT_LOAD_RENDER );
187:    if (error)
188:    {
189:        printf("FT_Load_Char error\n");
190:        return -1;
191:    }
192:
```

设置矩阵

变形

得到位图

图 6.26 设置矩阵、变形、加载位图

2 编译

编译命令(如果你使用的交叉编译链前缀不是 arm-buildroot-linux-gnueabihf，请自行修改命令)：

```
arm-buildroot-linux-gnueabihf-gcc -o freetype_show_font_angle freetype_show_font_angle.c  -lfreetype -lm
```

3 上机

将编译好的 freetype_show_font_angle 文件与 simsun.ttc 字体文件拷贝至开发板，这 2 个文件放在同一个目录下，然后执行以下命令。

```
[root@100ask:~]# ./freetype_show_font_angle  ./simsun.ttc   90  200
```

如果实验成功，我们将在屏幕中间看到一个蓝色的、旋转了 90 度的“繁”字。

6.6 使用 freetype 显示一行文字

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\10_freetype\04_show_line\show_line.c
```

本节的目的：在 LCD 上指定一个左上角坐标(x, y)，把一行文字显示出来。下图中，文字的外框用虚线表示，外框的左上角坐标就是(x, y)。

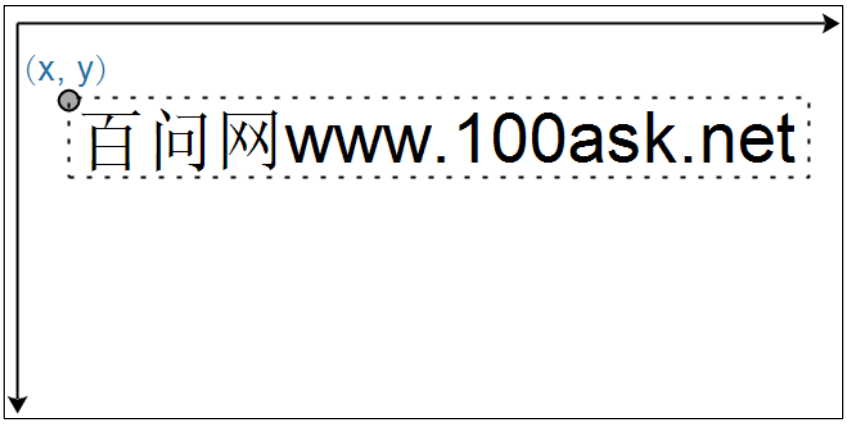


图 6.27 显示一行文字

6.6.1 笛卡尔坐标系

在 LCD 的坐标系中，原点在屏幕的左上角。对于笛卡尔坐标系，原点在左下角。freetype 使用笛卡尔坐标系，在显示时需要转换为 LCD 坐标系。

从图 6.28 可知，X 方向坐标值是一样的。

在 Y 方向坐标值需要换算，假设 LCD 的高度是 V。

在 LCD 坐标系中坐标是(x, y)，那么它在笛卡尔坐标系中的坐标值为(x, V-y)。

反过来也是一样的，在笛卡尔坐标系中坐标是(x, y)，那么它在 LCD 坐标系中坐标值为(x, V-y)。

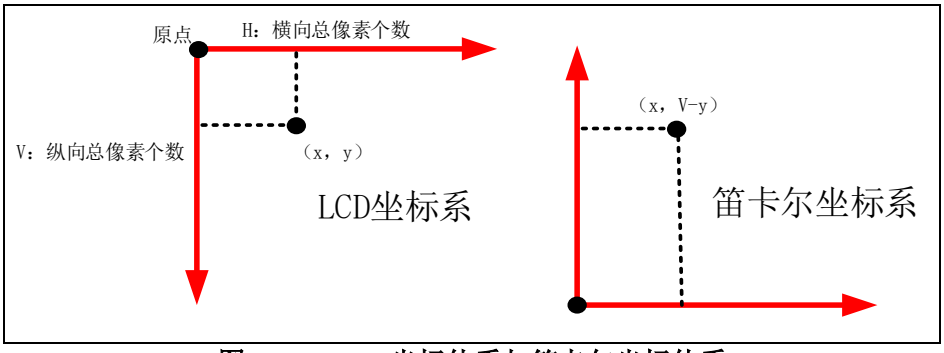


图 6.28 LCD 坐标体系与笛卡尔坐标体系

6.6.2 每个字符的大小可能不同

在使用 FT_Set_Pixel_Sizes 函数设置字体大小时，这只是“期望值”。比如“百问网 www.100ask.net”，如果把“.”显示得跟其他汉字一样大，不好看。

所以在显示一行文字时，后面文字的位置会受到前面文字的影响。

幸好，freetype 帮我们考虑到了这些影响。

对于 freetype 字体的尺寸(freetype Metrics)，需要参考图 6.29 这个文档：

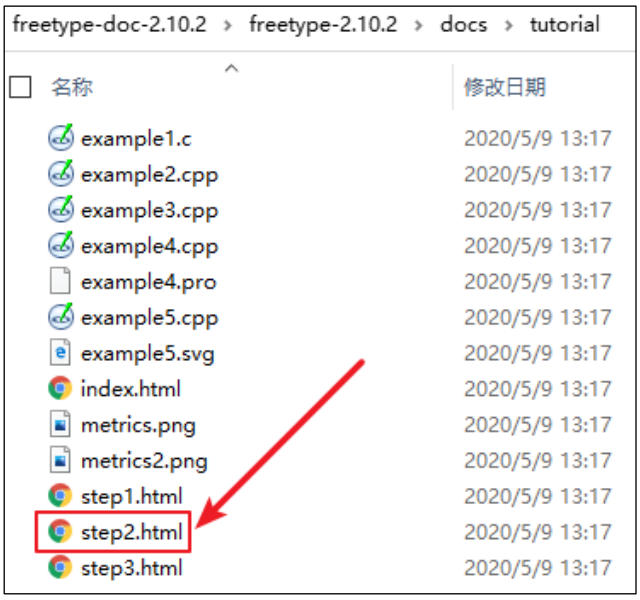


图 6.29 freetype 参考文档

上述文档中列出了一个图，摘录如下：

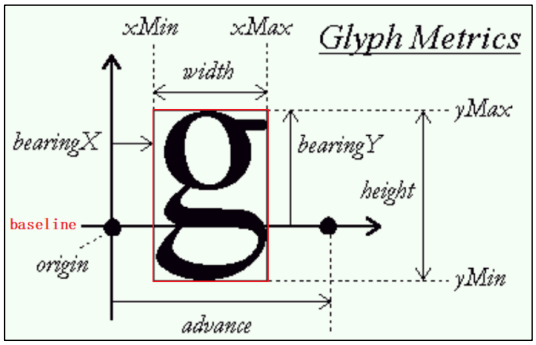


图 6.30 字符大小示意图

在显示一行文字时，这些文字会基于同一个基线来绘制位图：baseline。

在 baseline 上，每一个字符都有它的原点(origin)，比如上图中 baseline 左边的黑色圆点就是字母“g”的原点。当前 origin 加上 advance 就可以得到下一个字符的 origin，比如上图中 baseline 右边的黑色圆点。在显示一行中多个文件字时，后一个文字的原点依赖于前一个文字的原点及 advance。

字符的位图是有可能越过 baseline 的，比如上图中字母“g”在 baseline 下方还有图像。

上图中红色方框内就是字母“g”所点据的位图，它的四个角落不一定与原点重合。

上图中那些 xMin、xMax、yMin、yMax 如何获得？可以使用 FT_Glyph_Get_CBox 函数获得一个字体的这些参数，将会保存在一个 FT_BBox 结构体中，以后想计算一行文字的外框时要用到图 6.31 这些信息：

```
typedef struct  FT_BBox_
{
  FT_Pos   xMin, yMin;
  FT_Pos   xMax, yMax;

} FT_BBox;
```

图 6.31 字符外框信息

6.6.3 怎么在指定位置显示一行文字

要显示一行文字时，每一个字符都有自己外框：xMin、xMax、yMin、yMax。把这些字符的 xMin、yMin 中的最小值取出来，把这些字符的 xMax、yMax 中的最大值取出来，就可以确定这行文字的外框了。

要想在指定位置(x, y)显示一行文字，步骤如图 6.32 所示：

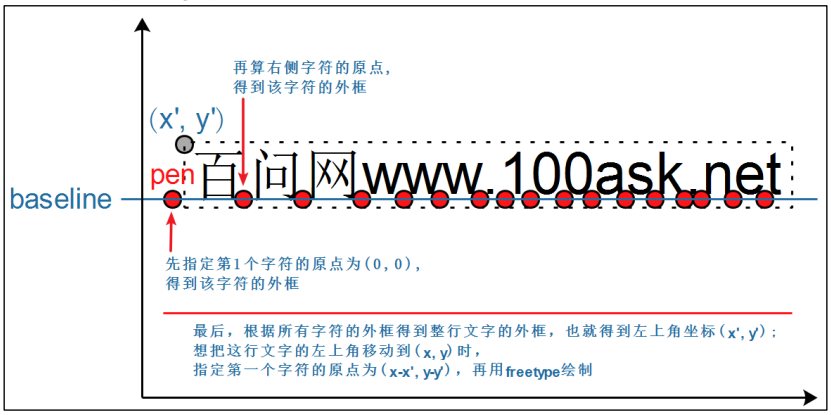


图 6.32 显示一行文字步骤

第1步 先指定第 1 个字符的原点 pen 坐标为(0, 0)，计算出它的外框

第2步 再计算右边字符的原点，也计算出它的外框

把所有字符都处理完后就可以得到一行文字的整体外框：假设外框左上角坐标为(x', y')。

第3步 想在(x, y)处显示这行文字，调整一下 pen 坐标即可。怎么调整？

pen 为(0, 0)时对应左上角(x', y')；那么左上角为(x, y)时就可以算出 pen 为(x-x', y-y')。

6.6.4 freetype 的几个重要数据结构

要想形象地理解程序，需要先介绍一下 freetype 中几个数据结构：

1 FT_Library

对应 freetype 库，使用 freetype 之前要先调用以下代码：

```
FT_Library  library; /* 对应 freetype 库 */
error = FT_Init_FreeType( &library ); /* 初始化 freetype 库 */
```

2 FT_Face

它对应一个矢量字体文件，在源码中使用 FT_New_Face 函数打开字体文件后，就可以得到一个 face。

为什么称之为 face？

估计是文字都是写在二维平面上的吧，正对着人脸？不用管原因了，总之认为它对应一个字体文件就可以。

代码如下：

```
error = FT_New_Face(library, font_file, 0, &face ); /* 加载字体文件 */
```

3 FT_GlyphSlot

插槽？用来保存字符的处理结果：比如转换后的 glyph、位图，如图 6.33：

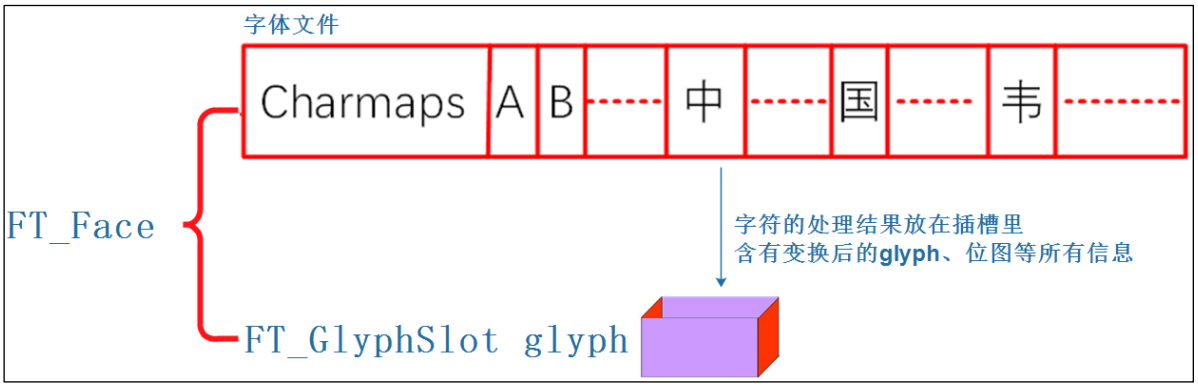


图 6.33 转换后的字符

一个 face 中有很多字符，生成一个字符的点阵位图时，位图保存在哪里？保存在插槽中：face->glyph。

生成第 1 个字符位图时，它保存在 face->glyph 中；生成第 2 个字符位图时，也会保存在 face->glyph 中，会覆盖第 1 个字符的位图。

代码如下：

```
FT_GlyphSlot  slot = face->glyph; /* 插槽：字体的处理结果保存在这里 */
```

4 FT_Glyph

字体文件中保存有字符的原始关键点信息，使用 freetype 的函数可以放大、缩小、旋转，这些新的关键点保存在插槽中(注意：位图也是保存在插槽中)。

新的关键点使用 FT_Glyph 来表示，可以使用这样的代码从 slot 中获得 glyph：

```
error = FT_Get_Glyph(slot , &glyph);
```

5 FT_BBox

FT_BBox 结构体定义如下，它表示一个字符的外框，即新 glyph 的外框：

```
typedef struct  FT_BBox_
{
  FT_Pos  xMin, yMin;
  FT_Pos  xMax, yMax;

} FT_BBox;
```

图 6.34 新的 glyph 外框

可以使用以下代码从 glyph 中获得这些信息：

```
FT_Glyph_Get_CBox(glyph, FT_GLYPH_BBOX_TRUNCATE, &bbox );
```

针对上述流程，示例代码如下：

```
FT_Library    library; /* 对应freetype库 */
FT_Face       face;    /* 对应字体文件 */
FT_GlyphSlot  slot;    /* 对应字符的处理结果：含glyph和位图 */
FT_Glyph      glyph;   /* 对应字符经过处理后的glyph即外形、轮廓 */
FT_BBox       bbox;    /* 字符的外框 */
FT_Vector     pen;     /* 字符的原点 */

error = FT_Init_FreeType( &library ); /* 初始化freetype库 */

error = FT_New_Face(library, font_file, 0, &face ); /* 加载字体文件 */

slot = face->glyph; /* 插槽：字体的处理结果保存在这里 */

FT_Set_Pixel_Sizes(face, 24, 0); /* 设置大小 */

/* 确定坐标 */
pen.x = 0; /* 单位：1/64 像素 */
pen.y = 0; /* 单位：1/64 像素 */

FT_Set_Transform( face, 0, &pen); /* 变形 */

/* 根据font_code加载字符，得到新的glyph和位图，结果保存在slot中 */
error = FT_Load_Char(face, font_code, FT_LOAD_RENDER);

error = FT_Get_Glyph(slot, &glyph); /* 从slot中得到新的glyph */

FT_Glyph_Get_CBox(glyph, FT_GLYPH_BBOX_TRUNCATE, &bbox ); /* 从glyph中得到字符外框 */

draw_bitmap(&slot->bitmap, lcd_x, lcd_y);  /* 位图也保存在slot->bitmap中 */
```

图 6.35 流程代码

6.6.5 计算一行文字的外框

前面提到过，一行文字中：后一个字符的原点=前一个字符的原点+advance。所以要计算一行文字的外框，需要按照排列顺序处理其中的每一个字符。
代码如下，注释写得很清楚了：

```
102 int compute_string_bbox(FT_Face      face, wchar_t *wstr, FT_BBox  *abbox)
103 {
104     int i;
105     int error;
106     FT_BBox bbox;
107     FT_BBox glyph_bbox;
108     FT_Vector pen;
109     FT_Glyph  glyph;
110     FT_GlyphSlot slot = face->glyph;
111
112     /* 初始化 */
113     bbox.xMin = bbox.yMin = 32000;
114     bbox.xMax = bbox.yMax = -32000;
115
116     /* 指定原点为(0, 0) */
117     pen.x = 0;
118     pen.y = 0;
119
120     /* 计算每个字符的 bounding box */
121     /* 先 translate, 再 load char, 就可以得到它的外框了 */
122     for (i = 0; i < wcslen(wstr); i++)
123     {
124         /* 转换：transformation */
125         FT_Set_Transform(face, 0, &pen);
126
127         /* 加载位图: load glyph image into the slot (erase previous one) */
128         error = FT_Load_Char(face, wstr[i], FT_LOAD_RENDER);
129         if (error)
130         {
131             printf("FT_Load_Char error\n");
132             return -1;
133         }
134
135         /* 取出 glyph */
136         error = FT_Get_Glyph(face->glyph, &glyph);
137         if (error)
138         {
139             printf("FT_Get_Glyph error!\n");
140             return -1;
141         }
142
143         /* 从 glyph 得到外框: bbox */
144         FT_Glyph_Get_CBox(glyph, FT_GLYPH_BBOX_TRUNCATE, &glyph_bbox);
145
146         /* 更新外框 */
147         if ( glyph_bbox.xMin < bbox.xMin )
148             bbox.xMin = glyph_bbox.xMin;
149
150         if ( glyph_bbox.yMin < bbox.yMin )
```

```
151             bbox.yMin = glyph_bbox.yMin;
152
153         if ( glyph_bbox.xMax > bbox.xMax )
154             bbox.xMax = glyph_bbox.xMax;
155
156         if ( glyph_bbox.yMax > bbox.yMax )
157             bbox.yMax = glyph_bbox.yMax;
158
159         /* 计算下一个字符的原点: increment pen position */
160         pen.x += slot->advance.x;
161         pen.y += slot->advance.y;
162     }
163
164     /* return string bbox */
165     *abbox = bbox;
166 }
```

6.6.6 调整原点并绘制

代码如下，也不复杂：

```
169 int display_string(FT_Face    face, wchar_t *wstr, int lcd_x, int lcd_y)
170 {
171     int i;
172     int error;
173     FT_BBox bbox;
174     FT_Vector pen;
175     FT_Glyph  glyph;
176     FT_GlyphSlot slot = face->glyph;
177
178     /* 把 LCD 坐标转换为笛卡尔坐标 */
179     int x = lcd_x;
180     int y = var.yres - lcd_y;
181
182     /* 计算外框 */
183     compute_string_bbox(face, wstr, &bbox);
184
185     /* 反推原点 */
186     pen.x = (x - bbox.xMin) * 64; /* 单位: 1/64 像素 */
187     pen.y = (y - bbox.yMax) * 64; /* 单位: 1/64 像素 */
188
189     /* 处理每个字符 */
190     for (i = 0; i < wcslen(wstr); i++)
191     {
192         /* 转换: transformation */
193         FT_Set_Transform(face, 0, &pen);
194
195         /* 加载位图: load glyph image into the slot (erase previous one) */
196         error = FT_Load_Char(face, wstr[i], FT_LOAD_RENDER);
197         if (error)
198         {
199             printf("FT_Load_Char error\n");
200             return -1;
201         }
202
203         /* 在 LCD 上绘制: 使用 LCD 坐标 */
```

```
204         draw_bitmap( &slot->bitmap,
205                      slot->bitmap_left,
206                      var.yres - slot->bitmap_top);
207
208         /* 计算下一个字符的原点: increment pen position */
209         pen.x += slot->advance.x;
210         pen.y += slot->advance.y;
211     }
212
213     return 0;
214 }
```

6.6.7 上机实验

编译命令(如果你使用的交叉编译链前缀不是 arm-buildroot-linux-gnueabihf，请自行修改命令)：

```
arm-buildroot-linux-gnueabihf-gcc  -o  show_line  show_line.c   -lfreetype
```

将编译好的 show_line 文件与 simsun.ttc 字体文件拷贝至开发板，这 2 个文件放在同一个目录下，然后执行以下命令(其中的 3 个数字分别表示 LCD 的 X 坐标、Y 坐标、字体大小)：

```
[root@100ask:~]# ./show_line ./simsun.ttc 10 200 80
```

如果实验成功，可以在 LCD 上看到一行文字“百问网 www.100ask.net”。

6.6.8 课后作业

① 修改程序，支持倾斜角度显示一行文字。
② 修改程序，支持显示多行文字。

第7章 输入系统应用编程

7.1 什么是输入系统

- 先来了解什么是输入设备？

常见的输入设备有键盘、鼠标、遥控杆、书写板、触摸屏等等,用户通过这些输入设备与 Linux 系统进行数据交换。

- 什么是输入系统？

输入设备种类繁多，能否统一它们的接口？既在驱动层面统一，也在应用程序层面统一？可以的。

Linux 系统为了统一管理这些输入设备，实现了一套能兼容所有输入设备的框架：输入系统。驱动开发人员基于这套框架开发出程序，应用开发人员就可以使用统一的 API 去使用设备。

7.2 输入系统框架及调试

7.2.1 框架概述

作为应用开发人员，可以只基于 API 使用输入子系统。但是了解内核中输入子系统的框架、了解数据流程，有助于解决开发过程中碰到的硬件问题、驱动问题。

输入系统框架如图 7.1 所示：

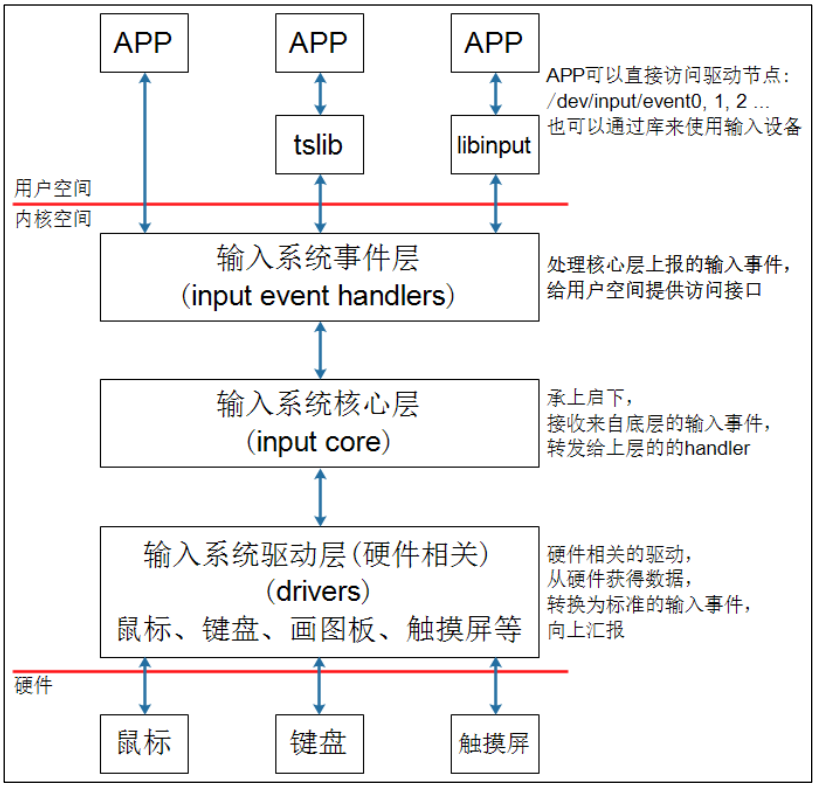


图 7.1 输入系统框架

假设用户程序直接访问/dev/input/event0 设备节点，或者使用 tslib 访问设备节点，数据的流程如下：

① APP 发起读操作，若无数据则休眠；

② 用户操作设备，硬件上产生中断；

③ 输入系统驱动层对应的驱动程序处理中断：

读取到数据，转换为标准的输入事件，向核心层汇报。

所谓输入事件就是一个“struct input_event”结构体。

④ 核心层可以决定把输入事件转发给上面哪个 handler 来处理：

从 handler 的名字来看，它就是用来处输入操作的。有多种 handler，比如：evdev_handler、kbd_handler、joydev_handler 等等。

最常用的是 evdev_handler：它只是把 input_event 结构体保存在内核 buffer 等，APP 来读取时就原原本本地返回。它支持多个 APP 同时访问输入设备，每个 APP 都可以获得同一份输入事件。

当 APP 正在等待数据时，evdev_handler 会把它唤醒，这样 APP 就可以返回数据。

⑤ APP 对输入事件的处理：

APP 获得数据的方法有 2 种：直接访问设备节点(比如/dev/input/event0,1,2,...)，或者通过 tslib、libinput 这类库来间接访问设备节点。这些库简化了对数据的处理。

要想深入理解整个输入系统，就必须研究内核的输入系统，这在后续的“驱动大全”中会讲解。

7.2.2 编写 APP 需要掌握的知识

基于编写应用程序的角度，只需要理解这些内容：

1 内核中怎么表示一个输入设备？

使用 input_dev 结构体来表示输入设备，它的内容如图 7.2：

```
// include/linux/input.h
struct input_dev {
    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];   // 支持哪类事件？key/rel/abs ？
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; // 支持按键的话，支持哪些按键
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; // 支持相对位移的话，支持哪些
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; // 支持绝对位移的话，支持哪些
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
```

图 7.2 输入设备结构体

2 APP 可以得到什么数据？

可以得到一系列的输入事件，就是一个一个“struct input_event”，它定义如图 7.3：

```
struct input_event {// include/uapi/linux/input.h
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};

struct timeval {// include/uapi/linux/time.h
    __kernel_time_t      tv_sec;     /* seconds */
    __kernel_suseconds_t    tv_usec;    /* microseconds */
};
```

图 7.3 输入事件结构体

每个输入事件 input_event 中都含有发生时间：timeval 表示的是“自系统启动以来过了多少时间”，它是一个结构体，含有“tv_sec、tv_usec”两项(即秒、微秒)。

输入事件 input_event 中更重要的是：type(哪类事件)、code(哪个事件)、value(事件值)，细讲如下：

① type：表示哪类事件

比如 EV_KEY 表示按键类、EV_REL 表示相对位移(比如鼠标)，EV_ABS 表示绝对位置(比如触摸屏)。有图 7.4 这几类事件(参考 Linux 内核头文件)：

```
/*
 * Event types
 *// include/uapi/linux/input-event-codes.h

#define EV_SYN          0x00
#define EV_KEY          0x01
#define EV_REL          0x02
#define EV_ABS          0x03
#define EV_MSC          0x04
#define EV_SW           0x05
#define EV_LED          0x11
#define EV_SND          0x12
#define EV_REP          0x14
#define EV_FF           0x15
#define EV_PWR          0x16
#define EV_FF_STATUS        0x17
#define EV_MAX          0x1f
#define EV_CNT          (EV_MAX+1)
```

图 7.4 EV_TYPE

② code：表示该类事件下的哪一个事件

比如对于 EV_KEY(按键)类事件，它表示键盘。键盘上有很多按键，比如数

字键 1、2、3，字母键 A、B、C 里等。所以可以有图 7.5 这些事件：

```
// include/uapi/linux/input-event-codes.h
#define KEY_RESERVED        0
#define KEY_ESC             1
#define KEY_1               2
#define KEY_2               3
#define KEY_3               4
#define KEY_4               5
#define KEY_5               6
#define KEY_6               7
#define KEY_7               8
#define KEY_8               9
#define KEY_9               10
#define KEY_0               11
#define KEY_MINUS           12
#define KEY_EQUAL           13
#define KEY_BACKSPACE       14
#define KEY_TAB             15
```

图 7.5 按键事件

对于触摸屏，它提供的是绝对位置信息，有 X 方向、Y 方向，还有压力值。所以 code 值有图 7.6 这些：

```
/*
 * Absolute axes
 */

#define ABS_X                   0x00
#define ABS_Y                   0x01
#define ABS_Z                   0x02
```

图 7.6 绝对位置信息

③ value：表示事件值

对于按键，它的 value 可以是 0(表示按键被按下)、1(表示按键被松开)、2(表示长按)；

对于触摸屏，它的 value 就是坐标值、压力值。

④ 事件之间的界线

APP 读取数据时，可以得到一个或多个数据，比如一个触摸屏的一个触点会上报 X、Y 位置信息，也可能会上报压力值。

- APP 怎么知道它已经读到了完整的数据？

驱动程序上报完一系列的数据后，会上报一个“同步事件”，表示数据上报完毕。APP 读到“同步事件”时，就知道已经读完了当前的数据。

同步事件也是一个 input_event 结构体，它的 type、code、value 三项都是 0。

3 输入子系统支持完整的 API 操作

支持这些机制：阻塞、非阻塞、POLL/SELECT、异步通知。

注意：如果你想深入理解上述机制，需要学习以下内容：

- 《第 5 篇 嵌入式 Linux 驱动开发基础知识 第十九章 驱动程序基石》

7.2.3 调试技巧

1 确定设备信息

输入设备的设备节点名为/dev/input/eventX(也可能是/dev/eventX，X表示0、1、2等数字)。查看设备节点，可以执行以下命令：

```
ls /dev/input/* -l
```

或

```
ls /dev/event* -l
```

可以看到图 7.7 类似下面的信息：

```
[root@100ask:~]# ls /dev/input/* -l
crw-rw---- 1 root input 13, 64 Feb  7 15:51 /dev/input/event0
crw-rw---- 1 root input 13, 65 Feb  7 15:51 /dev/input/event1

/dev/input/by-path:
total 0
lrwxrwxrwx 1 root root 9 Feb  7 15:51 platform-5c002000.i2c-event -> ../event0
lrwxrwxrwx 1 root root 9 Feb  7 15:51 platform-joystick-event -> ../event1
[root@100ask:~]#
```

图 7.7 查找设备节点

怎么知道这些设备节点对应什么硬件呢？可以在板子上执行以下命令：

```
cat /proc/bus/input/devices
```

这条指令的含义就是获取与 event 对应的相关设备信息，可以看到类似以下的结果：

```
[root@100ask:~]# cat /proc/bus/input/devices
I: Bus=0018 Vendor=0416 Product=038f Version=1060
N: Name="Goodix Capacitive TouchScreen"   触摸屏
P: Phys=input/ts
S: Sysfs=/devices/platform/soc/5c002000.i2c/i2c-2/2-005d/input/input0
U: Uniq=
H: Handlers=kbd event0   设备节点:/dev/input/event0
B: PROP=2
B: EV=b
B: KEY=400 0 0 0 0 0 0 20000000 0 0 0
B: ABS=2658000 3

I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="joystick"
P: Phys=gpio-keys/input0   GPIO按键
S: Sysfs=/devices/platform/joystick/input/input1
U: Uniq=
H: Handlers=kbd event1   设备节点:/dev/input/event1
B: PROP=0
B: EV=3
B: KEY=50000000
```

图 7.8 查看设备信息

那么这里的 I、N、P、S、U、H、B 对应的每一行是什么含义呢？

① I:id of the device(设备 ID)

该参数由结构体 struct input_id 来进行描述，驱动程序中会定义这样的结构体：

```
// include/uapi/linux/input.h
struct input_id {
    __u16 bustype;
    __u16 vendor;
    __u16 product;
    __u16 version;
};
```

图 7.9 设备 ID

② N:name of the device
设备名称
③ P:physical path to the device in the system hierarchy
系统层次结构中设备的物理路径。
④ S:sysfs path
位于 sys 文件系统的路径
⑤ U:unique identification code for the device(if device has it)
设备的唯一标识码
⑥ H:list of input handles associated with the device.
与设备关联的输入句柄列表。
⑦ B:bitmaps(位图)

```
PROP:device properties and quirks(设备属性)
EV:types of events supported by the device(设备支持的事件类型)
KEY:keys/buttons this device has(此设备具有的键/按钮)
MSC:miscellaneous events supported by the device(设备支持的其他事件)
LED:leds present on the device(设备上的指示灯)
```

值得注意的是 B 位图，比如上图中“B: EV=b”用来表示该设备支持哪类输入事件。b 的二进制是 1011，bit0、1、3 为 1，表示该设备支持 0、1、3 这三类事件，即 EV_SYN、EV_KEY、EV_ABS。

再举一个例子，“B: ABS=2658000 3”如何理解？

它表示该设备支持 EV_ABS 这一类事件中的哪一些事件。这是 2 个 32 位的数字：0x2658000、0x3，高位在前低位在后，组成一个 64 位的数字：“0x2658000,00000003”，数值为 1 的位有：0、1、47、48、50、53、54，即：0、1、0x2f、0x30、0x32、0x35、0x36，对应以下这些宏：

```
// include/uapi/linux/input-event-codes.h

#define ABS_X           0x00
#define ABS_Y           0x01

#define ABS_MT_SLOT         0x2f    /* MT slot being modified */
#define ABS_MT_TOUCH_MAJOR  0x30    /* Major axis of touching ellipse */
#define ABS_MT_TOUCH_MINOR  0x31    /* Minor axis (omit if circular) */
#define ABS_MT_WIDTH_MAJOR  0x32    /* Major axis of approaching ellipse */
#define ABS_MT_WIDTH_MINOR  0x33    /* Minor axis (omit if circular) */
#define ABS_MT_ORIENTATION  0x34    /* Ellipse orientation */
#define ABS_MT_POSITION_X   0x35    /* Center X touch position */
#define ABS_MT_POSITION_Y   0x36    /* Center Y touch position */
```

图 7.10 绝对信息宏定义

即这款输入设备支持上述的 ABS_X、ABS_Y、ABS_MT_SLOT、ABS_MT_TOUCH_MAJOR、ABS_MT_WIDTH_MAJOR、ABS_MT_POSITION_X、ABS_MT_POSITION_Y 这些绝对位置事件(它们的含义在后面讲解电容屏时再细讲)。

2 使用命令读取数据

调试输入系统时，直接执行类似下面的命令，然后操作对应的输入设备即可读出数据：

```
hexdump /dev/input/event0
```

在开发板上执行上述命令之后，点击按键或是点击触摸屏，就会打印图 7.11 信息：

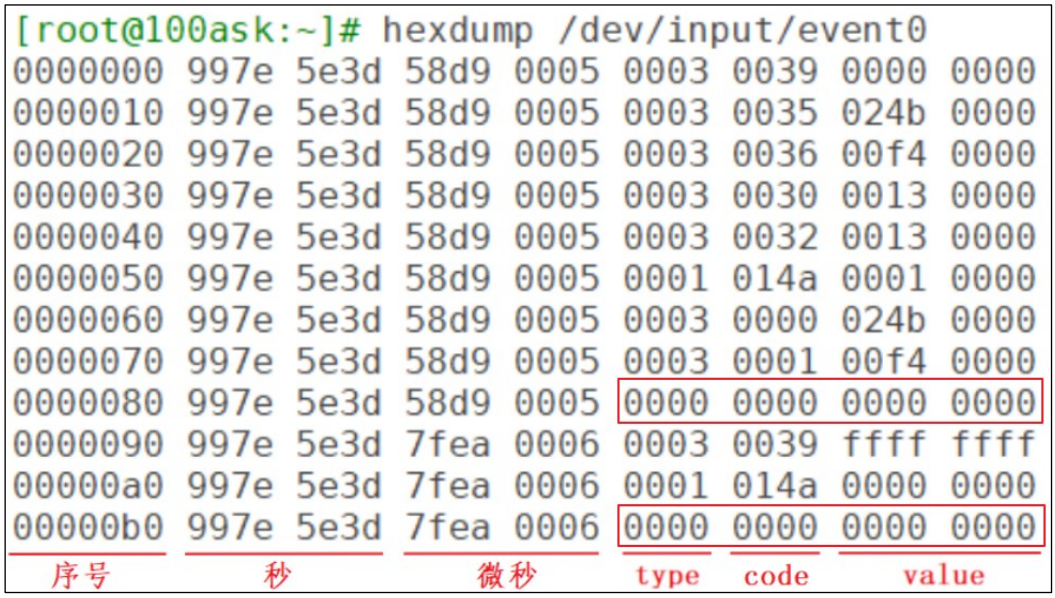


图 7.11 按键或触摸打印信息

图 7.11 中的 type 为 3，对应 EV_ABS；code 为 0x35 对应 ABS_MT_POSITION_X；code 为 0x36 对应 ABS_MT_POSITION_Y。

上图中还发现有 2 个同步事件：它的 type、code、value 都为 0。表示电容屏上报了 2 次完整的数据。

7.3 不使用库的应用程序示例

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\11_input\01_app_demo\
```

有些同学反馈：老师，你不从 0 写代码，我都不知道怎么写程序了。基础好的同学直接看这个文档，我录完这个视频后会补齐这个文档，你能看到这部分手册时肯定已经齐全了。

7.3.1 输入系统支持完整的 API 操作

支持这些机制：阻塞、非阻塞、POLL/SELECT、异步通知。

注意：如果你想深入理解上述机制，需要学习以下内容：

- 《>>>>>>>>第五篇第 19 章驱动程序基石》

作为 APP 开发人员，即使没有深入理解这些机制，也是可以编写出程序的。

7.3.2 APP 访问硬件的 4 种方式：妈妈怎么知道孩子醒了

图 7.12 举例示意图

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式

简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒

不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式

要浪费点时间，但是可以继续干活。

妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

妈妈、小孩互不耽误。

这 4 种方法没有优劣之分，在不同的场合使用不同的方法。

7.3.3 获取设备信息

通过 ioctl 获取设备信息，ioctl 的参数如下：

```
int ioctl(int fd, unsigned long request, ...);
```

有些驱动程序对 request 的格式有要求，它的格式如下：

```
// include/uapi/asm-generic/ioctl.h
#define _IOC(dir,type,nr,size) \
    (((dir)  << _IOC_DIRSHIFT) | \  // bit29
     ((type) << _IOC_TYPESHIFT) | \ // bit8
     ((nr)   << _IOC_NRSHIFT) | \   // bit0
     ((size) << _IOC_SIZESHIFT))    // bit16
```

图 7.13 request 格式要求

比如 dir 为_IOC_READ(即 2)时，表示 APP 要读数据；为_IOC_WRITE(即 4)时，

表示 APP 要写数据。

- size 表示这个 ioctl 能传输数据的最大字节数。
- type、nr 的含义由具体的驱动程序决定。

比如要读取输入设备的 evbit 时，ioctl 的 request 要写为“EVIOCGBIT(0, size)”，size 的大小可以由你决定：你想读多少字节就设置为多少。这个宏的定义如下：

```
#define EVIOCGBIT(ev,len)   _IOC(_IOC_READ, 'E', 0x20 + (ev), len)  /* get event bits */
```

图 7.14 size 宏定义

7.3.4 查询方式

APP 调用 open 函数时，传入“O_NONBLOCK”表示“非阻塞”。

APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据，否则也会立刻返回错误。

7.3.5 休眠-唤醒方式

APP 调用 open 函数时，不要传入“O_NONBLOCK”。

APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据；否则 APP 就会在内核态休眠，当有数据时驱动程序会把 APP 唤醒，read 函数恢复执行并返回数据给 APP。

7.3.6 POLL/SELECT 方式

1 功能介绍

POLL 机制、SELECT 机制是完全一样的，只是 APP 接口函数不一样。

简单地说，它们就是“定个闹钟”：在调用 poll、select 函数时可以传入“超时时间”。在这段时间内，条件合适时(比如有数据可读、有空间可写)就会立刻返回，否则等到“超时时间”结束时返回错误。

用法如下。

- APP 先调用 open 函数时。
- APP 不是直接调用 read 函数，而是先调用 poll 或 select 函数，这 2 个函数中可以传入“超时时间”。它们的作用是：如果驱动程序中有数据，则立刻返回；否则就休眠。在休眠期间，如果有人操作了硬件，驱动程序获得数据后就会把 APP 唤醒，导致 poll 或 select 立刻返回；如果在“超时时间”内无人操作硬件，则时间到后 poll 或 select 函数也会返回。APP 可以根据函数的返回值判断返回原因：有数据？无数据超时返回？
- APP 根据 poll 或 select 的返回值判断有数据之后，就调用 read 函数读取数据时，这时就会立刻获得数据。
- poll/select 函数可以监测多个文件，可以监测多种事件：

表 7-1 poll/select 监测事件

事件类型	说明
POLLIN	有数据可读
POLLRDNORM	等同于 POLLIN
POLLRDBAND	Priority band data can be read，有优先级较较高的“band data”可读 Linux 系统中很少使用这个事件
POLLPRI	高优先级数据可读
POLLOUT	可以写数据
POLLWRNORM	等同于 POLLOUT
POLLWRBAND	Priority data may be written
POLLERR	发生了错误
POLLHUP	挂起
POLLNVAL	无效的请求，一般是 fd 未 open

在调用 poll 函数时，要指明：

- 你要监测哪一个文件：哪一个 fd
- 你想监测这个文件的哪种事件：是 POLLIN、还是 POLLOUT

最后，在 poll 函数返回时，要判断状态。

应用程序代码如下：

```
struct pollfd fds[1];
int timeout_ms = 5000;
int ret;

fds[0].fd = fd;
fds[0].events = POLLIN;

ret = poll(fds, 1, timeout_ms);
if ((ret == 1) && (fds[0].revents & POLLIN))
{
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

2 现在编程：使用 POLL

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\11_input\01_app_demo\03_input_read_poll.c
```

精简过的核心源码如下：

```
14 int main(int argc, char **argv)
15 {
16      int fd;
26      struct pollfd fds[1];
……
61      fd = open(argv[1], O_RDWR | O_NONBLOCK);
……
94      while (1)
95      {
96              fds[0].fd = fd;
97              fds[0].events  = POLLIN;
98              fds[0].revents = 0;
99              ret = poll(fds, nfds, 5000);
```

```
100             if (ret > 0)
101             {
102                     if (fds[0].revents == POLLIN)
103                     {
104                             while (read(fd, &event, sizeof(event)) == sizeof(even
t))
105                             {
106                                     printf("get event: type = 0x%x, code = 0x%x, val
ue = 0x%x\n", event.type, event.code, event.value);
107                             }
108                     }
109             }
110             else if (ret == 0)
111             {
112                     printf("time out\n");
113             }
114             else
115             {
116                     printf("poll err\n");
117             }
118     }
119
120     return 0;
121 }
122
```

第 61 行：打开设备文件。

第 96~98 行：设置 pollfd 结构体。

第 96 行：想查询哪个文件(fd)？

第 97 行：想查询什么事件(POLLIN)？

第 98 行：先清除“返回的事件”(revents)。

第 99 行：使用 poll 函数查询事件，指定超时时间为 5000(ms)。

第 100、110 行判断返回值：大于 0 表示期待的事件发生了，等于 0 表示超时。

3 课后作业

- 使用 poll 函数监测多个输入设备。
- 使用 select 函数实现同样的功能。

7.3.7 异步通知方式

1 功能介绍

所谓同步，就是“你慢我等你”。

那么异步就是：你慢那你就自己玩，我做自己的事去了，有情况再通知我。

所谓异步通知，就是 APP 可以忙自己的事，当驱动程序用数据时它会主动给 APP 发信号，这会导致 APP 执行信号处理函数。

仔细想想“发信号”，这只有 3 个字，却可以引发很多问题：

- 谁发：驱动程序发

- 发什么：信号
- 发什么信号：SIGIO
- 怎么发：内核里提供有函数
- 发给谁：APP，APP 要把自己告诉驱动
- APP 收到后做什么：执行信号处理函数
- 信号处理函数和信号，之间怎么挂钩：APP 注册信号处理函数

小孩通知妈妈的事情有很多：饿了、渴了、想找人玩。

Linux 系统中也有很多信号，在 Linux 内核源文件 include\uapi\asm-generic\signal.h 中，有很多信号的宏定义：

```
#define SIGHUP       1
#define SIGINT       2
#define SIGQUIT      3
#define SIGILL       4
#define SIGTRAP      5
#define SIGABRT      6
#define SIGIOT       6
#define SIGBUS       7
#define SIGFPE       8
#define SIGKILL      9
#define SIGUSR1     10
#define SIGSEGV     11
#define SIGUSR2     12
#define SIGPIPE     13
#define SIGALRM     14
#define SIGTERM     15
#define SIGSTKFLT   16
#define SIGCHLD     17
#define SIGCONT     18
#define SIGSTOP     19
#define SIGTSTP     20
#define SIGTTIN     21
#define SIGTTOU     22
#define SIGURG      23
#define SIGXCPU     24
#define SIGXFSZ     25
#define SIGVTALRM   26
#define SIGPROF     27
#define SIGWINCH    28
#define SIGIO       29   <-- 驱动常用信号，表示有IO事件
#define SIGPOLL     SIGIO
```

图 7.15 信号宏定义

驱动程序通知 APP 时，它会发出“SIGIO”这个信号，表示有“IO 事件”要处理。

就 APP 而言，你想处理 SIGIO 信息，那么需要提供信号处理函数，并且要跟 SIGIO 挂钩。这可以通过一个 signal 函数来“给某个信号注册处理函数”，用法如下：

```
#include <signal.h>

typedef void (*sighandler_t)(int); // 1. 先编写函数

sighandler_t signal(int signum, sighandler_t handler); // 2. 注册
                    哪个信号？      信号处理函数
```

图 7.16 信号函数用法

除了注册 SIGIO 的处理函数，APP 还要做什么事？想想这几个问题：

- 内核里有那么多驱动，你想让哪一个驱动给你发 SIGIO 信号？

  APP 要打开驱动程序的设备节点。
- 驱动程序怎么知道要发信号给你而不是别人？APP 要把自己的进程 ID 告诉驱动程序。
- APP 有时候想收到信号，有时候又不想收到信号：应该可以把 APP 的意愿告诉驱动：设置 Flag 里面的 FASYNC 位为 1，使能“异步通知”。

2 应用编程

应用程序要做的事情有这几件：

① 编写信号处理函数：

```
static void sig_func(int sig)
{
    int val;
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

② 注册信号处理函数：

```
signal(SIGIO, sig_func);
```

③ 打开驱动：

```
fd = open(argv[1], O_RDWR);
```

④ 把进程 ID 告诉驱动：

```
fcntl(fd, F_SETOWN, getpid());
```

⑤ 使能驱动的 FASYNC 功能：

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
```

7.4 电阻屏和电容屏

触摸屏分为电阻屏、电容屏。电阻屏结构简单，在以前很流行；电容屏支持多点触摸，现在的手机基本都是使用电容屏。

注意：LCD、触摸屏不是一回事，LCD 是输出设备，触摸屏是输入设备。制作触摸屏时特意把它的尺寸做得跟 LCD 一模一样，并且把触摸屏覆盖在 LCD 上。

7.4.1 电阻屏

1 复习一下欧姆定律

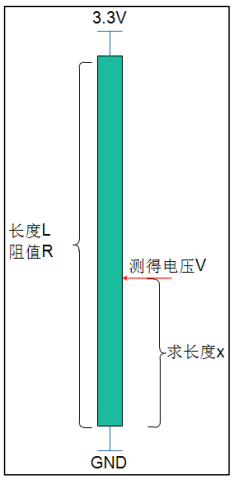


图 7.17 欧姆定律

图 7.17 中的电阻假设是均匀的，就是长度和阻值成正比关系。电阻长度为 L，阻值为 R，在两端施加 3.3V 电压。在某点测得电阻为 V，求上图中长度 X。

根据欧姆定律：3.3/R = V/Rx，

因为长度和阻值成正比关系，上述公式转换为：3.3/L = V/X，所以 X=LV/3.3。

2 电阻屏原理

电阻屏就是基于欧姆定律制作的，它有上下两层薄膜，这两层薄膜就是两个电阻，如下图所示：

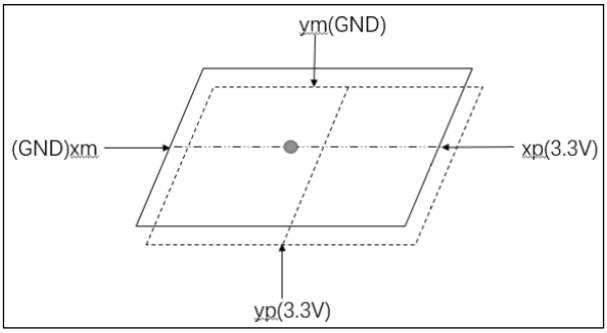


图 7.18 电阻屏原理

平时上下两层薄膜无触触，当点击触摸屏时，上下两层薄膜接触：这时就可以测量触点电压。过程如下：

① 测量 X 坐标：

在 xp、xm 两端施加 3.3V 电压，yp 和 ym 不施加电压(yp 就相当于探针)，测量 yp 电压值。该电压值就跟 X 坐标成正比关系，假设：

X = 3.3*Vyp/Xmax

② 测量 Y 坐标：

在 yp、ym 两端施加 3.3V 电压，xp 和 xm 不施加电压(xp 就相当于探针)，测量 xp 电压值。该电压值就跟 Y 坐标成正比关系，假设：

Y = 3.3*Vxp/Ymax

在实际使用时，电阻屏的 Xmax、Ymax 无从得知，所以使用之前要先较准：依次点击触摸屏的四个角和中心点，推算出 X、Y 坐标的公式：

X = func(Vyp)
Y = func(Vxp)

3 电阻屏数据

Linux 驱动程序中，会上报触点的 X、Y 数据，注意：这不是 LCD 的坐标值，需要 APP 再次处理才能转换为 LCD 坐标值。

对应的 input_event 结构体中，“type、code、value”如下：

按下时：

```
EV_KEY   BTN_TOUCH     1        /* 按下 */
EV_ABS   ABS_PRESSURE  1        /* 压力值，可以上报，也可以不报，可以是其他压力值 */
EV_ABS   ABS_X         x_value  /* X 坐标 */
EV_ABS   ABS_Y         y_value  /* Y 坐标 */
EV_SYNC  0             0        /* 同步事件 */
```

松开时：

```
EV_KEY   BTN_TOUCH     0        /* 松开 */
EV_ABS   ABS_PRESSURE  0        /* 压力值，可以上报，也可以不报 */
EV_SYNC  0             0        /* 同步事件 */
```

7.4.2 电容屏

1 原理

原理如图 7.19 所示：

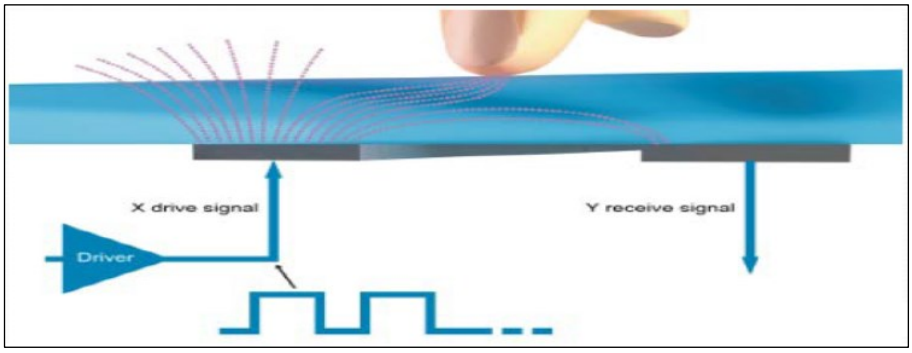


图 7.19 电容屏

电容屏中有一个控制芯片，它会周期性产生驱动信号，接收电极接收到信号，并可测量电荷大小。当电容屏被按下时，相当于引入了新的电容，从而影响了接收电极接收到的电荷大小。主控芯片根据电荷大小即可计算出触点位置。

怎么通过电荷计算出触点位置？这由控制芯片实现，这类芯片一般是 I2C 接口。

我们只需要编写程序，通过 I2C 读取芯片寄存器即可得到这些数据。

2 电容屏数据

参考文档：Linux 内核 Documentation\input\multi-touch-protocol.rst。

电容屏可以支持多点触摸(Multi touch)，驱动程序上报的数据中怎么分辨触点？

这有两种方法：Type A、Type B，这也对应两种类型的触摸屏：

① Type A

该类型的触摸屏不能分辨是哪一个触点，它只是把所有触点的坐标一股脑地上报，由软件来分辨这些数据分别属于哪一个触点。

Type A 已经过时，Linux 内核中都没有 Type A 的源码了。

② Type B

该类型的触摸屏能分辨是哪一个触点，上报数据时会先上报触点 ID，再上报它的数据。

具体例子如下，这是最简单的示例，使用场景分析来看看它上报的数据。

当有 2 个触点时(type, code, value)：

```
EV_ABS   ABS_MT_SLOT 0   // 这表示“我要上报一个触点信息了”，用来分隔触点信息
EV_ABS   ABS_MT_TRACKING_ID 45          // 这个触点的 ID 是 45
EV_ABS   ABS_MT_POSITION_X x[0]         // 触点 X 坐标
EV_ABS   ABS_MT_POSITION_Y y[0]         // 触点 Y 坐标
EV_ABS   ABS_MT_SLOT 1 // 这表示“我要上报一个触点信息了”，用来分隔触点信息
EV_ABS   ABS_MT_TRACKING_ID 46          // 这个触点的 ID 是 46
```

```
EV_ABS   ABS_MT_POSITION_X x[1]          // 触点 X 坐标
EV_ABS   ABS_MT_POSITION_Y y[1]          // 触点 Y 坐标
EV_SYNC  SYN_REPORT        0             // 全部数据上报完毕
```

当 ID 为 45 的触点正在移动时：

```
EV_ABS   ABS_MT_SLOT 0   // 这表示“我要上报一个触点信息了”，之前上报过 ID，就不用再上报 ID 了
EV_ABS   ABS_MT_POSITION_X x[0]   // 触点 X 坐标
EV_SYNC  SYN_REPORT        0      // 全部数据上报完毕
```

松开 ID 为 45 的触点时(在前面 slot 已经被设置为 0，这里这需要再重新设置 slot，slot 就像一个全局变量一样：如果它没变化的话，就无需再次设置)：

```
// 刚刚设置了 ABS_MT_SLOT 为 0，它对应 ID 为 45，这里设置 ID 为-1 就表示 ID 为 45 的触点被松开了
EV_ABS   ABS_MT_TRACKING_ID -1
EV_SYNC  SYN_REPORT        0     // 全部数据上报完毕
```

最后，松开 ID 为 46 的触点：

```
EV_ABS   ABS_MT_SLOT 1       // 这表示“我要上报一个触点信息了”，在前面设置过 slot 1 的 ID 为 46
EV_ABS   ABS_MT_TRACKING_ID -1  // ID 为-1，表示 slot 1 被松开，即 ID 为 46 的触点被松开
EV_SYNC  SYN_REPORT             // 全部数据上报完毕
```

3 电容屏的实验数据

假设你的开发板上电容屏对应的设备节点是/dev/input/event0，执行以下命令：

```
hexdump /dev/input/event0
```

然后用一个手指点击触摸屏，得到类似图 7.20 的数据：

```
[root@100ask:~]# hexdump /dev/input/event0
0000000 878d 5e3d 6026 0001 0003 0039 0000 0000 ABS_MT_TRACKING_ID 0
0000010 878d 5e3d 6026 0001 0003 0035 0236 0000 ABS_MT_POSITION_X
0000020 878d 5e3d 6026 0001 0003 0036 0146 0000 ABS_MT_POSITION_Y
0000030 878d 5e3d 6026 0001 0003 0030 0015 0000 ABS_MT_TOUCH_MAJOR
0000040 878d 5e3d 6026 0001 0003 0032 0015 0000 ABS_MT_WIDTH_MAJOR
0000050 878d 5e3d 6026 0001 0001 014a 0001 0000 EV_KEY BTN_TOUCH 1
0000060 878d 5e3d 6026 0001 0003 0000 0236 0000 ABS_X
0000070 878d 5e3d 6026 0001 0003 0001 0146 0000 ABS_Y
0000080 878d 5e3d 6026 0001 0000 0000 0000 0000 EV_SYNC
0000090 878d 5e3d 5cd4 0002 0003 0039 ffff ffff ABS_MT_TRACKING_ID -1
00000a0 878d 5e3d 5cd4 0002 0001 014a 0000 0000 EV_KEY BTN_TOUCH 0
00000b0 878d 5e3d 5cd4 0002 0000 0000 0000 0000 EV_SYNC
```

图 7.20 触摸屏打印信息

在上面的数据中，为了兼容老程序，它也上报了 ABS_X、ABS_Y 数据，电阻触摸屏就是使用这类型的数据。所以基于电阻屏的程序，也可以用在电容屏上。使用两个手指点击触摸屏时，得到类似如下的数据：

```
[root@100ask:~]# hexdump /dev/input/event0
0000000 8fb8 5e3d b2a9 0003 0003 002f 0000 0000 ABS_MT_SLOT  0
0000010 8fb8 5e3d b2a9 0003 0003 0039 0007 0000 ABS_MT_TRACKING_ID 7
0000020 8fb8 5e3d b2a9 0003 0003 0035 01d6 0000 ABS_MT_POSITION_X
0000030 8fb8 5e3d b2a9 0003 0003 0036 00e5 0000 ABS_MT_POSITION_Y
0000040 8fb8 5e3d b2a9 0003 0003 002f 0001 0000 ABS_MT_SLOT 1
0000050 8fb8 5e3d b2a9 0003 0003 0039 0008 0000 ABS_MT_TRACKING_ID 8
0000060 8fb8 5e3d b2a9 0003 0003 0035 0285 0000 ABS_MT_POSITION_X
0000070 8fb8 5e3d b2a9 0003 0003 0036 00fd 0000 ABS_MT_POSITION_Y
0000080 8fb8 5e3d b2a9 0003 0003 0030 0016 0000 ABS_MT_TOUCH_MAJOR
0000090 8fb8 5e3d b2a9 0003 0003 0032 0016 0000 ABS_MT_WIDTH_MAJOR
00000a0 8fb8 5e3d b2a9 0003 0001 014a 0001 0000 EV_KEY BTN_TOUCH 1
00000b0 8fb8 5e3d b2a9 0003 0003 0000 01d6 0000 ABS_X
00000c0 8fb8 5e3d b2a9 0003 0003 0001 00e5 0000 ABS_Y
00000d0 8fb8 5e3d b2a9 0003 0000 0000 0000 0000 EV_SYNC
00000e0 8fb8 5e3d 54c7 0004 0003 002f 0000 0000 ABS_MT_SLOT 0
00000f0 8fb8 5e3d 54c7 0004 0003 0039 ffff ffff ABS_MT_TRACKING_ID -1
0000100 8fb8 5e3d 54c7 0004 0003 002f 0001 0000 ABS_MT_SLOT 1
0000110 8fb8 5e3d 54c7 0004 0003 0039 ffff ffff ABS_MT_TRACKING_ID -1
0000120 8fb8 5e3d 54c7 0004 0001 014a 0000 0000 EV_KEY BTN_TOUCH 0
0000130 8fb8 5e3d 54c7 0004 0000 0000 0000 0000 EV_SYNC
```

图 7.21 双指触摸时打印信息

为了兼容老程序，它也上报了 ABS_X、ABS_Y 数据，但是只上报第 1 个触点的数据。

7.5 tslib

tslib 是一个触摸屏的开源库，可以使用它来访问触摸屏设备，可以给输入设备添加各种“filter”（过滤器，就是各种处理），地址是：http://www.tslib.org/。

编译 tslib 后，可以得到 libts 库，还可以得到各种工具：较准工具、测试工具。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\11_input\02_tslib\
```

7.5.1 tslib 框架分析

tslib 的主要代码如图 7.22：

```
src/ 接口函数
    ts_setup.c
    ts_open.c
    ts_config.c
plugins/ 插件/module
    linear.c
    dejitter.c
    pthres.c
    input-raw.c
tests/  测试程序
    ts_test.c
    ts_test_mt.c
    ts_print.c
    ts_print_mt.c
```

图 7.22 tslib 主要代码

核心在于“plugins”目录里的“插件”，或称为“module”。这个目录下的每个文件都是一个 module，每个 module 都提供 2 个函数：read、read_mt，前者用于读取单点触摸屏的数据，后者用于读取多点触摸屏的数据。

要分析 tslib 的框架，先看看示例程序怎么使用，我们参考 ts_test.c 和 ts_test_mt.c，前者用于一般触摸屏(比如电阻屏、单点电容屏)，后者用于多点触摸屏。

一个图就可以弄清楚 tslib 的框架：

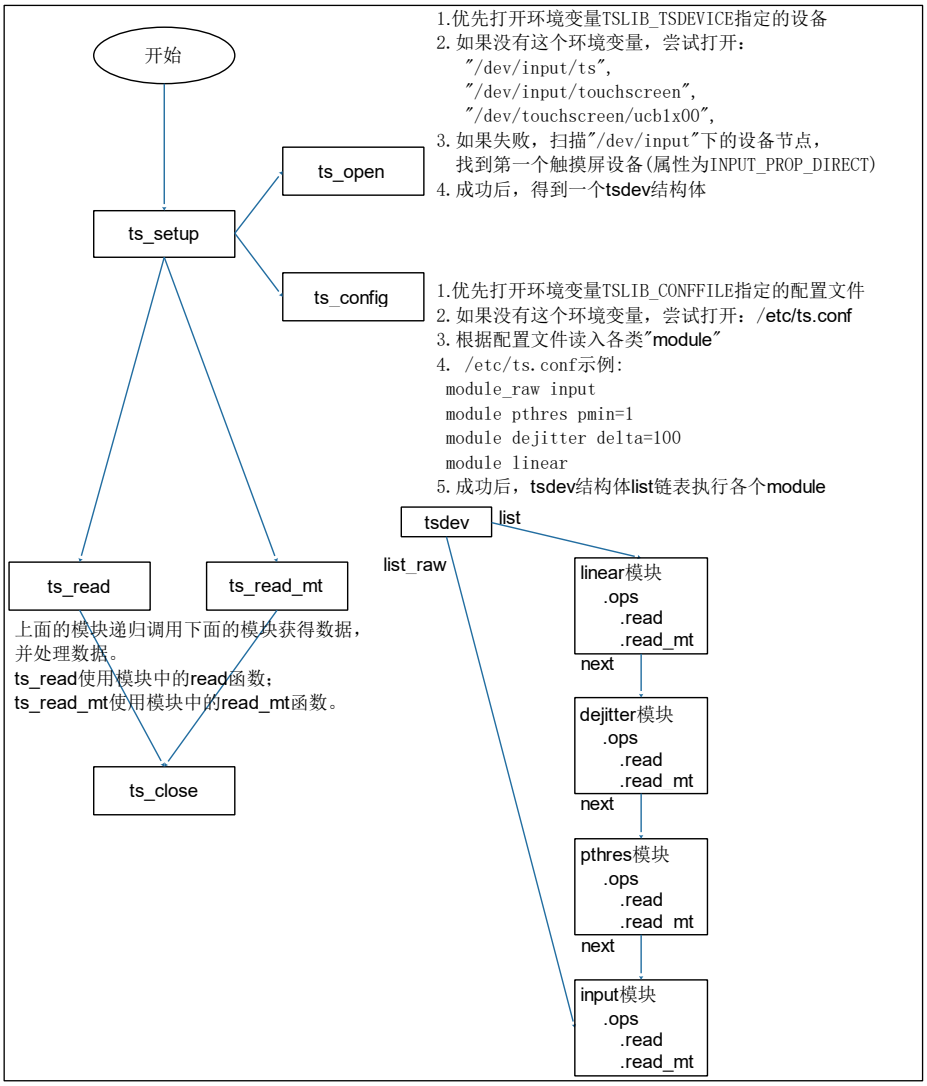


图 7.23 tslib 框架

调用 ts_open 后，可以打开某个设备节点，构造出一个 tsdev 结构体。然后调用 ts_config 读取配置文件的处理，假设/etc/ts.conf 内容如下：

```
module_raw input
module pthres pmin=1
module dejitter delta=100
module linear
```

每行表示一个“module”或“moduel_raw”。

对于所有的“module”，都会插入 tsdev.list 链表头，也就是 tsdev.list 执行配置文件中最后一个“module”，配置文件中第一个“module”位于链表的尾部。

对于所有的“module_raw”，都会插入 tsdev.list_raw 链表头，一般只有一个“module_raw”。

注意：tsdev.list 中最后一个“module”会指向 ts_dev.list_raw 的头部。

无论是调用 ts_read 还是 ts_read_mt，都是通过 tsdev.list 中的模块来处理数据的。这写模块是递归调用的，比如 linear 模块的 read 函数如图 7.24：

```c
static int linear_read(struct tslib_module_info *info, struct ts_sample *samp,
                       int nr_samples)
{
    struct tslib_linear *lin = (struct tslib_linear *)info;
    int ret;
    int xtemp, ytemp;

    ret = info->next->ops->read(info->next, samp, nr_samples);
    if (ret >= 0) {
        int nr;

        for (nr = 0; nr < ret; nr++, samp++) {
        #ifdef DEBUG
            fprintf(stderr,
                "BEFORE CALIB--------------------> %d %d %d\n",
```

1.使用链表中下一个模块来读数据

2.读出数据后，处理

图 7.24 linear 的 read

linear 模块的 read_raw 函数如图 7.25：

```c
static int linear_read_mt(struct tslib_module_info *info,
                          struct ts_sample_mt **samp,
                          int max_slots, int nr_samples)
{
    struct tslib_linear *lin = (struct tslib_linear *)info;
    int ret;
    int xtemp, ytemp;
    int i;
    int nr;

    if (!info->next->ops->read_mt)
        return -ENOSYS;

    ret = info->next->ops->read_mt(info->next, samp, max_slots, nr_samples);
    if (ret < 0)
        return ret;

    for (nr = 0; nr < ret; nr++) {
    #ifdef DEBUG
        printf("LINEAR:    read %d samples (mem: %d nr x %d slots)\n",
               ret, nr_samples, max_slots);
```

1.使用链表中下一个模块来读数据

2.得到数据后，处理

图 7.25 linear 的 read_raw

因为是递归调用，所有最先使用 input 模块读取设备节点得到原始数据，再依次经过 pthres 模块、dejitter 模块、linear 模块处理后，才返回最终数据。

7.5.2 交叉编译、测试 tslib

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\11_input\02_tslib\tslib-1.21.tar.xz
```

本节会涉及这些内容：交叉编译 tslib，运行自带的测试程序，后面会自己写一个简单的测试程序。

1 交叉编译 tslib

参考：

《>>>>>>>>第三篇 2.6.3 配置交叉编译工具链》

《>>>>>>>>第四篇 6.4 交叉编译程序：以 freetype 为例》

```
// 对于 IMX6ULL，命令如下
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
```

```
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin
```

交叉编译 tslib：

```
// 对于 IMX6ULL，命令如下
./configure --host=arm-linux-gnueabihf  --prefix=/
make
make install DESTDIR=$PWD/tmp
```

确定工具链中头文件、库文件目录：

```
// 对于 IMX6ULL，命令如下
echo 'main(){}'| arm-linux-gnueabihf-gcc -E -v -
```

把头文件、库文件放到工具链目录下：

```
// 对于 IMX6ULL，命令如下
cd tslib-1.21/tmp/
cp include/* /home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/include
cp -d lib/*so*  /home/book/100ask_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/lib/
```

2 测试 tslib

把库文件放到单板上：运行程序要用。先在开发板上使用 NFS 挂载 Ubuntu 的目录，再把前面编译出来的 tslib-1.21/tmp/部分文件复制到板子上，示例命令如下：

```
cp  /mnt/tslib-1.21/tmp/lib/*so*  -d     /lib
cp  /mnt/tslib-1.21/tmp/bin/*            /bin
cp  /mnt/tslib-1.21/tmp/etc/ts.conf  -d  /etc
```

对于 IMX6ULL，首先需要关闭默认的 qt gui 程序，才可以执行 ts_test_mt 测试命令，关闭 qt 命令如下所示：

```
mv  /etc/init.d/S07hmi  /root
reboot
```

在单板上执行测试程序：

```
ts_test_mt
```

7.5.3 自己写一个测试程序

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\11_input\02_tslib\mt_cal_distance.c
```

1 接口函数深入分析

前面演示过，用两个手指点击屏幕时，可以得到类似图 7.26 的数据：

```
[root@100ask:~]# hexdump /dev/input/event0
0000000 8fb8 5e3d b2a9 0003 0003 002f 0000 0000 ABS_MT_SLOT  0
0000010 8fb8 5e3d b2a9 0003 0003 0039 0007 0000 ABS_MT_TRACKING_ID 7
0000020 8fb8 5e3d b2a9 0003 0003 0035 01d6 0000 ABS_MT_POSITION_X
0000030 8fb8 5e3d b2a9 0003 0003 0036 00e5 0000 ABS_MT_POSITION_Y
0000040 8fb8 5e3d b2a9 0003 0003 002f 0001 0000 ABS_MT_SLOT 1
0000050 8fb8 5e3d b2a9 0003 0003 0039 0008 0000 ABS_MT_TRACKING_ID 8
0000060 8fb8 5e3d b2a9 0003 0003 0035 0285 0000 ABS_MT_POSITION_X
0000070 8fb8 5e3d b2a9 0003 0003 0036 00fd 0000 ABS_MT_POSITION_Y
0000080 8fb8 5e3d b2a9 0003 0003 0030 0016 0000 ABS_MT_TOUCH_MAJOR
0000090 8fb8 5e3d b2a9 0003 0003 0032 0016 0000 ABS_MT_WIDTH_MAJOR
00000a0 8fb8 5e3d b2a9 0003 0001 014a 0001 0000 EV_KEY BTN_TOUCH 1
00000b0 8fb8 5e3d b2a9 0003 0003 0000 01d6 0000 ABS_X
00000c0 8fb8 5e3d b2a9 0003 0003 0001 00e5 0000 ABS_Y
00000d0 8fb8 5e3d b2a9 0003 0000 0000 0000 0000 EV_SYNC
00000e0 8fb8 5e3d 54c7 0004 0003 002f 0000 0000 ABS_MT_SLOT 0
00000f0 8fb8 5e3d 54c7 0004 0003 0039 ffff ffff ABS_MT_TRACKING_ID -1
0000100 8fb8 5e3d 54c7 0004 0003 002f 0001 0000 ABS_MT_SLOT 1
0000110 8fb8 5e3d 54c7 0004 0003 0039 ffff ffff ABS_MT_TRACKING_ID -1
0000120 8fb8 5e3d 54c7 0004 0001 014a 0000 0000 EV_KEY BTN_TOUCH 0
0000130 8fb8 5e3d 54c7 0004 0000 0000 0000 0000 EV_SYNC
```

图 7.26 双指触摸屏幕的打印信息

驱动程序使用 slot、tracking_id 来标识一个触点；当 tracking_id 等于 -1 时，标识这个触点被松开了。

触摸屏可能支持多个触点，比如 5 个：tslib 为了简化处理，即使只有 2 个触点，ts_read_mt 函数也会返回 5 个触点数据，可以根据标志位判断数据是否有效。

ts_read_mt 函数原型如图 7.27：

```
int ts_read_mt(struct tsdev *ts, struct ts_sample_mt **samp, int max_slots,
               int nr)
```

图 7.27 ts_read_mt 函数原型

假设 nr 设置为 1，max_slots 设置为 5，那么读到的数据保存在：samp[0][0]、samp[0][1]、samp[0][2]、samp[0][3]、samp[0][4]中。

假设 nr 设置为 2，max_slots 设置为 5，那么读到的数据保存在：samp[0][0]、samp[0][1]、samp[0][2]、samp[0][3]、samp[0][4]和 samp[1][0]、samp[1][1]、samp[1][2]、samp[1][3]、samp[1][4]中。

ts_sample_mt 结构体如图 7.28：

```
struct ts_sample_mt {
    /* ABS_MT_* event codes. linux/include/uapi/linux/input-event-codes.h
     * has the definitions.
     */
    int     x;
    int     y;
    unsigned int    pressure;
    int     slot;
    int     tracking_id;

    int     tool_type;
    int     tool_x;
    int     tool_y;
    unsigned int    touch_major;
    unsigned int    width_major;
    unsigned int    touch_minor;
    unsigned int    width_minor;
    int     orientation;
    int     distance;
    int     blob_id;

    struct timeval  tv;

    /* BTN_TOUCH state */
    short       pen_down;

    /* valid is set != 0 if this sample
     * contains new data; see below for the
     * bits that get set.
     * valid is set to 0 otherwise
     */
    short       valid;
} « end ts_sample_mt » ;
```

valid非0时，
表示含有新数据

图 7.28 ts_sample_mt 结构体

2 编写代码

实现一个程序，不断打印 2 个触点的距离。

思路：假设是 5 点触摸屏，调用一次 ts_read_mt 可以得到 5 个新数据；使用新旧数据进行判断，如果有 2 个触点，就打印出距离。

第8章 网络通信

8.1 网络通信概述

本章配套的视频，我没有重新录制，而是修改这个视频得来：2012 年录制的“第 3 期项目视频”中的“第 1 课第 6.1 节_一小时学会网络编程_两个简单例子_tcp_udp”。

在 2012 年录制的视频，质量一点都不比现在的差。

悦己之作，方能悦人。很高兴，我从来没降低要求。

使用 GIT 下载资料后，视频中涉及的文档和图片，位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\doc_pic\08.网络编程 source\12_socket\
```

8.1.1 IP 和端口

所有的数据传输，都有三个要素 ：源、目的、长度。

怎么表示源或者目的呢？请看图 8.1：

图 8.1 网络源和目的

所以，在网络传输中需要使用“IP 和端口”来表示源或目的。

8.1.2 网络传输中的 2 个对象：server 和 client

我们经常访问网站，这涉及 2 个对象：网站服务器，浏览器。网站服务器平时安静地呆着，浏览器主动发起数据请求。网站服务器、浏览器可以抽象成 2 个软件的概念：server 程序、client 程序。

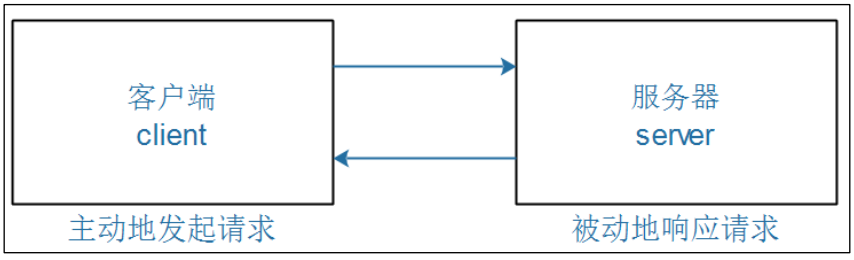


图 8.2 网络客户端和服务器

8.1.3 两种传输方式：TCP/UDP

在一般的网络书籍中，网络协议被分为 5 层，如图 8.3 所示：

图 8.3 网络协议层

应用层：它是体系结构中的最高层，直接为用户的应用进程（例如电子邮件、文件传输和终端仿真）提供服务。在因特网中的应用层协议很多，如支持万维网应用的 HTTP 协议，支持电子邮件的 SMTP 协议，支持文件传送的 FTP 协议，DNS，POP3，SNMP，Telnet 等等。

运输层：负责向两个主机中进程之间的通信提供服务。

运输层主要使用以下两种协议：

① 传输控制协议 TCP(Transmission Control Protocol)：面向连接的，数据传输的单位是报文段，能够提供可靠的交付。

② 用户数据包协议 UDP(User Datagram Protocol)：无连接的，数据传输的单位是用户数据报，不保证提供可靠的交付，只能提供“尽最大努力交付”。

- 网络层：负责将被称为数据包（datagram）的网络层分组从一台主机移动到另一台主机。
- 链路层：因特网的网络层通过源和目的地之间的一系列路由器路由数据报。
- 物理层：在物理层上所传数据的单位是比特。物理层的任务就是透明地传送比特流。

这些层对于初学者来说很难理解，我们只需要知道：我们需要使用“运输层”编写应用程序，我们的应用程序位于“应用层”。

使用“运输层”时，可以选择 TCP 协议，也可以选择 UDP 协议。

1 TCP 和 UDP 原理上的区别

TCP 向它的应用程序提供了面向连接的服务。这种服务有 2 个特点：可靠传输、流量控制（即发送方/接收方速率匹配）。它包括了应用层报文划分为短报文，并提供拥塞控制机制。

UDP 协议向它的应用程序提供无连接服务。它没有可靠性，没有流量控制，也没有拥塞控制。

2 为何存在 UDP 协议

既然 TCP 提供了可靠数据传输服务，而 UDP 不能提供，那么 TCP 是否总是首选呢？

答案是否定的，因为有许多应用更适合用 UDP，举个例子：视频通话时，使用 UDP，偶尔的丢包、偶尔的花屏时可以忍受的；如果使用 TCP，每个数据包都要确保可靠传输，当它出错时就重传，这会导致后续的数据包被阻滞，视频效果反而不好。

使用 UDP 时，有如下特点：

① 关于何时发送什么数据控制的更为精细

采用 UDP 时只要应用进程将数据传递给 UDP，UDP 就会立即将其传递给网络层。而 TCP 有重传机制，而不管可靠交付需要多长时间。但是实时应用通常不希望过分的延迟报文段的传送，且能容忍一部分数据丢失。

② 无需建立连接，不会引入建立连接时的延迟。

③ 无连接状态，能支持更多的活跃客户。

④ 分组首部开销较小。

3 TCP/UDP 网络通信大概交互图

下面我们分别画出运用 TCP 协议和运用 UDP 协议的客户端和服务器大概交互图。

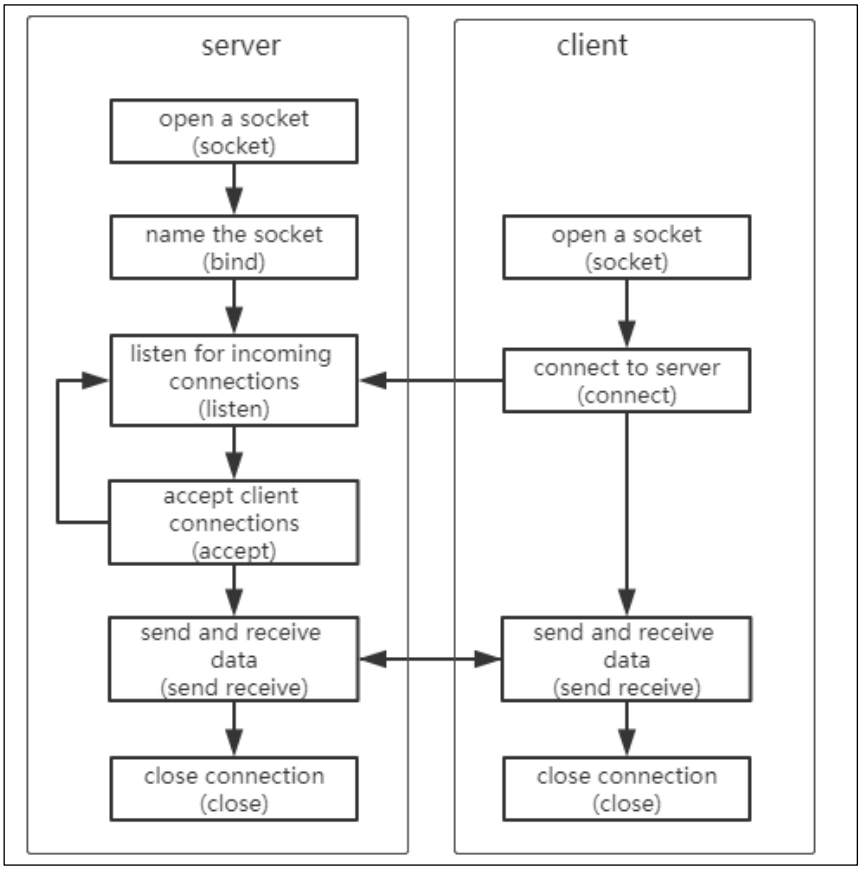


图 8.4 面向连接的 TCP 流模式

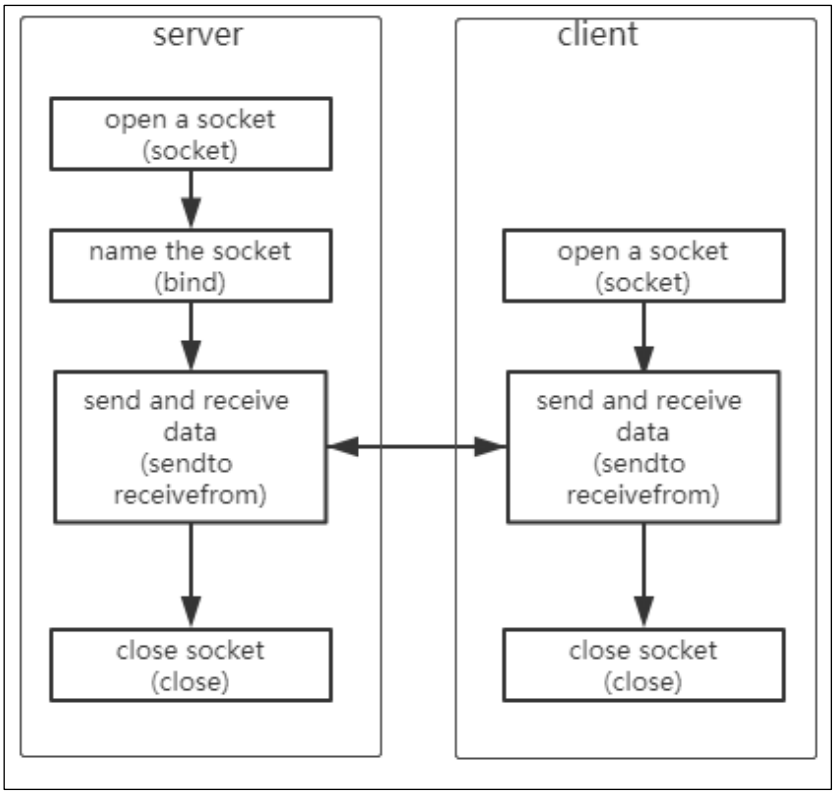


图 8.5UDP 用户数据包模式

8.2 网络编程主要函数介绍

8.2.1 socket 函数

```
int socket(int domain, int type,int protocol);
```

此函数用于创建一个套接字。

- **domain** 是网络程序所在的主机采用的通讯协族(AF_UNIX 和 AF_INET 等)。
  - AF_UNIX 只能够用于单一的 Unix 系统进程间通信，而 AF_INET 是针对 Internet 的，因而可以允许远程通信使用。
- **type** 是网络程序所采用的通讯协议(SOCK_STREAM,SOCK_DGRAM 等)。
  - SOCK_STREAM 表明用的是 TCP 协议，这样会提供按顺序的，可靠，双向，面向连接的比特流。
  - SOCK_DGRAM 表明用的是 UDP 协议，这样只会提不可靠，无连接的通信。
- 关于 **protocol**，由于指定了 type，所以这个地方一般只要用 0 来代替就可以了。

此函数执行成功时返回文件描述符，失败时返回-1,看 errno 可知道出错的详细情况。

8.2.2 bind 函数

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

从函数用于将地址绑定到一个套接字。

- **sockfd** 是由 socket 函数调用返回的文件描述符。
- **my_addr** 是一个指向 sockaddr 的指针。
- **addrlen** 是 sockaddr 结构的长度。

sockaddr 的定义：

```
struct sockaddr{
unisgned short  as_family;
char sa_data[14];
};
```

不过由于系统的兼容性，我们一般使用另外一个结构(struct sockaddr_in) 来代替。

sockaddr_in 的定义：

```
struct sockaddr_in{
unsigned short   sin_family;
unsigned short     sin_port;
struct in_addr   sin_addr;
unsigned char   sin_zero[8];
}
```

如果使用 Internet 所以 sin_family 一般为 AF_INET。

- sin_addr 设置为 INADDR_ANY 表示可以和任何的主机通信。
- sin_port 是要监听的端口号。
- bind 将本地的端口同 socket 返回的文件描述符捆绑在一起.成功是返回 0，失败的情况和 socket 一样。

8.2.3 listen 函数

```
int listen(int sockfd,int backlog);
```

此函数宣告服务器可以接受连接请求。

- **sockfd** 是 bind 后的文件描述符。
- **backlog** 设置请求排队的最大长度。当有多个客户端程序和服务端相连时，使用这个表示可以介绍的排队长度。
- listen 函数将 bind 的文件描述符变为监听套接字，返回的情况和 bind 一样。

8.2.4 accept 函数

```
int accept(int sockfd, struct sockaddr *addr,int *addrlen);
```

服务器使用此函数获得连接请求，并且建立连接。

- **sockfd** 是 listen 后的文件描述符。
- **addr，addrlen** 是用来给客户端的程序填写的,服务器端只要传递指针就可以了， bind,listen 和 accept 是服务器端用的函数。
- accept 调用时，服务器端的程序会一直阻塞到有一个客户程序发出了连接。accept 成功时返回最后的服务器端的文件描述符，这个时候服务器端可以向该描述符写信息了，失败时返回-1 。

8.2.5 connect 函数

```
int connect(int sockfd, struct sockaddr * serv_addr,int addrlen);
```

可以用 connect 建立一个连接，在 connect 中所指定的地址是想与之通信的服务器的地址。

- **sockfd** 是 socket 函数返回的文件描述符。
- **serv_addr** 储存了服务器端的连接信息，其中 sin_add 是服务端的地址。
- **addrlen** 是 serv_addr 的长度
- connect 函数是客户端用来同服务端连接的.成功时返回 0，sockfd 是同服务端通讯的文件描述符，失败时返回-1。

8.2.6 send 函数

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- **sockfd** 指定发送端套接字描述符；
- **buf** 指明一个存放应用程序要发送数据的缓冲区；
- **len** 指明实际要发送的数据的字节数；
- **flags** 一般置 0。
- 客户或者服务器应用程序都用 send 函数来向 TCP 连接的另一端发送数据

8.2.7 recv 函数

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- **sockfd** 指定接收端套接字描述符；
- **buf** 指明一个缓冲区，该缓冲区用来存放 recv 函数接收到的数据；
- **len** 指明 buf 的长度；
- **flags** 一般置 0。
- 客户或者服务器应用程序都用 recv 函数从 TCP 连接的另一端接收数据。

8.2.8 recvfrom 函数

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                      struct sockaddr *src_addr, socklen_t *addrlen);
```

- recvfrom 通常用于无连接套接字，因为此函数可以获得发送者的地址。
- **src_addr** 是一个 struct sockaddr 类型的变量，该变量保存源机的 IP 地址及端口号。
- **addrlen** 常置为 sizeof （struct sockaddr）。

8.2.9 sendto 函数

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                    const struct sockaddr *dest_addr, socklen_t addrlen);
```

- sendto 和 send 相似，区别在于 sendto 允许在无连接的套接字上指定一个目标地址。
- **dest_addr** 表示目地机的 IP 地址和端口号信息，

- **addrlen** 常常被赋值为 sizeof （struct sockaddr）。
- **sendto** 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1。

8.3 TCP 编程

本章配套的视频，我没有重新录制，而是修改这个视频得来：2012 年录制的“第 3 期项目视频”中的“第 1 课第 6.1 节_一小时学会网络编程_两个简单例子_tcp_udp”。

在 2012 年录制的视频，质量一点都不比现在的差。

悦己之作，方能悦人。很高兴，我从来没降低要求。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\12_socket\tcp\
```

8.4 UDP 编程

本章配套的视频，我没有重新录制，而是修改这个视频得来：2012 年录制的“第 3 期项目视频”中的“第 1 课第 6.1 节_一小时学会网络编程_两个简单例子_tcp_udp”。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\source\12_socket\udp\udp2\
```

第9章 多线程编程

本章将分为两大部分进行讲解，第一部分将引出线程的使用场景及基本概念，通过示例代码来说明一个线程创建到退出到回收的基本流程。第二部分则会通过示例代码来说明如果控制好线程，从临界资源访问与线程的执行顺序控制上引出互斥锁、信号量的概念与使用方法。

9.1 线程的使用

9.1.1 为什么要使用多线程

在编写代码时，是否会遇到以下的场景会感觉到难以下手？

要做 2 件事，一件需要阻塞等待，另一件需要实时进行。例如播放器：一边在屏幕上播放视频，一边在等待用户的按键操作。如果使用单线程的话，程序必须一会查询有无按键，一会播放视频。查询按键太久，就会导致视频播放卡顿；视频播放太久，就无法及时响应用户的操作。并且查询按键和播放视频的代码混杂在一起，代码丑陋。

如果使用多线程，线程 1 单独处理按键，线程 2 单独处理播放，可以完美解决上述问题。

9.1.2 线程概念

所谓线程，就是操作系统所能调度的最小单位。普通的进程，只有一个线程在执行对应的逻辑。我们可以通过多线程编程，使一个进程可以去执行多个不同的任务。相比多进程编程而言，线程享有共享资源，即在进程中出现的全局变量，每个线程都可以去访问它，与进程共享“4G”内存空间，使得系统资源消耗减少。本章节来讨论 Linux 下 POSIX 线程。

9.1.3 线程的标识 pthread_t

对于进程而言，每一个进程都有一个唯一对应的 PID 号来表示该进程，而对于线程而言，也有一个“类似于进程的 PID 号”，名为 tid，其本质是一个 pthread_t 类型的变量。线程号与进程号是表示线程和进程的唯一标识，但是对于线程号而言，其仅仅在其所属的进程上下文中才有意义。

```
获取线程号
#include <pthread.h>
pthread_t pthread_self(void);
成功：返回线程号
```

在程序中，可以通过函数 pthread_self，来返回当前线程的线程号，例程 1 给出了打印线程 tid 号。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
```

source\13_thread\01_文档配套源码\Pthread_Text1.c

测试例程 1：（Phtread_txex1.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3
4   int main()
5   {
6       pthread_t tid = pthread_self();//获取主线程的 tid 号
7       printf("tid = %lu\n",(unsigned long)tid);
8       return 0;
9   }
```

注意：因采用 POSIX 线程接口，故在要编译的时候包含 pthread 库，使用 gcc 编译应 gcc xxx.c -lpthread 方可编译多线程程序。

编译结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text1.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
tid = 140253463397568
```

图 9.1 例程 1 编译结果

9.1.4 线程的创建

怎么创建线程呢？使用 pthread_create 函数：

```
创建线程
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routi
ne) (void *), void *arg);
```

- 该函数第一个参数为 pthread_t 指针，用来保存新建线程的线程号；
- 第二个参数表示了线程的属性，一般传入 NULL 表示默认属性；
- 第三个参数是一个函数指针，就是线程执行的函数。这个函数返回值为 void*，形参为 void*。
- 第四个参数则表示为向线程处理函数传入的参数，若不传入，可用 NULL 填充，有关线程传参后续小节会有详细的说明，接下来通过一个简单例程来使用该函数创建出一个线程。

使用 GIT 下载所有源码后，本节源码位于如下目录：

01_all_series_quickstart\
  04_嵌入式 Linux 应用开发基础知识\
    source\13_thread\01_文档配套源码\Pthread_Text2.c

测试例程 2：（Phtread_txex2.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <errno.h>
5
6   void *fun(void *arg)
7   {
8       printf("pthread_New = %lu\n",(unsigned long)pthread_self());//打印线程的 tid 号
9   }
10
11  int main()
12  {
```

```
13
14      pthread_t tid1;
15      int ret = pthread_create(&tid1,NULL,fun,NULL);//创建线程
16      if(ret != 0){
17          perror("pthread_create");
18          return -1;
19      }
20
21      /*tid_main 为通过 pthread_self 获取的线程 ID，tid_new 通过执行 pthread_create 成功
后 tid 指向的空间*/
22      printf("tid_main = %lu tid_new = %lu \n",(unsigned long)pthread_self(),(unsig
ned long)tid1);
23
24      /*因线程执行顺序随机，不加 sleep 可能导致主线程先执行，导致进程结束，无法执行到子线
程*/
25      sleep(1);
26
27      return 0;
28  }
29
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text2.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
tid_main = 140172800087872 tid_new = 140172791580416
pthread_New = 140172791580416
```

图 9.2 例程 2 运行结果

通过 pthread_create 确实可以创建出来线程，主线程中执行 pthread_create 后的 tid 指向了线程号空间，与子线程通过函数 pthread_self 打印出来的线程号一致。

特别说明的是，当主线程伴随进程结束时，所创建出来的线程也会立即结束，不会继续执行。并且创建出来的线程的执行顺序是随机竞争的，并不能保证哪一个线程会先运行。可以将上述代码中 sleep 函数进行注释，观察实验现象。

去掉上述代码 25 行后运行结果：.

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text2.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
tid_main = 140028882487104 tid_new = 140028873979648
```

图 9.3 去掉 25 行后的运行结果

上述运行代码 3 次，其中有 2 次被进程结束，无法执行到子线程的逻辑，最后一次则执行到了子线程逻辑后结束的进程。如此可以说明，线程的执行顺序不受控制，且整个进程结束后所产生的线程也随之被释放，在后续内容中将会描述如何控制线程执行。

9.1.5 向线程传入参数

pthread_create()的最后一个参数的为 void*类型的数据，表示可以向线程传递一个 void*数据类型的参数，线程的回调函数中可以获取该参数，例程 3 举例了如何向线程传入变量地址与变量值。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
```

04_嵌入式 Linux 应用开发基础知识\source\13_thread\01_文档配套源码\Pthread_Text3.c

测试例程 3：（Phtread_txex3.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <errno.h>
5
6   void *fun1(void *arg)
7   {
8       printf("%s:arg = %d Addr = %p\n",__FUNCTION__,*(int *)arg,arg);
9   }
10
11  void *fun2(void *arg)
12  {
13      printf("%s:arg = %d Addr = %p\n",__FUNCTION__,(int)(long)arg,arg);
14  }
15
16  int main()
17  {
18
19      pthread_t tid1,tid2;
20      int a = 50;
21      int ret = pthread_create(&tid1,NULL,fun1,(void *)&a);//创建线程传入变量 a 的地址
22      if(ret != 0){
23          perror("pthread_create");
24          return -1;
25      }
27      ret = pthread_create(&tid2,NULL,fun2,(void *)(long)a);//创建线程传入变量 a 的值
28      if(ret != 0){
29          perror("pthread_create");
30          return -1;
31      }
32      sleep(1);
33      printf("%s:a = %d Add = %p \n",__FUNCTION__,a,&a);
34      return 0;
35  }
36
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text3.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
fun2:arg = 50 Addr = 0x32
fun1:arg = 50 Addr = 0x7ffe9c005b70
main:a = 50 Add = 0x7ffe9c005b70
```

图 9.4 例程 3 运行结果

本例程展示了如何利用线程创建函数的第四个参数向线程传入数据，举例了如何以地址的方式传入值、以变量的方式传入值，例程代码的 21 行，是将变量 a 先行取地址后，再次强制类型转化为 void*后传入线程，线程处理的回调函数中，先将万能指针 void*转化为 int*，再次取地址就可以获得该地址变量的值，其本质在于地址的传递。例程代码的 27 行，直接将 int 类型的变量强制转化为 void*进行传递（针对不同位数机器，指针对其字数不同，需要 int 转化为 long 在转指针，否则可能会发生警告），在线程处理回调函数中，直接将 void*数据转

化为 int 类型即可，本质上是在传递变量 a 的值。

上述两种方法均可得到所要的值，但是要注意其本质，一个为地址传递，一个为值的传递。当变量发生改变时候，传递地址后，该地址所对应的变量也会发生改变，但传入变量值的时候，即使地址指针所指的变量发生变化，但传入的为变量值，不会受到指针的指向的影响，实际项目中切记两者之间的区别。具体说明见例程 4。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text4.c
```

测试例程 4：（Phtread_txex4.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <errno.h>
5
6   void *fun1(void *arg)
7   {
8       while(1){
9
10          printf("%s:arg = %d Addr = %p\n",__FUNCTION__,*(int *)arg,arg);
11          sleep(1);
12      }
13  }
14
15  void *fun2(void *arg)
16  {
17      while(1){
18
19          printf("%s:arg = %d Addr = %p\n",__FUNCTION__,(int)(long)arg,arg);
20          sleep(1);
21      }
22  }
23
24  int main()
25  {
26
27      pthread_t tid1,tid2;
28      int a = 50;
29      int ret = pthread_create(&tid1,NULL,fun1,(void *)&a);
30      if(ret != 0){
31          perror("pthread_create");
32          return -1;
33      }
34      sleep(1);
35      ret = pthread_create(&tid2,NULL,fun2,(void *)(long)a);
36      if(ret != 0){
37          perror("pthread_create");
38          return -1;
39      }
40      while(1){
41          a++;
```

```
42          sleep(1);
43          printf("%s:a = %d Add = %p \n",__FUNCTION__,a,&a);
44      }
45      return 0;
46  }
47
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text4.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
fun1:arg = 50 Addr = 0x7ffdae267370
fun1:arg = 51 Addr = 0x7ffdae267370
fun2:arg = 50 Addr = 0x32
fun2:arg = 50 Addr = 0x32
main:a = 51 Add = 0x7ffdae267370
fun1:arg = 52 Addr = 0x7ffdae267370
main:a = 52 Add = 0x7ffdae267370
fun1:arg = 53 Addr = 0x7ffdae267370
fun2:arg = 50 Addr = 0x32
fun2:arg = 50 Addr = 0x32
main:a = 53 Add = 0x7ffdae267370
fun1:arg = 54 Addr = 0x7ffdae267370
^C
```

图 9.5 例程 4 运行结果

上述例程讲述了如何向线程传递一个参数，在处理实际项目中，往往会遇到传递多个参数的问题，我们可以通过结构体来进行传递，解决此问题。

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text5.c
```

测试例程 5：（Phtread_txex5.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <errno.h>
6
7   struct Stu{
8       int Id;
9       char Name[32];
10      float Mark;
11  };
12
13  void *fun1(void *arg)
14  {
15      struct Stu *tmp = (struct Stu *)arg;
16      printf("%s:Id = %d Name = %s Mark = %.2f\n",__FUNCTION__,tmp->Id,tmp->Name,tm
p->Mark);
17
18  }
19
20  int main()
21  {
22
23      pthread_t tid1,tid2;
24      struct Stu stu;
25      stu.Id = 10000;
26      strcpy(stu.Name,"ZhangSan");
27      stu.Mark = 94.6;
28
```

```
29      int ret = pthread_create(&tid1,NULL,fun1,(void *)&stu);
30      if(ret != 0){
31          perror("pthread_create");
32          return -1;
33      }
34      printf("%s:Id = %d Name = %s Mark = %.2f\n",__FUNCTION__,stu.Id,stu.Name,stu.
Mark);
35      sleep(1);
36      return 0;
37  }
38
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text5.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
main:Id = 10000 Name = ZhangSan Mark = 94.60
fun1:Id = 10000 Name = ZhangSan Mark = 94.60
```

图 9.6 例程 5 运行结果

9.1.6 线程的退出与回收

线程的退出情况有三种：第一种是进程结束，进程中所有的线程也会随之结束。第二种是通过函数 pthread_exit 来主动的退出线程。第三种被其他线程调用 pthread_cancel 来被动退出。

当线程结束后，主线程可以通过函数 pthread_join/pthread_tryjoin_np 来回收线程的资源，并且获得线程结束后需要返回的数据。

1 线程主动退出

pthread_exit 函数原型如下：

```
线程主动退出
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_exit 函数为线程退出函数，在退出时候可以传递一个 void*类型的数据带给主线程，若选择不传出数据，可将参数填充为 NULL。

2 线程被动退出

pthread_cancel 函数原型如下：

```
线程被动退出，其他线程使用该函数让另一个线程退出
#include <pthread.h>
int pthread_cancel(pthread_t thread);
成功：返回 0
```

该函数传入一个 tid 号，会强制退出该 tid 所指向的线程，若成功执行会返回 0。

3 线程资源回收(阻塞方式)

pthread_join 函数原型如下：

```
线程资源回收（阻塞）
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

该函数为线程回收函数，默认状态为阻塞状态，直到成功回收线程后才返

回。第一个参数为要回收线程的 tid 号，第二个参数为线程回收后接受线程传出的数据。

4 线程资源回收(非阻塞方式)

pthread_tryjoin_np 函数原型如下：

```
线程资源回收（非阻塞）
#define _GNU_SOURCE
#include <pthread.h>
int pthread_tryjoin_np(pthread_t thread, void **retval);
```

该函数为非阻塞模式回收函数，通过返回值判断是否回收掉线程，成功回收则返回 0，其余参数与 pthread_join 一致。

5 程序示例 1

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text6.c
```

测试例程 6：（Phtread_txex6.c）

```
1   #include <pthread.h>
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <errno.h>
5
6   void *fun1(void *arg)
7   {
8       static int tmp = 0;//必须要 static 修饰，否则 pthread_join 无法获取到正确值
9       //int tmp = 0;
10      tmp = *(int *)arg;
11      tmp+=100;
12      printf("%s:Addr = %p tmp = %d\n",__FUNCTION__,&tmp,tmp);
13      pthread_exit((void *)&tmp);//将变量 tmp 取地址转化为 void*类型传出
14  }
15
16
17  int main()
18  {
19
20      pthread_t tid1;
21      int a = 50;
22      void *Tmp = NULL;//因 pthread_join 第二个参数为 void**类型
23      int ret = pthread_create(&tid1,NULL,fun1,(void *)&a);
24      if(ret != 0){
25          perror("pthread_create");
26          return -1;
27      }
28      pthread_join(tid1,&Tmp);
29      printf("%s:Addr = %p Val = %d\n",__FUNCTION__,Tmp,*(int *)Tmp);
30      return 0;
31  }
```

运行结果：

图 9.7 例程 6 运行结果

上述例程先通过 23 行将变量以地址的形式传入线程，在线程中做出了自加 100 的操作，当线程退出的时候通过线程传参，用 void*类型的数据通过 pthread_join 接受。此例程去掉了之前加入的 sleep 函数，原因是 pthread_join 函数具备阻塞的特性，直至成功收回掉线程后才会冲破阻塞，因此不需要靠考虑主线程会执行到 30 行结束进程的情况。特别要说明的是例程第 8 行，当变量从线程传出的时候，需要加 static 修饰，对生命周期做出延续，否则无法传出正确的变量值。

6 程序示例 2

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text7.c
```

测试例程 7：（Phtread_txex7.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6
7   void *fun(void *arg)
8   {
9       printf("Pthread:%d Come !\n",(int )(long)arg+1);
10      pthread_exit(arg);
11  }
12
13
14  int main()
15  {
16      int ret,i,flag = 0;
17      void *Tmp = NULL;
18      pthread_t tid[3];
19      for(i = 0;i < 3;i++){
20          ret = pthread_create(&tid[i],NULL,fun,(void *)(long)i);
21          if(ret != 0){
22              perror("pthread_create");
23              return -1;
24          }
25      }
26      while(1){//通过非阻塞方式收回线程，每次成功回收一个线程变量自增，直至 3 个线程全数回
收
27          for(i = 0;i <3;i++){
28              if(pthread_tryjoin_np(tid[i],&Tmp) == 0){
29                  printf("Pthread : %d exit !\n",(int )(long )Tmp+1);
30                  flag++;
```

```
31              }
32          }
33          if(flag >= 3) break;
34      }
35      return 0;
36  }
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text7.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
Pthread:1 Come !
Pthread : 1 exit !
Pthread:2 Come !
Pthread : 2 exit !
Pthread:3 Come !
Pthread : 3 exit !
```

图 9.8 例程 7 运行结果

例程 7 展示了如何使用非阻塞方式来回收线程，此外也展示了多个线程可以指向同一个回调函数的情况。例程 6 通过阻塞方式回收线程几乎规定了线程回收的顺序，若最先回收的线程未退出，则一直会被阻塞，导致后续先退出的线程无法及时的回收。

通过函数 pthread_tryjoin_np，使用非阻塞回收，线程可以根据退出先后顺序自由的进行资源的回收。

7 程序示例 3

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text8.c
```

测试例程 8：（Phtread_txex8.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6
7   void *fun1(void *arg)
8   {
9       printf("Pthread:1 come!\n");
10      while(1){
11          sleep(1);
12      }
13  }
14
15  void *fun2(void *arg)
16  {
17      printf("Pthread:2 come!\n");
18      pthread_cancel((pthread_t )(long)arg);//杀死线程 1，使之强制退出
19      pthread_exit(NULL);
20  }
21
22  int main()
23  {
24      int ret,i,flag = 0;
25      void *Tmp = NULL;
```

```
26      pthread_t tid[2];
27      ret = pthread_create(&tid[0],NULL,fun1,NULL);
28      if(ret != 0){
29          perror("pthread_create");
30          return -1;
31      }
32      sleep(1);
33      ret = pthread_create(&tid[1],NULL,fun2,(void *)tid[0]);//传输线程 1 的线程号
34      if(ret != 0){
35          perror("pthread_create");
36          return -1;
37      }
38      while(1){//通过非阻塞方式收回线程，每次成功回收一个线程变量自增，直至 2 个线程全数回
收

39          for(i = 0;i <2;i++){
40              if(pthread_tryjoin_np(tid[i],NULL) == 0){
41                  printf("Pthread : %d exit !\n",i+1);
42                  flag++;
43              }
44          }
45          if(flag >= 2) break;
46      }
47      return 0;
48  }
49
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text8.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
Pthread:1 come!
Pthread:2 come!
Pthread : 2 exit !
Pthread : 1 exit !
```

图 9.9 例程 8 运行结果

例程 8 展示了如何利用 pthread_cancel 函数主动的将某个线程结束。27 行与 33 行创建了线程，将第一个线程的线程号传参形式传入了第二个线程。第一个的线程执行死循环睡眠逻辑，理论上除非进程结束，其永远不会结束，但在第二个线程中调用了 pthread_cancel 函数，相当于向该线程发送一个退出的指令，导致线程被退出，最终资源被非阻塞回收掉。此例程要注意第 32 行的 sleep 函数，一定要确保线程 1 先执行，因线程是无序执行，故加入该睡眠函数控制顺序，在本章后续，会讲解通过加锁、信号量等手段来合理的控制线程的临界资源访问与线程执行顺序控制。

9.2 线程的控制

9.2.1 多线程编临界资源访问

当线程在运行过程中，去操作公共资源，如全局变量的时候，可能会发生彼此“矛盾”现象。例如线程 1 企图想让变量自增，而线程 2 企图想要变量自减，两个线程存在互相竞争的关系导致变量永远处于一个“平衡状态”，两个线程互

相竞争，线程 1 得到执行权后将变量自加，当线程 2 得到执行权后将变量自减，变量似乎永远在某个范围内浮动，无法到达期望数值，如例程 9 所示。

使用 GIT 下载所有源码后，本节源码位于如下目录：

01_all_series_quickstart
04_嵌入式 Linux 应用开发基础知识
source\13_thread\01_文档配套源码\Pthread_Text9.c

测试例程 9：（Phtread_txex9.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6
7
8   int Num = 0;
9
10  void *fun1(void *arg)
11  {
12      while(Num < 3){
13          Num++;
14          printf("%s:Num = %d\n",__FUNCTION__,Num);
15          sleep(1);
16      }
17      pthread_exit(NULL);
18  }
19
20  void *fun2(void *arg)
21  {
22      while(Num > -3){
23          Num--;
24          printf("%s:Num = %d\n",__FUNCTION__,Num);
25          sleep(1);
26      }
27      pthread_exit(NULL);
28  }
29
30  int main()
31  {
32      int ret;
33      pthread_t tid1,tid2;
34      ret = pthread_create(&tid1,NULL,fun1,NULL);
35      if(ret != 0){
36          perror("pthread_create");
37          return -1;
38      }
39      ret = pthread_create(&tid2,NULL,fun2,NULL);
40      if(ret != 0){
41          perror("pthread_create");
42          return -1;
43      }
44      pthread_join(tid1,NULL);
45      pthread_join(tid2,NULL);
46      return 0;
47  }
```

运行结果：

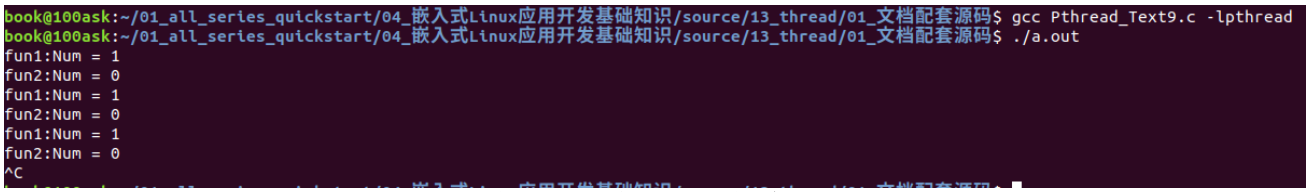


图 9.10 例程 9 运行结果

为了解决上述对临界资源的竞争问题，pthread 线程引出了互斥锁来解决临界资源访问。通过对临界资源加锁来保护资源只被单个线程操作，待操作结束后解锁，其余线程才可获得操作权。

9.2.2 互斥锁 API 简述

多个线程都要访问某个临界资源，比如某个全局变量时，需要互斥地访问：我访问时，你不能访问。

可以使用以下函数进行互斥操作。

1 初始化互斥量

函数原型如下：

```
int pthread_mutex_init(phtread_mutex_t *mutex,
                       const pthread_mutexattr_t *restrict attr);
```

该函数初始化一个互斥量，第一个参数是改互斥量指针，第二个参数为控制互斥量的属性，一般为 NULL。当函数成功后会返回 0，代表初始化互斥量成功。

当然初始化互斥量也可以调用宏来快速初始化，代码如下：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITALIZER;
```

2 互斥量加锁/解锁

函数原型如下：

```
互斥量加锁（阻塞）/解锁
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
成功：返回 0
```

lock 函数与 unlock 函数分别为加锁解锁函数，只需要传入已经初始化好的 pthread_mutex_t 互斥量指针。成功后会返回 0。

当某一个线程获得了执行权后，执行 lock 函数，一旦加锁成功后，其余线程遇到 lock 函数时候会发生阻塞，直至获取资源的线程执行 unlock 函数后。unlock 函数会唤醒其他正在等待互斥量的线程。

特别注意的是，当获取 lock 之后，必须在逻辑处理结束后执行 unlock，否则会发生死锁现象！导致其余线程一直处于阻塞状态，无法执行下去。在使用互斥量的时候，尤其要注意使用 pthread_cancel 函数，防止发生死锁现象！

3 互斥量加锁(非阻塞方式)

函数原型如下：

```
互斥量加锁（非阻塞）
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

该函数同样也是一个线程加锁函数，但该函数是非阻塞模式通过返回值来判断是否加锁成功，用法与上述阻塞加锁函数一致。

4 互斥量加锁(非祖师方式)

函数原型如下：

```
互斥量销毁
#include <pthread.h>
 int pthread_mutex_destory(pthread_mutex_t *mutex);
成功：返回 0
```

该函数是用于销毁互斥量的，传入互斥量的指针，就可以完成互斥量的销毁，成功返回 0。

5 程序示例

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text10.c
```

测试例程 10：（Phtread_txex10.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6
7   pthread_mutex_t mutex;//互斥量变量 一般申请全局变量
8
9   int Num = 0;//公共临界变量
10
11  void *fun1(void *arg)
12  {
13      pthread_mutex_lock(&mutex);//加锁 若有线程获得锁，则会阻塞
14      while(Num < 3){
15          Num++;
16          printf("%s:Num = %d\n",__FUNCTION__,Num);
17          sleep(1);
18      }
19      pthread_mutex_unlock(&mutex);//解锁
20      pthread_exit(NULL);//线程退出 pthread_join 会回收资源
21  }
22
23  void *fun2(void *arg)
24  {
25      pthread_mutex_lock(&mutex);//加锁 若有线程获得锁，则会阻塞
26      while(Num > -3){
27          Num--;
28          printf("%s:Num = %d\n",__FUNCTION__,Num);
29          sleep(1);
30      }
```

```
31      pthread_mutex_unlock(&mutex);//解锁
32      pthread_exit(NULL);//线程退出 pthread_join 会回收资源
33  }
34
35  int main()
36  {
37      int ret;
38      pthread_t tid1,tid2;
39      ret = pthread_mutex_init(&mutex,NULL);//初始化互斥量
40      if(ret != 0){
41          perror("pthread_mutex_init");
42          return -1;
43      }
44      ret = pthread_create(&tid1,NULL,fun1,NULL);//创建线程 1
45      if(ret != 0){
46          perror("pthread_create");
47          return -1;
48      }
49      ret = pthread_create(&tid2,NULL,fun2,NULL);//创建线程 2
50      if(ret != 0){
51          perror("pthread_create");
52          return -1;
53      }
54      pthread_join(tid1,NULL);//阻塞回收线程 1
55      pthread_join(tid2,NULL);//阻塞回收线程 2
56      pthread_mutex_destroy(&mutex);//销毁互斥量
57      return 0;
58  }
59
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text10.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
fun1:Num = 1
fun1:Num = 2
fun1:Num = 3
fun2:Num = 2
fun2:Num = 1
fun2:Num = 0
fun2:Num = -1
fun2:Num = -2
fun2:Num = -3
^C
```

图 9.11 例程 10 运行结果

上述例程通过加入互斥量，保证了临界变量某一时刻只被某一线程控制，实现了临界资源的控制。需要说明的是，线程加锁在循环内与循环外的情况。本历程在进入 while 循环前进行了加锁操作，在循环结束后进行的解锁操作，如果将加锁解锁全部放入 while 循环内，作为单核的机器，执行结果无异，当有多核机器执行代码时，可能会发生“抢锁”现象，这取决于操作系统底层的实现。

9.2.3 多线程编执行顺序控制

解决了临界资源的访问，但似乎对线程的执行顺序无法得到控制，因线程都是无序执行，之前采用 sleep 强行延时的方法勉强可以控制执行顺序，但此方法在实际项目情况往往是不可取的，其仅仅可解决线程创建的顺序，当创建

之后执行的顺序又不会受到控制，于是便引入了信号量的概念，解决线程执行顺序。

例程 11 将展示线程的执行的随机性。

使用 GIT 下载所有源码后，本节源码位于如下目录：

01_all_series_quickstart
04_嵌入式 Linux 应用开发基础知识
source\13_thread\01_文档配套源码\Pthread_Text11.c

测试例程 11：（Phtread_txex11.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6
7   void *fun1(void *arg)
8   {
9       printf("%s:Pthread Come!\n",__FUNCTION__);
10      pthread_exit(NULL);
11  }
12
13  void *fun2(void *arg)
14  {
15      printf("%s:Pthread Come!\n",__FUNCTION__);
16      pthread_exit(NULL);
17  }
18
19  void *fun3(void *arg)
20  {
21      printf("%s:Pthread Come!\n",__FUNCTION__);
22      pthread_exit(NULL);
23  }
24
25  int main()
26  {
27      int ret;
28      pthread_t tid1,tid2,tid3;
29      ret = pthread_create(&tid1,NULL,fun1,NULL);
30      if(ret != 0){
31          perror("pthread_create");
32          return -1;
33      }
34      ret = pthread_create(&tid2,NULL,fun2,NULL);
35      if(ret != 0){
36          perror("pthread_create");
37          return -1;
38      }
39      ret = pthread_create(&tid3,NULL,fun3,NULL);
40      if(ret != 0){
41          perror("pthread_create");
42          return -1;
43      }
44      pthread_join(tid1,NULL);
45      pthread_join(tid2,NULL);
```

```
46      pthread_join(tid3,NULL);
47      return 0;
48  }
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ gcc Pthread_Text11.c -lpthread
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
```

图 9.12 例程 11 运行结果

通过上述例程可以发现，多次执行该函数其次序是无序的，线程之间的竞争无法控制，通过使用信号量来使得线程顺序为可控的。

9.2.4 信号量 API 简述

注意：信号量跟互斥量不一样，互斥量用来防止多个线程同时访问某个临界资源。信号量起通知作用，线程 A 在等待某件事，线程 B 完成了这件事后就可以给线程 A 发信号。

1 初始化信号量

函数原型如下：

```
int sem_init(sem_t *sem,int pshared,unsigned int value);
```

- 该函数可以初始化一个信号量，第一个参数传入 sem_t 类型指针；
- 第二个参数传入 0 代表线程控制，否则为进程控制；
- 第三个参数表示信号量的初始值，0 代表阻塞，1 代表运行。
- 待初始化结束信号量后，若执行成功会返回 0。

2 2．信号量 P/V 操作

函数原型如下：

```
#include <pthread.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
成功：返回 0
```

- sem_wait 函数作用为检测指定信号量是否有资源可用，若无资源可用会阻塞等待，若有资源可用会自动的执行“sem-1”的操作。所谓的“sem-1”是与上述初始化函数中第三个参数值一致，成功执行会返回 0。
- sem_post 函数会释放指定信号量的资源，执行“sem+1”操作。

通过以上 2 个函数可以完成所谓的 PV 操作，即信号量的申请与释放，完成对线程执行顺序的控制。

3 信号量申请(非阻塞方式)

函数原型如下：

```
#include <pthread.h>
int sem_trywait(sem_t *sem);
成功：返回 0
```

此函数是信号量申请资源的非阻塞函数，功能与 sem_wait 一致，唯一区

别在于此函数为非阻塞。

4 信号量销毁

函数原型如下：

```
#include <pthread.h>
int sem_destory(sem_t *sem);
成功：返回 0
```

该函数为信号量销毁函数，执行过后可将信号量进行销毁。

5 程序示例

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\
source\13_thread\01_文档配套源码\Pthread_Text12.c
```

测试例程 12：（Phtread_txex12.c）

```
1   #define _GNU_SOURCE
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <errno.h>
6   #include <semaphore.h>
7
8   sem_t sem1,sem2,sem3;//申请的三个信号量变量
9
10  void *fun1(void *arg)
11  {
12      sem_wait(&sem1);//因 sem1 本身有资源，所以不被阻塞 获取后 sem1-1 下次会会阻塞
13      printf("%s:Pthread Come!\n",__FUNCTION__);
14      sem_post(&sem2);// 使得 sem2 获取到资源
15      pthread_exit(NULL);
16  }
17
18  void *fun2(void *arg)
19  {
20      sem_wait(&sem2);//因 sem2 在初始化时无资源会被阻塞，直至 14 行代码执行 不被阻塞 sem
2-1 下次会阻塞
21      printf("%s:Pthread Come!\n",__FUNCTION__);
22      sem_post(&sem3);// 使得 sem3 获取到资源
23      pthread_exit(NULL);
24  }
25
26  void *fun3(void *arg)
27  {
28      sem_wait(&sem3);//因 sem3 在初始化时无资源会被阻塞，直至 22 行代码执行 不被阻塞 sem
3-1 下次会阻塞
29      printf("%s:Pthread Come!\n",__FUNCTION__);
30      sem_post(&sem1);// 使得 sem1 获取到资源
31      pthread_exit(NULL);
32  }
33
34  int main()
35  {
```

```
36      int ret;
37      pthread_t tid1,tid2,tid3;
38      ret = sem_init(&sem1,0,1);  //初始化信号量 1 并且赋予其资源
39      if(ret < 0){
40          perror("sem_init");
41          return -1;
42      }
43      ret = sem_init(&sem2,0,0); //初始化信号量 2 让其阻塞
44      if(ret < 0){
45          perror("sem_init");
46          return -1;
47      }
48      ret = sem_init(&sem3,0,0); //初始化信号 3 让其阻塞
49      if(ret < 0){
50          perror("sem_init");
51          return -1;
52      }
53      ret = pthread_create(&tid1,NULL,fun1,NULL);//创建线程 1
54      if(ret != 0){
55          perror("pthread_create");
56          return -1;
57      }
58      ret = pthread_create(&tid2,NULL,fun2,NULL);//创建线程 2
59      if(ret != 0){
60          perror("pthread_create");
61          return -1;
62      }
63      ret = pthread_create(&tid3,NULL,fun3,NULL);//创建线程 3
64      if(ret != 0){
65          perror("pthread_create");
66          return -1;
67      }
68      /*回收线程资源*/
69      pthread_join(tid1,NULL);
70      pthread_join(tid2,NULL);
71      pthread_join(tid3,NULL);
72
73      /*销毁信号量*/
74      sem_destroy(&sem1);
75      sem_destroy(&sem2);
76      sem_destroy(&sem3);
78      return 0;
79  }
```

运行结果：

```
book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/13_thread/01_文档配套源码$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
```

图 9.13 例程 12 运行结果

该例程加入了信号量，使得线程的执行顺序变为可控的。在初始化信号量时，将信号量 1 填入资源，第一个线程调用 sem_wait 函数可以成功获得信号量，在执行完逻辑后使用 sem_pos 函数来释放。当执行函数 sem_wait 后，会执行 sem 自减操作，使下一次竞争被阻塞，直至通过 sem_pos 被释放。

上述例程因 38 行初始化信号量 1 时候，使其默认获取到资源；第 43、48 行初始化信号量 2、3 时候，使之没有资源。于是在线程处理函数中，每个线程通过 sem_wait 函数来等待资源，发生阻塞。因信号量 1 初始值为有资源，故可以先执行线程 1 的逻辑。待执行完第 12 行 sem_wait 函数，会导致 sem1-1，使得下一次此线程会被阻塞。继而执行至 14 行，通过 sem_post 函数使 sem2 信号量获取资源，从而冲破阻塞执行线程 2 的逻辑...以此类推完成线程的有序控制。

9.2.5 条件变量

参考《Unix_Linux_Windows_OpenMP 多线程编程.pdf》，作者不详。

条件变量时一种同步机制，用来通知其他线程条件满足了。一般是用来通知对方共享数据的状态信息，因此条件变量时结合互斥量来使用的。

1 创建和销毁条件变量

函数原型如下：

```
#include <pthread.h>
// 初始化条件变量
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);//cond_at
tr 通常为 NULL
// 销毁条件变量
int pthread_cond_destroy(pthread_cond_t *cond);
这些函数成功时都返回 0
```

2 等待条件变量

函数原型如下：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

这需要结合互斥量一起使用，示例代码如下：

```
pthread_mutex_lock(&g_tMutex);
// 如果条件不满足则，会 unlock g_tMutex
// 条件满足后被唤醒，会 lock g_tMutex
pthread_cond_wait(&g_tConVar, &g_tMutex);
/* 操作临界资源 */
pthread_mutex_unlock(&g_tMutex);
```

3 通知条件变量

函数原型如下：

```
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_signal 函数只会唤醒一个等待 cond 条件变量的线程，示例代码如下：

```
pthread_cond_signal(&g_tConVar);
```

4 4．程序示例

视频里现场编程。使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
04_嵌入式 Linux 应用开发基础知识\source\13_thread\02_视频配套源码
```

9.3 总结

9.3.1 线程使用流程图

有关多线程的创建流程如图 9.14 所示，首先需要创建线程，一旦线程创建完成后，线程与线程之间会发生竞争执行，抢占时间片来执行线程逻辑。在创建线程时候，可以通过创建线程的第四个参数传入参数，在线程退出时亦可传出参数被线程回收函数所回收，获取到传出的参数。

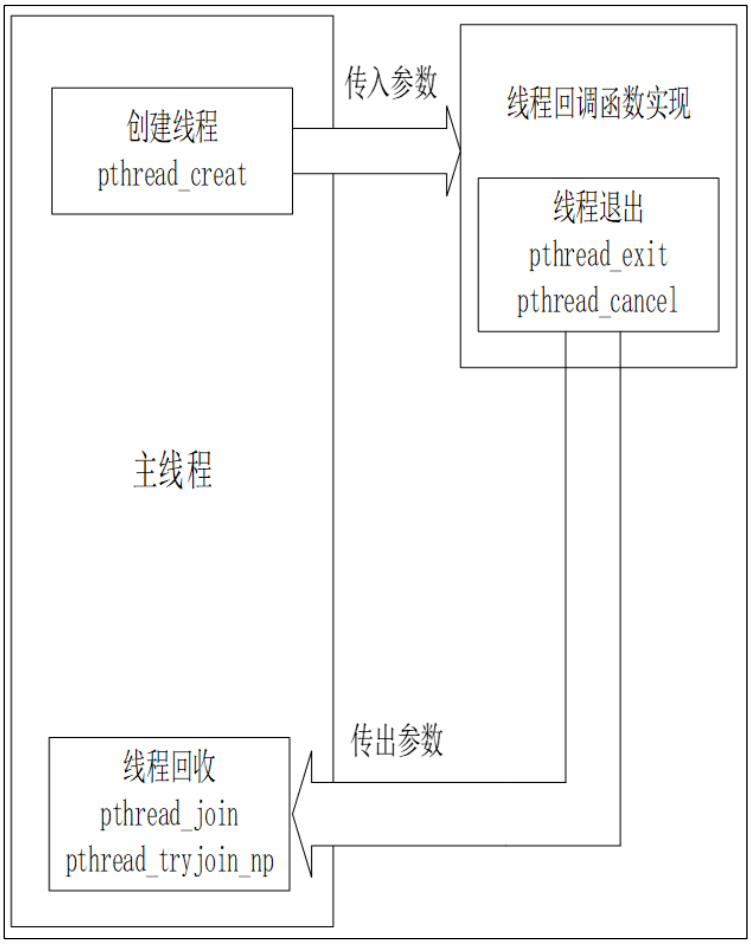


图 9.14 线程使用流程图

9.3.2 互斥量使用流程图

当多个线程出现后，会遇到同时操作临界公共资源的问题，当线程操作公共资源时需要对线程进行保护加锁，防止其与线程在此线程更改变量时同时更改变量，待逻辑执行完毕后再次解锁，使其余线程再度开始竞争。互斥锁创建流程下图所示。

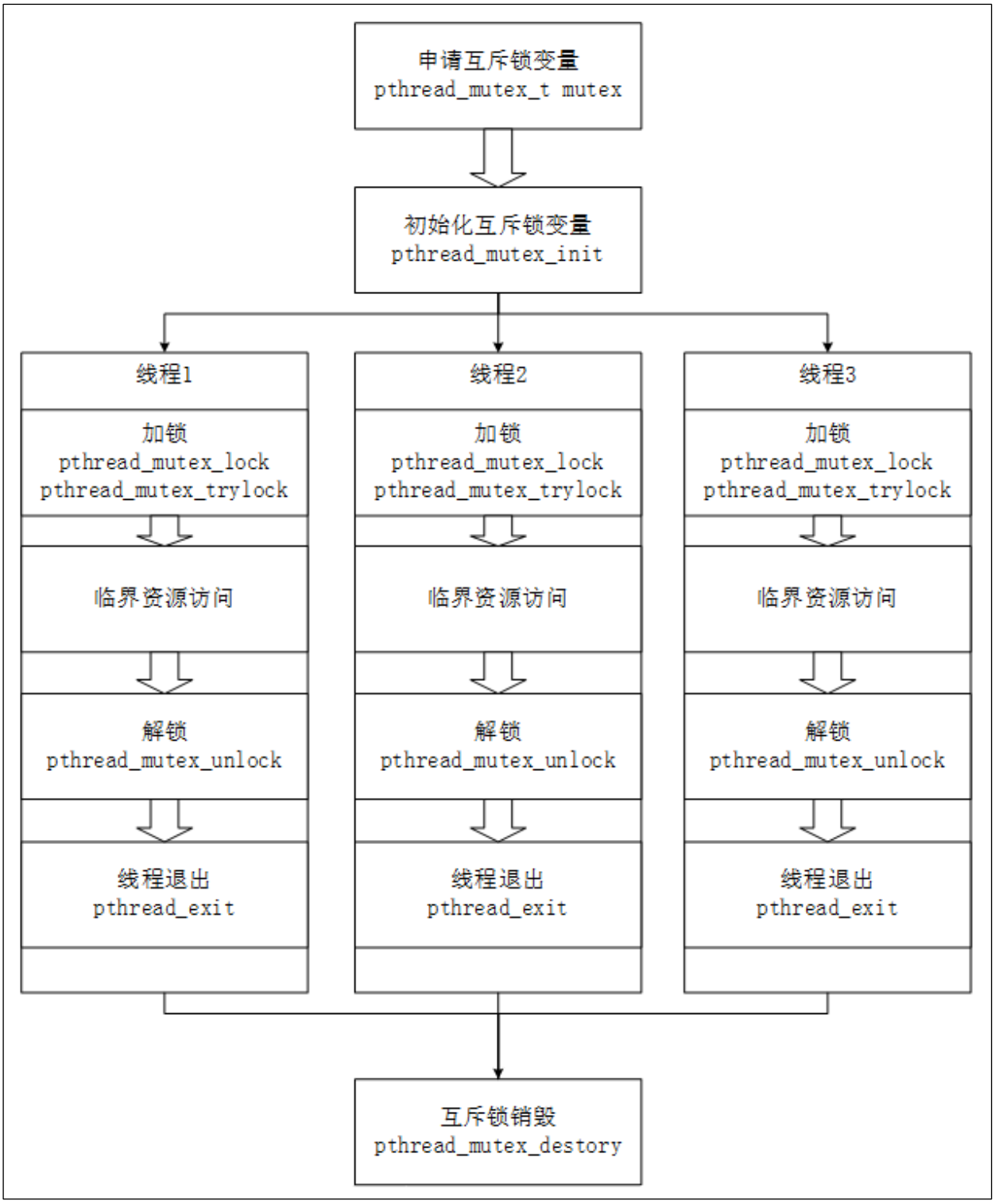


图 9.15 9.3.2 互斥量使用流程图

9.3.3 信号量使用流程图

当多个线程出现后，同时会遇到无序执行的问题。有时候需要对线程的执行顺序做出限定，变引入了信号量，通过 PV 操作来控制线程的执行顺序，如下图所示。

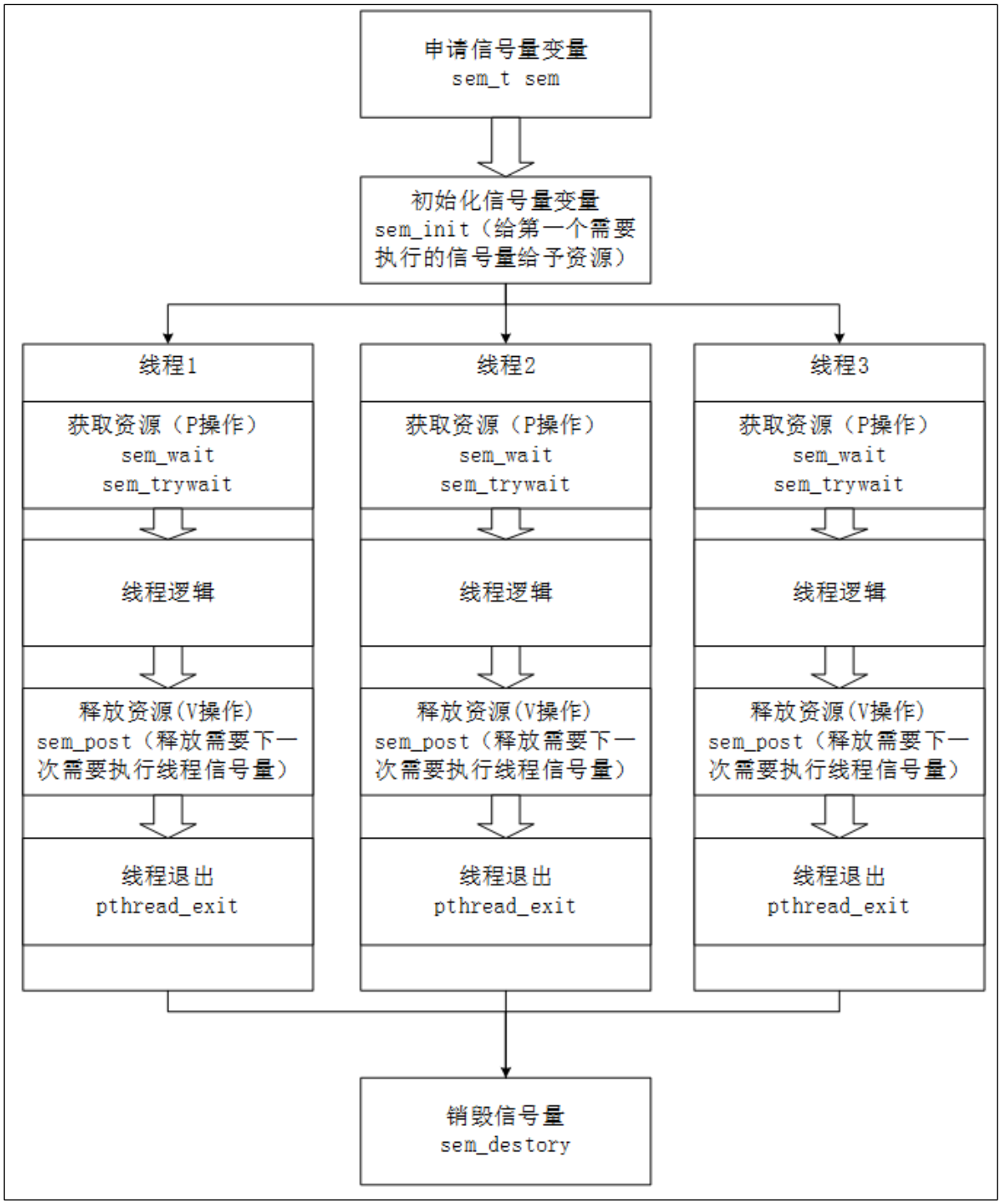


图 9.16 9.3.3 信号量使用流程图

第10章 I2C 应用编程

10.1 I2C 视频介绍

参考资料：I2Ctools
https://mirrors.edge.kernel.org/pub/software/utils/i2c-tools/

10.1.1 I2C 硬件框架

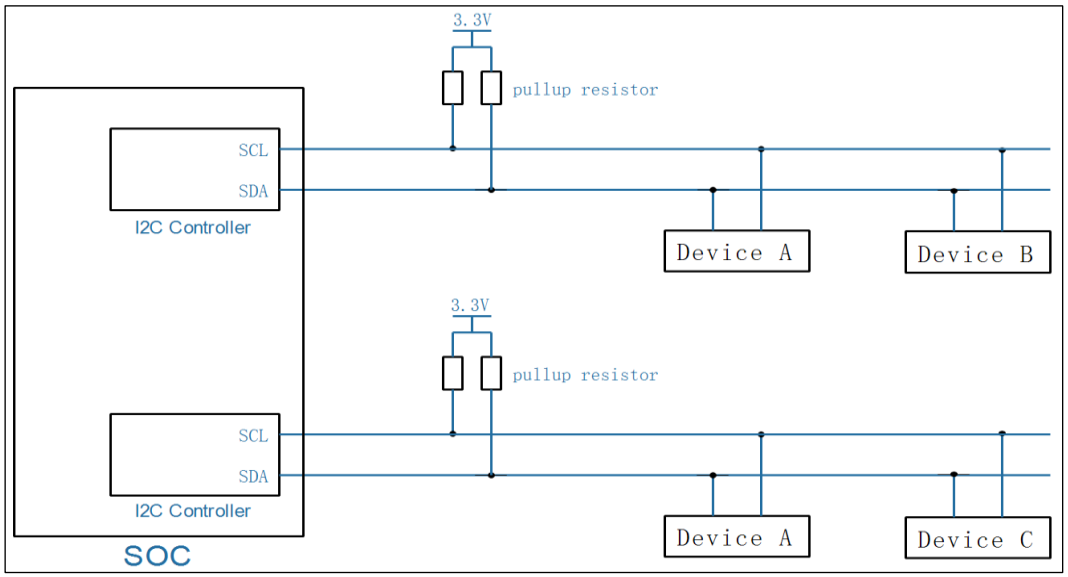


图 10.1 I2C 总线拓扑图

- 在一个芯片(SoC)内部，有一个或多个 I2C 控制器
- 在一个 I2C 控制器上，可以连接一个或多个 I2C 设备
- I2C 总线只需要 2 条线：时钟线 SCL、数据线 SDA
- 在 I2C 总线的 SCL、SDA 线上，都有上拉电阻

10.1.2 I2C 软件框架

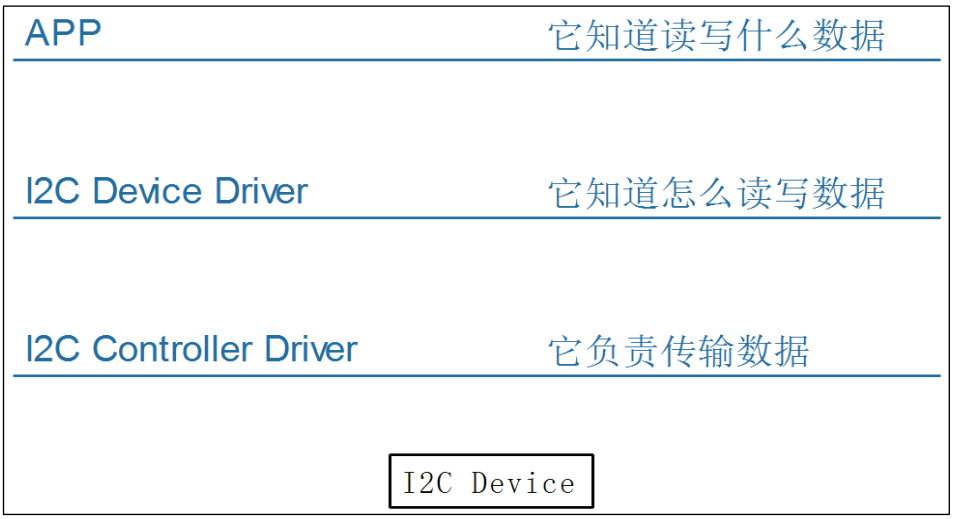


图 10.2 I2C 软件框架

以 I2C 接口的存储设备 AT24C02 为例：

- APP：
  - 提出要求：把字符串"www.100ask.net"写入 AT24C02 地址 16 开始的地方
  - 它是大爷，不关心底层实现的细节
  - 它只需要调用设备驱动程序提供的接口
- AT24C02 驱动：
  - 它知道 AT24C02 要求的地址、数据格式
  - 它知道发出什么信号才能让 AT24C02 执行擦除、烧写工作
  - 它知道怎么判断数据是否烧写成功
  - 它构造好一系列的数据，发给 I2C 控制器
- I2C 控制器驱动
  - 它根据 I2C 协议发出各类信号：I2C 设备地址、I2C 存储地址、数据
  - 它根据 I2C 协议判断

10.1.3 我们讲什么

1 对于 Linux

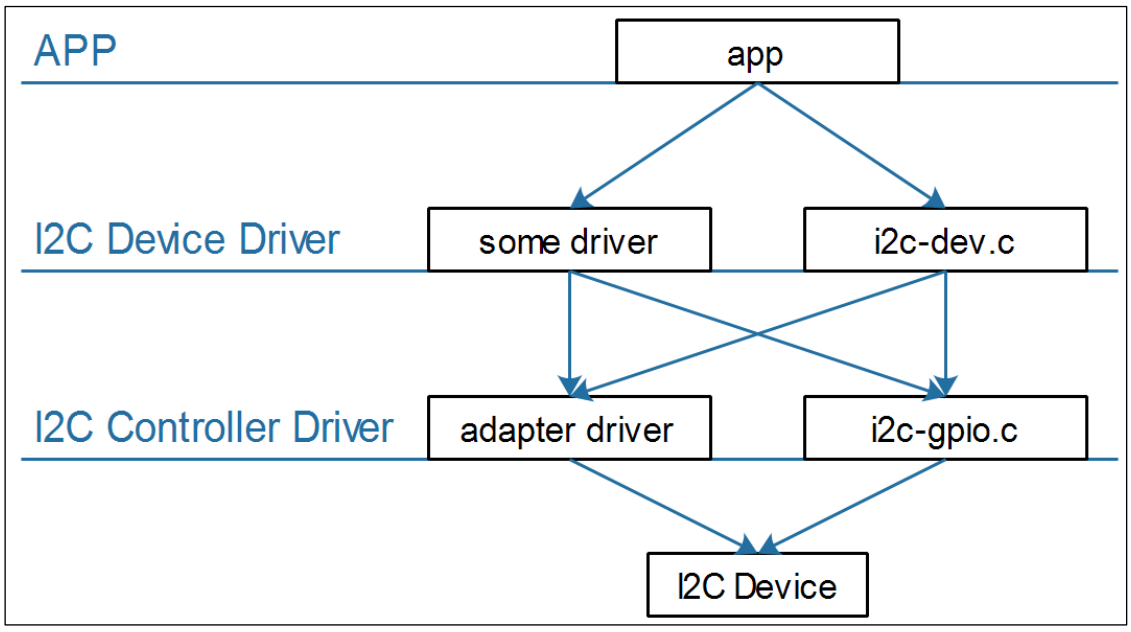


图 10.3 Linux I2C 结构

从上到下：

- 先讲 I2C 协议
- APP 可以通过两类驱动程序访问设备
  - I2C 设备自己的驱动程序
  - 内核自带的 i2c-dev.c 驱动程序，它是 i2c 控制器驱动程序暴露给用户空间的驱动程序(i2c-dev.c)
- I2C Device Driver

- I2C 设备自己的驱动程序
- 内核自带的 i2c-dev.c 驱动程序，它是 i2c 控制器驱动程序暴露给用户空间的驱动程序(i2c-dev.c)

- I2C Controller Driver
  - 芯片 I2C 控制器的驱动程序(称为 adapter)
  - 使用 GPIO 模拟的 I2C 控制器驱动程序(i2c-gpio.c)

2 对于单片机/裸机

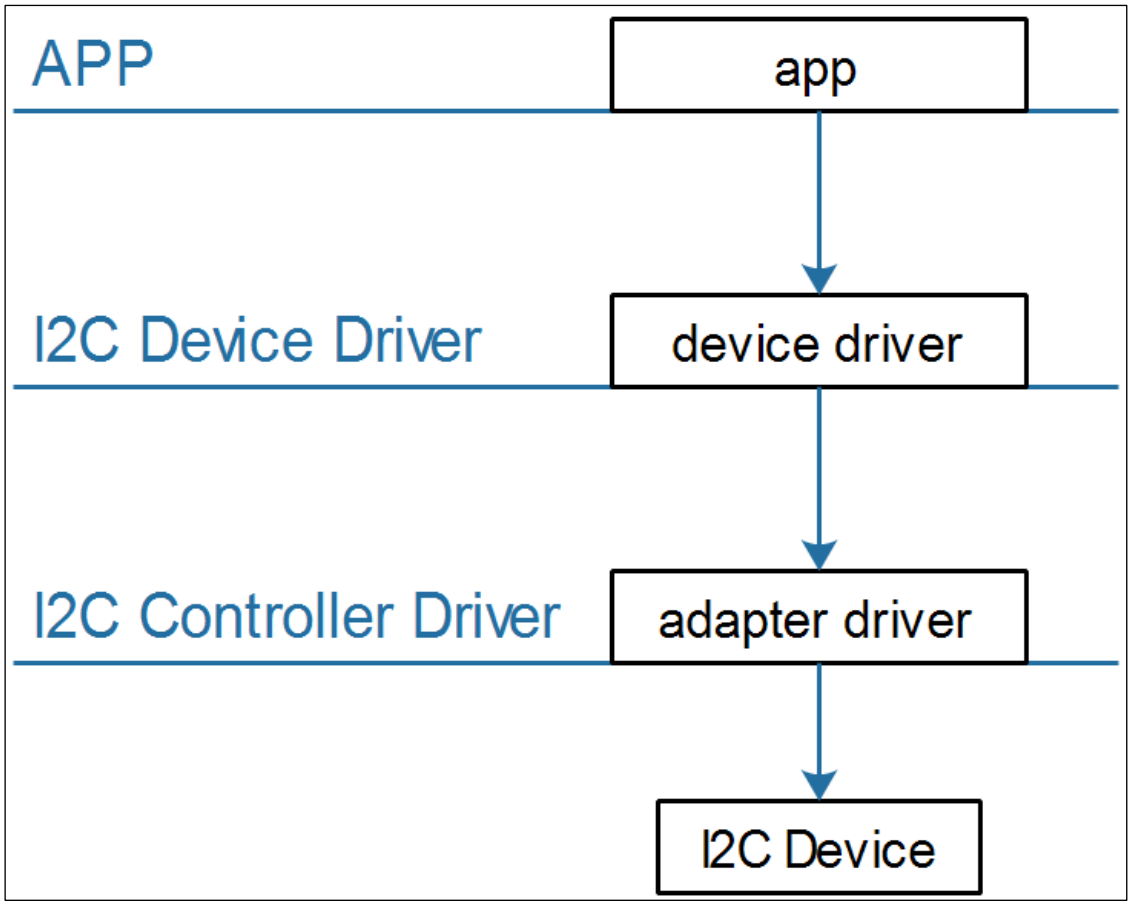


图 10.4 单片机 I2C 结构

从上到下：

- 先讲 I2C 协议
- APP
- I2C Device Driver
- I2C Controller Driver(也被称为 adapter)

10.2 I2C 协议

参考资料：

i2c_spec.pdf

10.2.1 硬件连接

I2C 在硬件上的接法如下所示，主控芯片引出两条线 SCL,SDA 线，在一条 I2C 总线上可以接很多 I2C 设备，我们还会放一个上拉电阻（放一个上拉电阻的原因以后我们再说）。

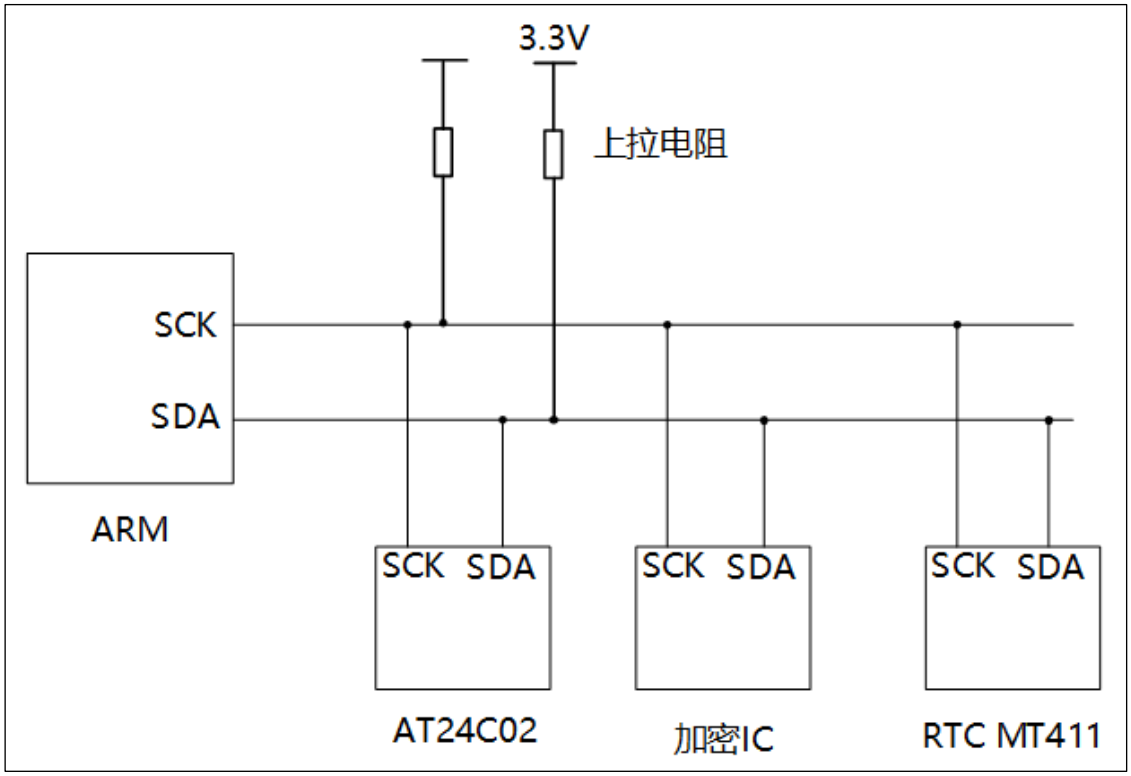


图 10.5 I2C 硬件连接

10.2.2 传输数据类比

怎么通过 I2C 传输数据，我们需要把数据从主设备发送到从设备上去，也需要把数据从从设备传送到主设备上去，数据涉及到双向传输。

举个例子：

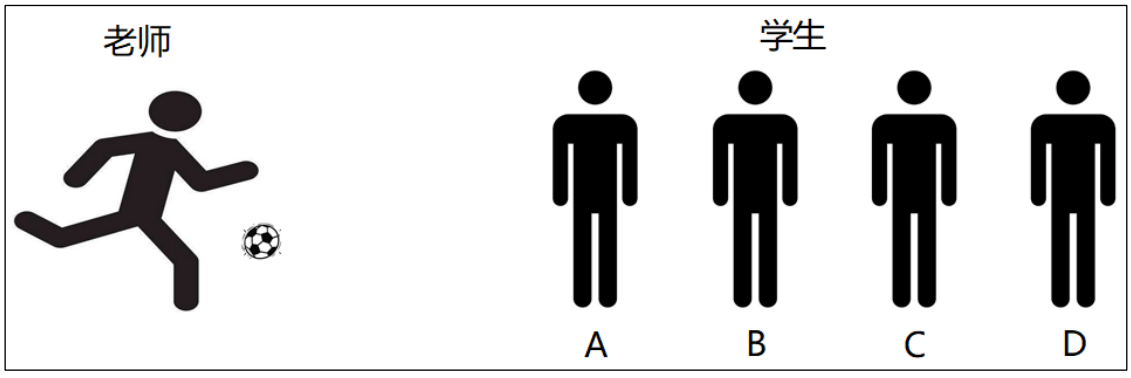


图 10.6 I2C 举例示意图

体育老师：可以把球发给学生，也可以把球从学生中接过来。

- 发球：
  - 老师：开始了(start)

- 老师：A！我要发球给你！(地址/方向)
- 学生 A：到！(回应)
- 老师把球发出去（传输）
- A 收到球之后，应该告诉老师一声（回应）
- 老师：结束（停止）

- 接球：
  - 老师：开始了(start)
  - 老师：B！把球发给我！(地址/方向)
  - 学生 B：到！
  - B 把球发给老师（传输）
  - 老师收到球之后，给 B 说一声，表示收到球了（回应）
  - 老师：结束（停止）

我们就使用这个简单的例子，来解释一下 IIC 的传输协议：

- 老师说开始了，表示开始信号(start)
- 老师提醒某个学生要发球，表示发送地址和方向(address/read/write)
- 老师发球/接球，表示数据的传输
- 收到球要回应：回应信号(ACK)
- 老师说结束，表示 IIC 传输结束(P)

10.2.3 IIC 传输数据的格式

1 写操作

流程如下：

- 主芯片要发出一个 start 信号
- 然后发出一个设备地址(用来确定是往哪一个芯片写数据)，方向(读/写，0 表示写，1 表示读)
- 从设备回应(用来确定这个设备是否存在)，然后就可以传输数据
- 主设备发送一个字节数据给从设备，并等待回应
- 每传输一字节数据，接收方要有一个回应信号（确定数据是否接受完成)，然后再传输下一个数据。
- 数据发送完之后，主芯片就会发送一个停止信号。
- 下图：白色背景表示"主→从"，灰色背景表示"从→主"

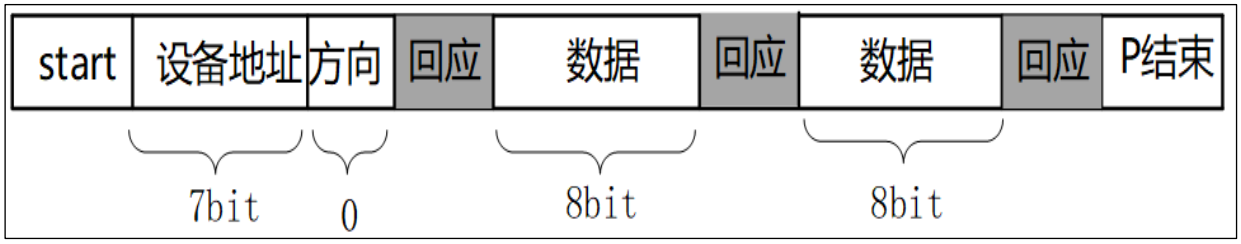


图 10.7 I2C 写传输格式

2 读操作

流程如下：

- 主芯片要发出一个 start 信号
- 然后发出一个设备地址(用来确定是往哪一个芯片写数据)，方向(读/写，0 表示写，1 表示读)
- 从设备回应(用来确定这个设备是否存在)，然后就可以传输数据
- 从设备发送一个字节数据给主设备，并等待回应
- 每传输一字节数据，接收方要有一个回应信号（确定数据是否接受完成)，然后再传输下一个数据。
- 数据发送完之后，主芯片就会发送一个停止信号。

下图：白色背景表示"主→从"，灰色背景表示"从→主"

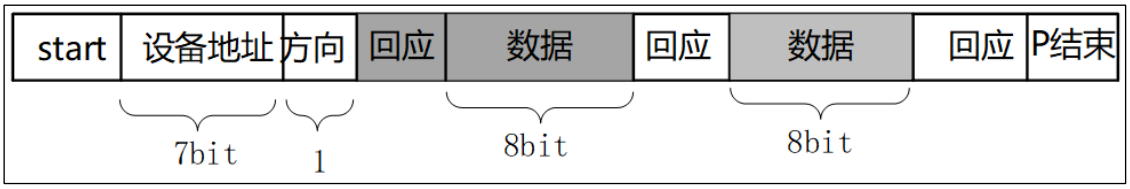


图 10.8 I2C 读传输格式

3 I2C 信号

I2C 协议中数据传输的单位是字节，也就是 8 位。但是要用到 9 个时钟：前面 8 个时钟用来传输 8 数据，第 9 个时钟用来传输回应信号。传输时，先传输最高位(MSB)。

- 开始信号（S)：SCL 为高电平时，SDA 山高电平向低电平跳变，开始传送数据。
- 结束信号（P)：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。
- 响应信号(ACK)：接收器在接收到 8 位数据后，在第 9 个时钟周期，拉低 SDA
- SDA 上传输的数据必须在 SCL 为高电平期间保持稳定，SDA 上的数据只能在 SCL 为低电平期间变化

I2C 协议信号如下：

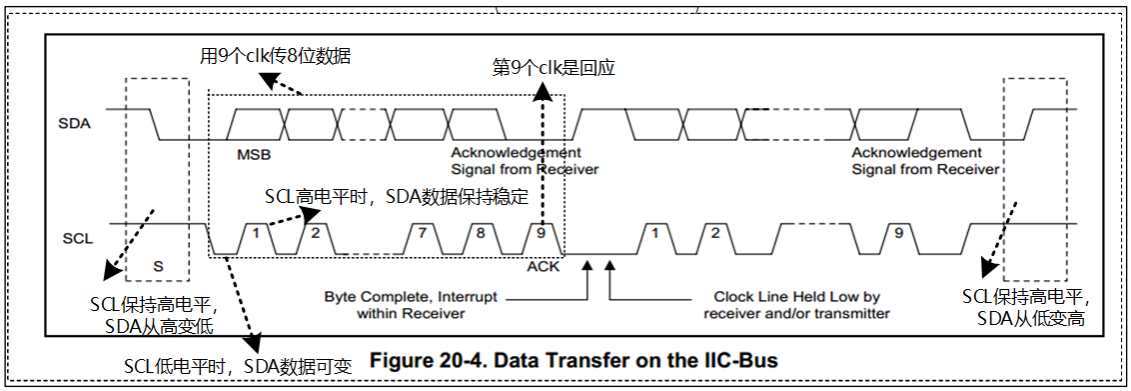


图 10.9 I2C 协议信号

4 协议细节

- 如何在 SDA 上实现双向传输？

- 主芯片通过一根 SDA 线既可以把数据发给从设备，也可以从 SDA 上读取数据，连接 SDA 线的引脚里面必然有两个引脚（发送引脚/接受引脚）。

- 主、从设备都可以通过 SDA 发送数据，肯定不能同时发送数据，怎么错开时间？在 9 个时钟里：
  - 前 8 个时钟由主设备发送数据的话，第 9 个时钟就由从设备发送数据；
  - 前 8 个时钟由从设备发送数据的话，第 9 个时钟就由主设备发送数据。
- 双方设备中，某个设备发送数据时，另一方怎样才能不影响 SDA 上的数据？
  - 设备的 SDA 中有一个三极管，使用开极/开漏电路(三极管是开极，CMOS 管是开漏，作用一样)，如下图：

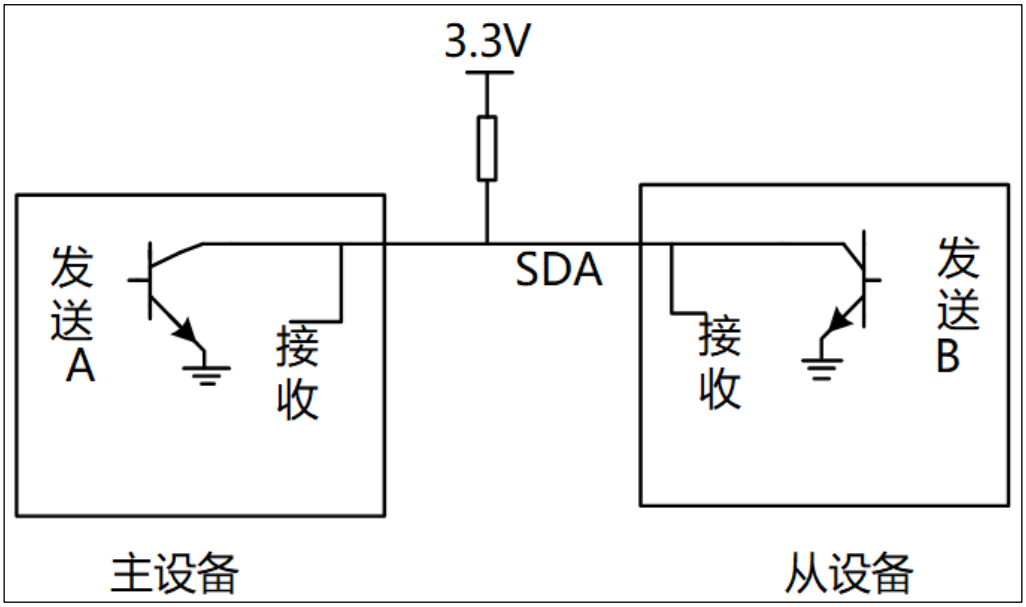


图 10.10 I2C 上拉/开漏电路

真值表如下：

A	B	SDA
0	0	1(由上拉电阻决定)
0	1	0
1	0	0
1	1	0

图 10.11 开漏真值表

从真值表和电路图我们可以知道：

- 当某一个芯片不想影响 SDA 线时，那就不驱动这个三极管
- 想让 SDA 输出高电平，双方都不驱动三极管(SDA 通过上拉电阻变为高电平)
- 想让 SDA 输出低电平，就驱动三极管

从下面的例子可以看看数据是怎么传的（实现双向传输）。

举例：主设备发送（8bit）给从设备

- 前 8 个 clk

    - 从设备不要影响 SDA，从设备不驱动三极管
    - 主设备决定数据，主设备要发送 1 时不驱动三极管，要发送 0 时驱动三极管
- 第 9 个 clk，由从设备决定数据
    - 主设备不驱动三极管
    - 从设备决定数据，要发出回应信号的话，就驱动三极管让 SDA 变为 0

从这里也可以知道 ACK 信号是低电平

从上面的例子，就可以知道怎样在一条线上实现双向传输，这就是 SDA 上要使用上拉电阻的原因。

为何 SCL 也要使用上拉电阻？在第 9 个时钟之后，如果有某一方需要更多的时间来处理数据，它可以一直驱动三极管把 SCL 拉低。

当 SCL 为低电平时候，大家都不应该使用 IIC 总线，只有当 SCL 从低电平变为高电平的时候，IIC 总线才能被使用。

当它就绪后，就可以不再驱动三极管，这是上拉电阻把 SCL 变为高电平，其他设备就可以继续使用 I2C 总线了。

对于 IIC 协议它只能规定怎么传输数据，数据是什么含义由从设备决定。

10.3 SMBus 协议

参考资料：

- Linux 内核文档：Documentation\i2c\smbus-protocol.rst
- SMBus 协议：
- http://www.smbus.org/specs/
- SMBus_3_0_20141220.pdf
- I2CTools:

https://mirrors.edge.kernel.org/pub/software/utils/i2c-tools/

10.3.1 SMBus 是 I2C 协议的一个子集

SMBus: System Management Bus，系统管理总线。

SMBus 最初的目的是为智能电池、充电电池、其他微控制器之间的通信链路而定义的。

SMBus 也被用来连接各种设备，包括电源相关设备，系统传感器，EEPROM 通讯设备等等。

SMBus 为系统和电源管理这样的任务提供了一条控制总线，使用 SMBus 的系统，设备之间发送和接收消息都是通过 SMBus，而不是使用单独的控制线，这样可以节省设备的管脚数。

SMBus 是基于 I2C 协议的，SMBus 要求更严格，SMBus 是 I2C 协议的子集。

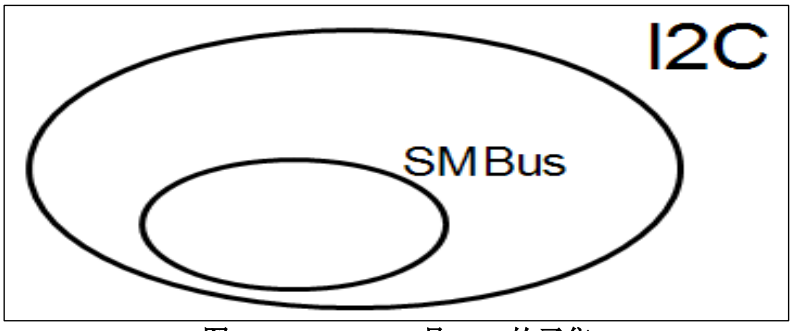


图 10.12 SMBus 是 I2C 的子集

SMBus 有哪些更严格的要求？跟一般的 I2C 协议有哪些差别？

- VDD 的极限值不一样
  - I2C 协议：范围很广，甚至讨论了高达 12V 的情况
  - SMBus：1.8V~5V
- 最小时钟频率、最大的 Clock Stretching
  - Clock Stretching 含义:某个设备需要更多时间进行内部的处理时，它可以把 SCL 拉低占住 I2C 总线
  - I2C 协议：时钟频率最小值无限制，Clock Stretching 时长也没有限制
  - SMBus：时钟频率最小值是 10KHz，Clock Stretching 的最大时间值也有限制
- 地址回应(Address Acknowledge)：一个 I2C 设备接收到它的设备地址后，是否必须发出回应信号？
  - I2C 协议：没有强制要求必须发出回应信号
  - SMBus：强制要求必须发出回应信号，这样对方才知道该设备的状态：busy，failed，或是被移除了
- SMBus 协议明确了数据的传输格式
  - I2C 协议：它只定义了怎么传输数据，但是并没有定义数据的格式，这完全由设备来定义
  - SMBus：定义了几种数据格式(后面分析)
- REPEATED START Condition(重复发出 S 信号)

  比如读 EEPROM 时，涉及 2 个操作：

  ① 把存储地址发给设备

  ② 读数据

在写、读之间，可以不发出 P 信号，而是直接发出 S 信号：这个 S 信号就是 REPEATED START，如图 10.13 所示

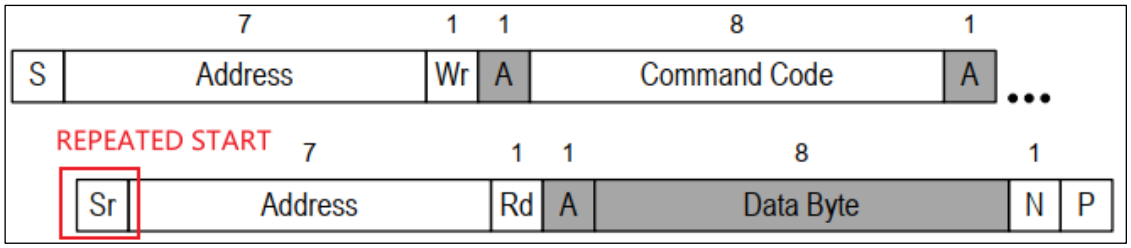


图 10.13 REPEATED START Condition

- SMBus Low Power Version：SMBus 也有低功耗的版本

10.3.2 SMBus 协议分析

对于 I2C 协议，它只定义了怎么传输数据，但是并没有定义数据的格式，这完全由设备来定义。

对于 SMBus 协议，它定义了几种数据格式。

注意：下面文档中的 Functionality flag 是 Linux 的某个 I2C 控制器驱动所支持的功能。比如 Functionality flag: I2C_FUNC_SMBUS_QUICK，表示需要 I2C 控制器支持 SMBus Quick Command

1 symbols(符号)

```
S     (1 bit) : Start bit(开始位)
Sr    (1 bit) : 重复的开始位
P     (1 bit) : Stop bit(停止位)
R/W#  (1 bit) : Read/Write bit. Rd equals 1, Wr equals 0.(读写位)
A, N  (1 bit) : Accept and reverse accept bit.(回应位)
Address(7 bits): I2C 7 bit address. Note that this can be expanded as usual to
                get a 10 bit I2C address.
                (地址位，7 位地址)
Command Code  (8 bits): Command byte, a data byte which often selects a register on
                the device.
                (命令字节，一般用来选择芯片内部的寄存器)
Data Byte (8 bits): A plain data byte. Sometimes, I write DataLow, DataHigh
                for 16 bit data.
                (数据字节，8 位；如果是 16 位数据的话，用 2 个字节来表示：DataLow、DataHigh)
Count (8 bits): A data byte containing the length of a block operation.
                (在 block 操作总，表示数据长度)
[..]:           Data sent by I2C device, as opposed to data sent by the host
                adapter.
                (中括号表示 I2C 设备发送的数据，没有中括号表示 host adapter 发送的数据)
```

2 SMBus Quick Command

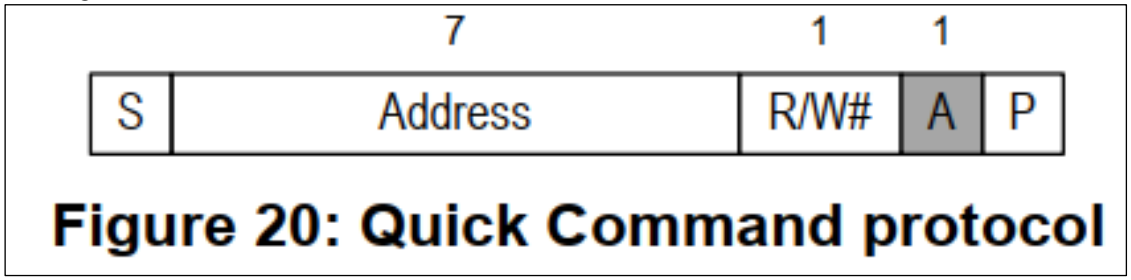


图 10.14 2 SMBus Quick Command

只是用来发送一位数据：R/W#本意是用来表示读或写，但是在 SMBus 里可以用来表示其他含义。比如某些开关设备，可以根据这一位来决定是打开还是关闭。

Functionality flag: I2C_FUNC_SMBUS_QUICK

3 3 SMBus Receive Byte

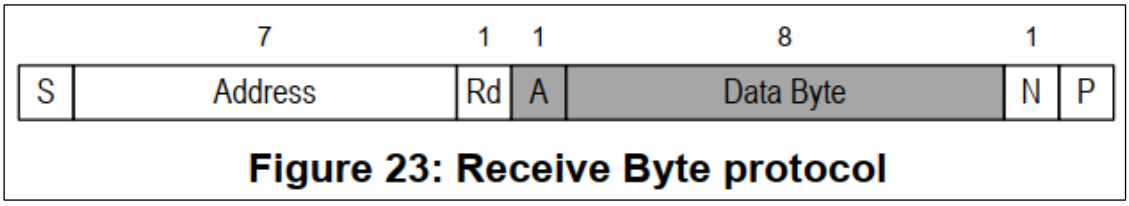


图 10.15 SMBus Receive Byte

I2C-tools 中的函数：i2c_smbus_read_byte()。读取一个字节，Host adapter 接收到一个字节后不需要发出回应信号(上图中 N 表示不回应)。

Functionality flag: I2C_FUNC_SMBUS_READ_BYTE

4 4 SMBus Send Byte

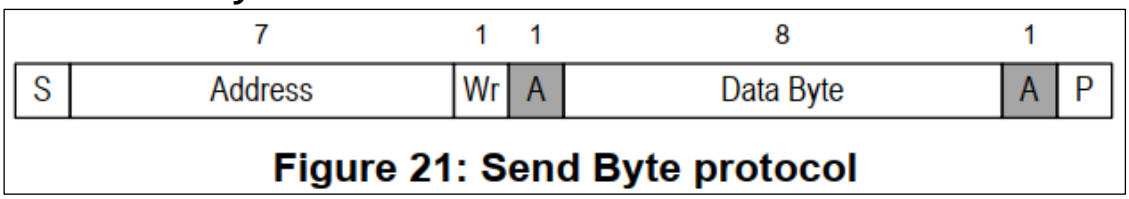


图 10.16 SMBus Send Byte

I2C-tools 中的函数：i2c_smbus_write_byte()。发送一个字节。

Functionality flag: I2C_FUNC_SMBUS_WRITE_BYTE

5 SMBus Read Byte

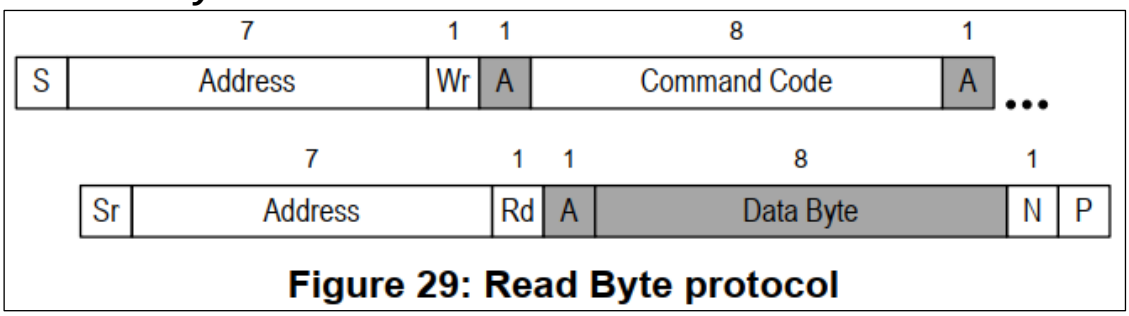


图 10.17 5 SMBus Read Byte

I2C-tools 中的函数：i2c_smbus_read_byte_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再读取一个字节的数据。上面介绍的 SMBus Receive Byte 是不发送 Comand，直接读取数据。

Functionality flag: I2C_FUNC_SMBUS_READ_BYTE_DATA

6 6 SMBus Read Word

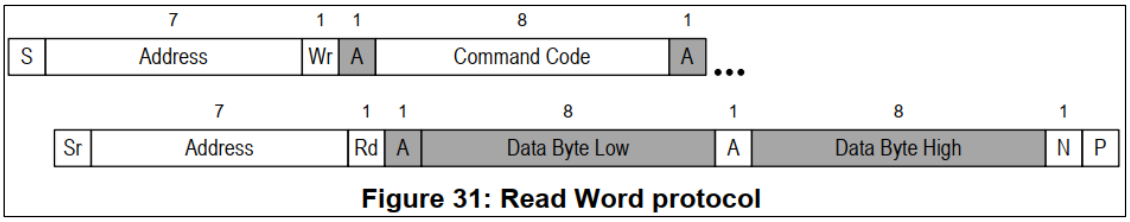


Figure 31: Read Word protocol

图 10.18 SMBus Read Word

I2C-tools 中的函数：i2c_smbus_read_word_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再读取 2 个字节的数据。

```
Functionality flag: I2C_FUNC_SMBUS_READ_WORD_DATA
```

7 SMBus Write Byte

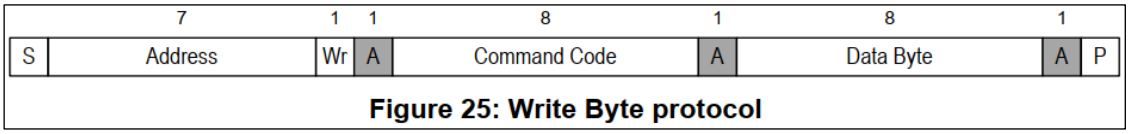


Figure 25: Write Byte protocol

图 10.19 SMBus Write Byte

I2C-tools 中的函数：i2c_smbus_write_byte_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发出 1 个字节的数据。

```
Functionality flag: I2C_FUNC_SMBUS_WRITE_BYTE_DATA
```

8 SMBus Write Word

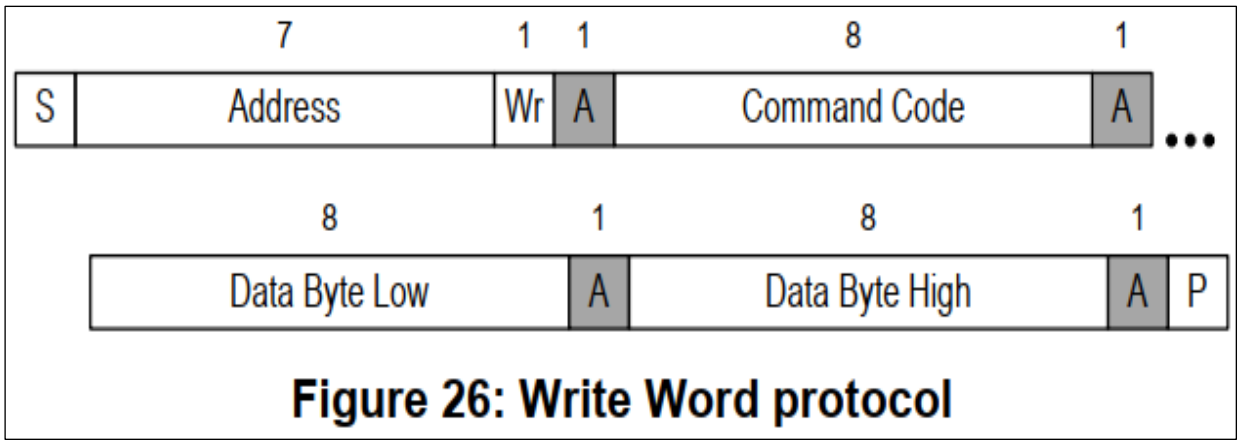


Figure 26: Write Word protocol

图 10.20 8 SMBus Write Word

I2C-tools 中的函数：i2c_smbus_write_word_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发出 1 个字节的数据。

```
Functionality flag: I2C_FUNC_SMBUS_WRITE_WORD_DATA
```

9 SMBus Block Read

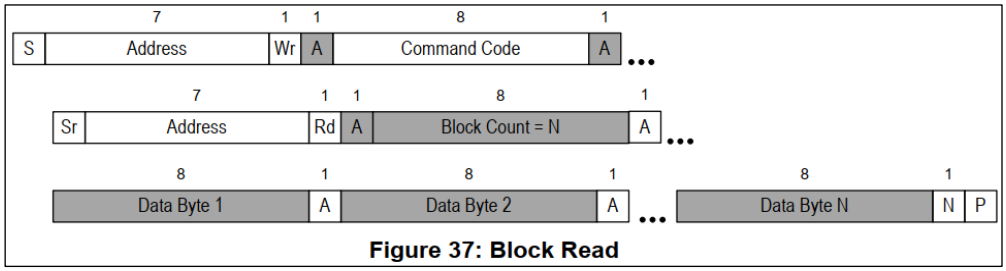


Figure 37: Block Read

图 10.21 9 SMBus Block Read

I2C-tools 中的函数：i2c_smbus_read_block_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发起度操作：

- 先读到一个字节(Block Count)，表示后续要读的字节数
- 然后读取全部数据

Functionality flag: I2C_FUNC_SMBUS_READ_BLOCK_DATA

10 SMBus Block Write

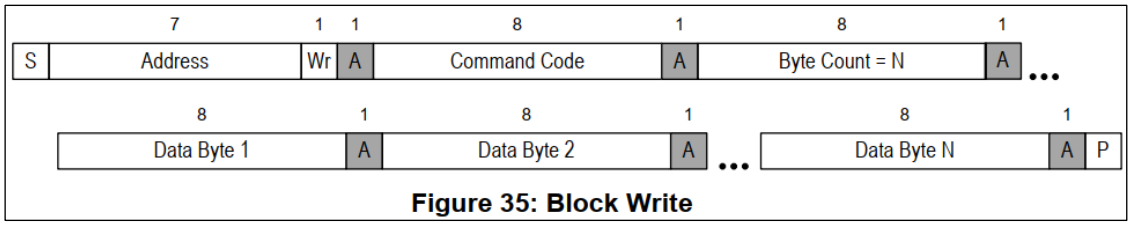


Figure 35: Block Write

图 10.22 10 SMBus Block Write

I2C-tools 中的函数：i2c_smbus_write_block_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发出 1 个字节的 Byte Conut(表示后续要发出的数据字节数)，最后发出全部数据。

Functionality flag: I2C_FUNC_SMBUS_WRITE_BLOCK_DATA

11 I2C Block Read

在一般的 I2C 协议中，也可以连续读出多个字节。它跟 SMBus Block Read 的差别在于设备发出的第 1 个数据不是长度 N，如下图所示：

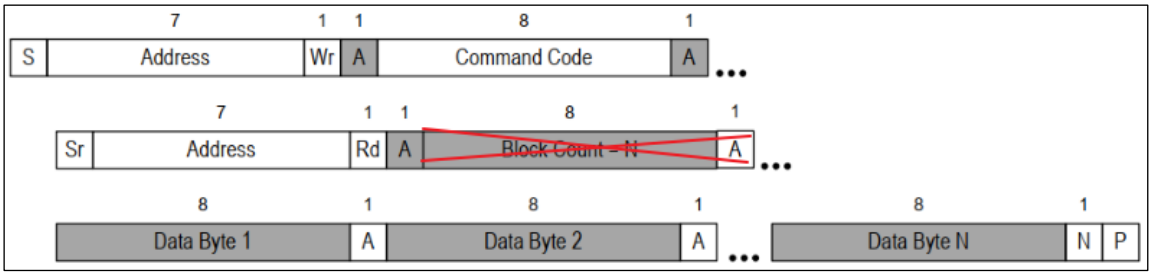


图 10.23 11 I2C Block Read

I2C-tools 中的函数：i2c_smbus_read_i2c_block_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发出 1 个字节的 Byte Conut(表示后续要发出的数据字节数)，最后发出全部数据。

Functionality flag: I2C_FUNC_SMBUS_READ_I2C_BLOCK

12 I2C Block Write

在一般的 I2C 协议中，也可以连续发出多个字节。它跟 SMBus Block Write 的差别在于发出的第 1 个数据不是长度 N，如下图所示：

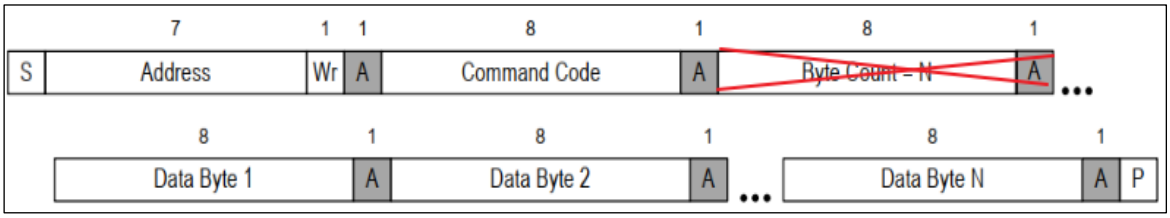


图 10.24 12 I2C Block Write

I2C-tools 中的函数：i2c_smbus_write_i2c_block_data()。先发出 Command Code(它一般表示芯片内部的寄存器地址)，再发出 1 个字节的 Byte Conut(表示后续要发出的数据字节数)，最后发出全部数据。

Functionality flag: I2C_FUNC_SMBUS_WRITE_I2C_BLOCK

13 SMBus Block Write - Block Read Process Call

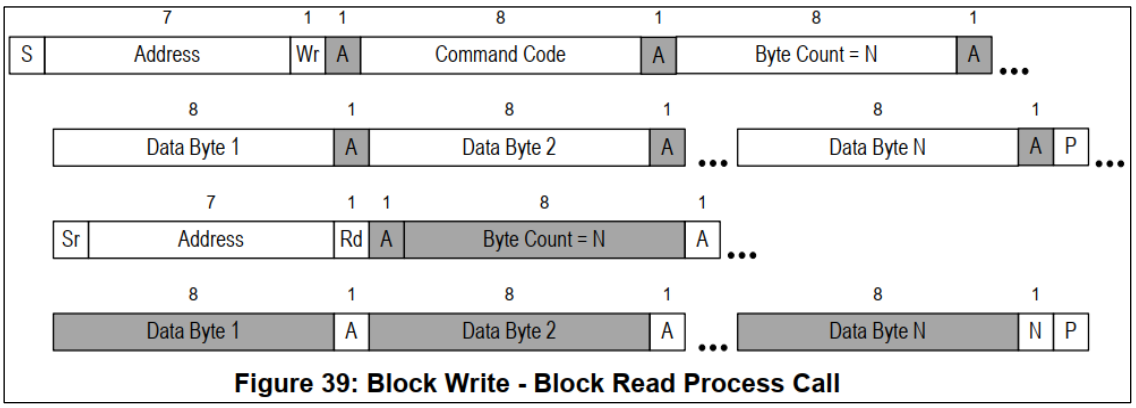


Figure 39: Block Write - Block Read Process Call

图 10.25 13 SMBus Block Write - Block Read Process Call

先写一块数据，再读一块数据。

Functionality flag: I2C_FUNC_SMBUS_BLOCK_PROC_CALL

14 Packet Error Checking (PEC)

PEC 是一种错误校验码，如果使用 PEC，那么在 P 信号之前，数据发送方要发送一个字节的 PEC 码(它是 CRC-8 码)。以 SMBus Send Byte 为例，下图中，一个未使用 PEC，另一个使用 PEC：

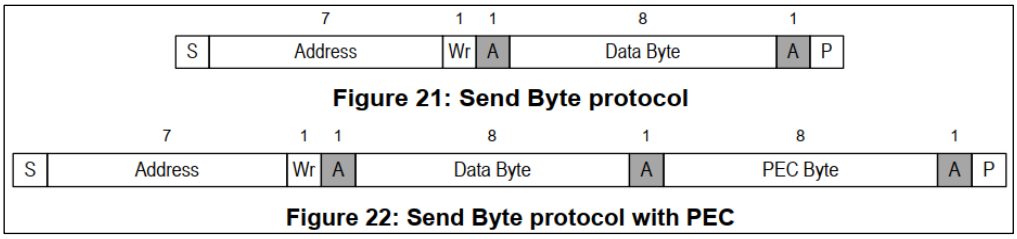


图 10.26 Packet Error Checking

10.3.3 SMBus 和 I2C 的建议

因为很多设备都实现了 SMBus，而不是更宽泛的 I2C 协议，所以优先使用 SMBus。即使 I2C 控制器没有实现 SMBus，软件方面也是可以使用 I2C 协议来模拟 SMBus。所以：Linux 建议优先使用 SMBus。

10.4 I2C 系统的重要结构体

参考资料：

- Linux 驱动程序：*（某版本的 Linux，比如 Linux-4.9.88）/drivers/i2c*
- I2CTools：

https://mirrors.edge.kernel.org/pub/software/utils/i2c-tools/

10.4.1 重要结构体

使用一句话概括 I2C 传输：APP 通过 I2C Controller 与 I2C Device 传输数据。

在 Linux 中要思索下面几个问题。

- 怎么表示 I2C Controller
- 一个芯片里可能有多个 I2C Controller，比如第 0 个、第 1 个、……
- 对于使用者，只要确定是第几个 I2C Controller 即可
- 使用 i2c_adapter 表示一个 I2C BUS，或称为 I2C Controller，里面有 2 个重要的成员：

a) nr：第几个 I2C BUS(I2C Controller)

b) i2c_algorithm，里面有该 I2C BUS 的传输函数，用来收发 I2C 数据

i2c_adapter 原型：

```
/*
 * i2c_adapter is the structure used to identify a physical i2c bus along
 * with the access algorithms necessary to access it.
 */
struct i2c_adapter {
    struct module *owner;
    unsigned int class;       /* classes to allow probing for */
    const struct i2c_algorithm *algo; /* the algorithm to access the bus */
    void *algo_data;

    /* data fields that are valid for all devices   */
    const struct i2c_lock_operations *lock_ops;
    struct rt_mutex bus_lock;
    struct rt_mutex mux_lock;

    int timeout;            /* in jiffies */
    int retries;
    struct device dev;      /* the adapter device */
    unsigned long locked_flags; /* owned by the I2C core */
#define I2C_ALF_IS_SUSPENDED        0
#define I2C_ALF_SUSPEND_REPORTED    1

    int nr;
    char name[48];
    struct completion dev_released;

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;

    struct i2c_bus_recovery_info *bus_recovery_info;
    const struct i2c_adapter_quirks *quirks;

    struct irq_domain *host_notify_domain;
} « end i2c_adapter » ;
```

图 10.27 i2c_adapter 原型

i2c_algorithm 原型：

```
struct i2c_algorithm {
    /*
     * If an adapter algorithm can't do I2C-level access, set master_xfer
     * to NULL. If an adapter algorithm can do SMBus access, set
     * smbus_xfer. If set to NULL, the SMBus protocol is simulated
     * using common I2C messages.
     *
     * master_xfer should return the number of messages successfully
     * processed, or a negative value on error
     */
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
               int num);
    int (*master_xfer_atomic)(struct i2c_adapter *adap,
                  struct i2c_msg *msgs, int num);
    int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
              unsigned short flags, char read_write,
              u8 command, int size, union i2c_smbus_data *data);
    int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,
                 unsigned short flags, char read_write,
                 u8 command, int size, union i2c_smbus_data *data);

    /* To determine what the adapter supports */
    u32 (*functionality)(struct i2c_adapter *adap);

#if IS_ENABLED(CONFIG_I2C_SLAVE)
    int (*reg_slave)(struct i2c_client *client);
    int (*unreg_slave)(struct i2c_client *client);
#endif
} « end i2c_algorithm » ;
```

图 10.28 i2c_algorithm 原型

1 怎么表示 I2C Device

- 一个 I2C Device，一定有**设备地址**
- 它连接在哪个 I2C Controller 上，即对应的 i2c_adapter 是什么

使用 i2c_client 来表示一个 I2C Device

```
struct i2c_client {
    unsigned short flags;       /* div., see below      */
#define I2C_CLIENT_PEC      0x04    /* Use Packet Error Checking */
#define I2C_CLIENT_TEN      0x10    /* we have a ten bit chip address */
                    /* Must equal I2C_M_TEN below */
#define I2C_CLIENT_SLAVE    0x20    /* we are the slave */
#define I2C_CLIENT_HOST_NOTIFY  0x40    /* We want to use I2C host notify */
#define I2C_CLIENT_WAKE     0x80    /* for board_info; true iff can wake */
#define I2C_CLIENT_SCCB     0x9000  /* Use Omnivision SCCB protocol */
                    /* Must match I2C_M_STOP|IGNORE_NAK */

    unsigned short addr;        /* chip address - NOTE: 7bit    */
                    /* addresses are stored in the  */
                    /* _LOWER_ 7 bits       */
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter;    /* the adapter we sit on    */
    struct device dev;      /* the device structure     */
    int init_irq;           /* irq set at initialization    */
    int irq;            /* irq issued by device     */
    struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb;    /* callback for slave mode  */
#endif
} « end i2c_client » ;
```

图 10.29 i2c_client

2 怎么表示要传输的数据

在上面的 i2c_algorithm 结构体中可以看到要传输的数据被称为：i2c_msg

i2c_msg 原型：

```
struct i2c_msg {
    __u16 addr; /* slave address            */
    __u16 flags;
#define I2C_M_RD        0x0001  /* read data, from slave to master */
                    /* I2C_M_RD is guaranteed to be 0x0001! */
#define I2C_M_TEN       0x0010  /* this is a ten bit chip address */
#define I2C_M_DMA_SAFE      0x0200  /* the buffer of this message is DMA safe */
                    /* makes only sense in kernelspace */
                    /* userspace buffers are copied anyway */
#define I2C_M_RECV_LEN      0x0400  /* length will be first received byte */
#define I2C_M_NO_RD_ACK     0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK    0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_REV_DIR_ADDR  0x2000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NOSTART       0x4000  /* if I2C_FUNC_NOSTART */
#define I2C_M_STOP      0x8000  /* if I2C_FUNC_PROTOCOL_MANGLING */
    __u16 len;      /* msg length               */
    __u8 *buf;      /* pointer to msg data          */
};
```

图 10.30 i2c_msg

- i2c_msg 中的 flags 用来表示传输方向：bit 0 等于 I2C_M_RD 表示读，bit 0 等于 0 表示写
- 一个 i2c_msg 要么是读，要么是写

举例：设备地址为 0x50 的 EEPROM，要读取它里面存储地址为 0x10 的一个字节，应该构造几个 i2c_msg？要构造 2 个 i2c_msg

c) 第一个 i2c_msg 表示写操作，把要访问的存储地址 0x10 发给设备

d) 第二个 i2c_msg 表示读操作

代码如下

```
u8 data_addr = 0x10;
i8 data;
struct i2c_msg msgs[2];

msgs[0].addr   = 0x50;
msgs[0].flags  = 0;
msgs[0].len    = 1;
msgs[0].buf    = &data_addr;

msgs[1].addr   = 0x50;
msgs[1].flags  = I2C_M_RD;
msgs[1].len    = 1;
msgs[1].buf    = &data;
```

10.4.2 内核里怎么传输数据

使用一句话概括 I2C 传输：

a) APP 通过 I2C Controller 与 I2C Device 传输数据

b) APP 通过 i2c_adapter 与 i2c_client 传输 i2c_msg

c) 内核函数 i2c_transfer

i2c_msg 里含有 addr，所以这个函数里不需要 i2c_client

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
```

图 10.31 i2c_transfer

10.5 无需编写驱动程序即可访问 I2C 设备

APP 访问硬件肯定是需要驱动程序的，对于 I2C 设备，内核提供了驱动程序 ***drivers/i2c/i2c-dev.c***，通过它可以直接使用下面的 I2C 控制器驱动程序来访问 I2C 设备。

框架如下：

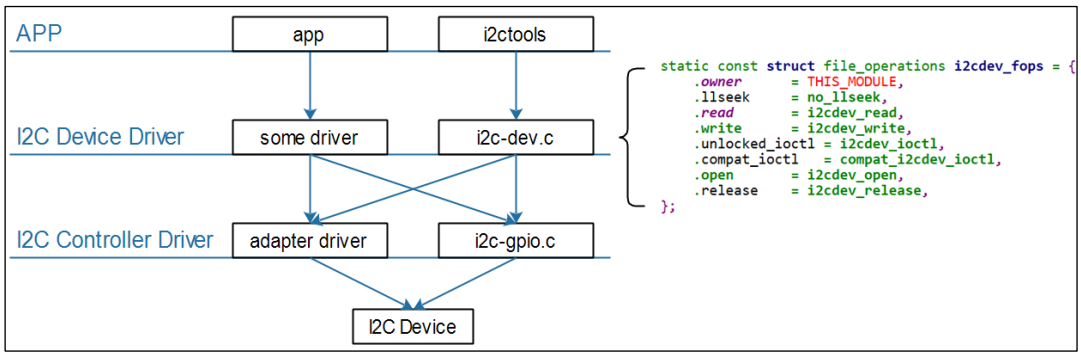


图 10.32 I2C 控制器框架

i2c-tools 是一套好用的工具，也是一套示例代码。

10.5.1 体验 I2C-Tools

使用一句话概括 I2C 传输：APP 通过 I2C Controller 与 I2C Device 传输数据。

所以使用 I2C-Tools 时也需要指定：

- 哪个 I2C 控制器(或称为 I2C BUS、I2C Adapter)
- 哪个 I2C 设备(设备地址)
- 数据：读还是写、数据本身

1 交叉编译

- 配置环境：vim ~/.bashrc

```
export ARCH=arm
export CROSS_COMPILE= arm-buildroot-linux-gnueabihf-gcc-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

- 修改 I2C-Tools 的 Makefile 指定交叉编译工具链

```
CC      ?= gcc
AR      ?= ar
STRIP   ?= strip
```

改为(指定交叉编译工具链前缀，去掉问号)：

```
CC      = $(CROSS_COMPILE)gcc
AR      = $(CROSS_COMPILE)ar
STRIP   = $(CROSS_COMPILE)strip
```

在 Makefile 中，“?=”在第一次设置变量时才会起效果，如果之前设置过该变量，则不会起效果。

2 执行 make

- 执行 make 时，是动态链接，需要把 libi2c.so 也放到单板上
- 想静态链接的话，执行：make USE_STATIC_LIB=1

3 用法

- i2cdetect：I2C 检测

```
// 列出当前的 I2C Adapter(或称为 I2C Bus、I2C Controller)
i2cdetect -l

// 打印某个 I2C Adapter 的 Functionalities，I2CBUS 为 0、1、2 等整数
i2cdetect -F I2CBUS

// 看看有哪些 I2C 设备，I2CBUS 为 0、1、2 等整数
i2cdetect -y -a I2CBUS

// 效果如下
# i2cdetect -l
i2c-1   i2c             STM32F7 I2C(0x40013000)                 I2C adapter
i2c-2   i2c             STM32F7 I2C(0x5c002000)                 I2C adapter
i2c-0   i2c             STM32F7 I2C(0x40012000)                 I2C adapter
```

```
# i2cdetect -F 0
Functionalities implemented by /dev/i2c-0:
I2C                              yes
SMBus Quick Command              yes
SMBus Send Byte                  yes
SMBus Receive Byte               yes
SMBus Write Byte                 yes
SMBus Read Byte                  yes
SMBus Write Word                 yes
SMBus Read Word                  yes
SMBus Process Call               yes
SMBus Block Write                yes
SMBus Block Read                 yes
SMBus Block Process Call         yes
SMBus PEC                        yes
I2C Block Write                  yes
I2C Block Read                   yes

// --表示没有该地址对应的设备，UU 表示有该设备并且它已经有驱动程序，
// 数值表示有该设备但是没有对应的设备驱动
# i2cdetect -y -a 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: 00 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- UU -- -- -- 1e --
20: -- -- UU -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

- i2cget：I2C 读

使用说明如下：

```
# i2cget
Usage: i2cget [-f] [-y] [-a] I2CBUS CHIP-ADDRESS [DATA-ADDRESS [MODE]]
  I2CBUS is an integer or an I2C bus name
  ADDRESS is an integer (0x03 - 0x77, or 0x00 - 0x7f if -a is given)
  MODE is one of:
    b (read byte data, default)
    w (read word data)
    c (write byte/read byte)
Append p for SMBus PEC
```

使用示例：

```
// 读一个字节：I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
i2cget -f -y I2CBUS CHIP-ADDRESS

// 读某个地址上的一个字节：
//    I2CBUS 为 0、1、2 等整数，表示 I2C Bus
//    CHIP-ADDRESS 表示设备地址
//    DATA-ADDRESS：芯片上寄存器地址
//    MODE：有 2 个取值，b-使用`SMBus Read Byte`先发出 DATA-ADDRESS，再读一个字节，中间无
P 信号
//                  c-先 write byte，在 read byte，中间有 P 信号
i2cget -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS MODE
```

```
// 读某个地址上的 2 个字节：
//    I2CBUS 为 0、1、2 等整数，表示 I2C Bus
//    CHIP-ADDRESS 表示设备地址
//    DATA-ADDRESS：芯片上寄存器地址
//    MODE：w-表示先发出 DATA-ADDRESS，再读 2 个字节
i2cget -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS MODE
```

- i2cset：I2C 写

使用说明如下：

```
# i2cset
Usage: i2cset [-f] [-y] [-m MASK] [-r] [-a] I2CBUS CHIP-ADDRESS DATA-ADDRESS [VALUE]
 ... [MODE]
  I2CBUS is an integer or an I2C bus name
  ADDRESS is an integer (0x03 - 0x77, or 0x00 - 0x7f if -a is given)
  MODE is one of:
    c (byte, no value)
    b (byte data, default)
    w (word data)
  i (I2C block data)
    s (SMBus block data)
Append p for SMBus PEC
```

使用示例：

```
// 写一个字节：I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
  //          DATA-ADDRESS 就是要写的数据
  i2cset -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS

  // 给 address 写 1 个字节(address, value):
  //          I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
  //          DATA-ADDRESS：8 位芯片寄存器地址；
  //          VALUE：8 位数值
  //          MODE：可以省略，也可以写为 b
  i2cset -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS VALUE [b]

  // 给 address 写 2 个字节(address, value):
  //          I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
  //          DATA-ADDRESS：8 位芯片寄存器地址；
  //          VALUE：16 位数值
  //          MODE：w
  i2cset -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS VALUE w

  // SMBus Block Write：给 address 写 N 个字节的数据
  //   发送的数据有：address, N, value1, value2, ..., valueN
  //   跟`I2C Block Write`相比，需要发送长度 N
  //          I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
  //          DATA-ADDRESS：8 位芯片寄存器地址；
  //          VALUE1~N：N 个 8 位数值
  //          MODE：s
  i2cset -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS VALUE1 ... VALUEN s

  // I2C Block Write：给 address 写 N 个字节的数据
  //   发送的数据有：address, value1, value2, ..., valueN
  //   跟`SMBus Block Write`相比，不需要发送长度 N
```

```
  //           I2CBUS 为 0、1、2 等整数，表示 I2C Bus；CHIP-ADDRESS 表示设备地址
  //           DATA-ADDRESS：8 位芯片寄存器地址；
  //           VALUE1~N：N 个 8 位数值
  //           MODE：i
  i2cset -f -y I2CBUS CHIP-ADDRESS DATA-ADDRESS VALUE1 ... VALUEN i
```

- i2ctransfer：I2C 传输(不是基于 SMBus)

使用说明如下：

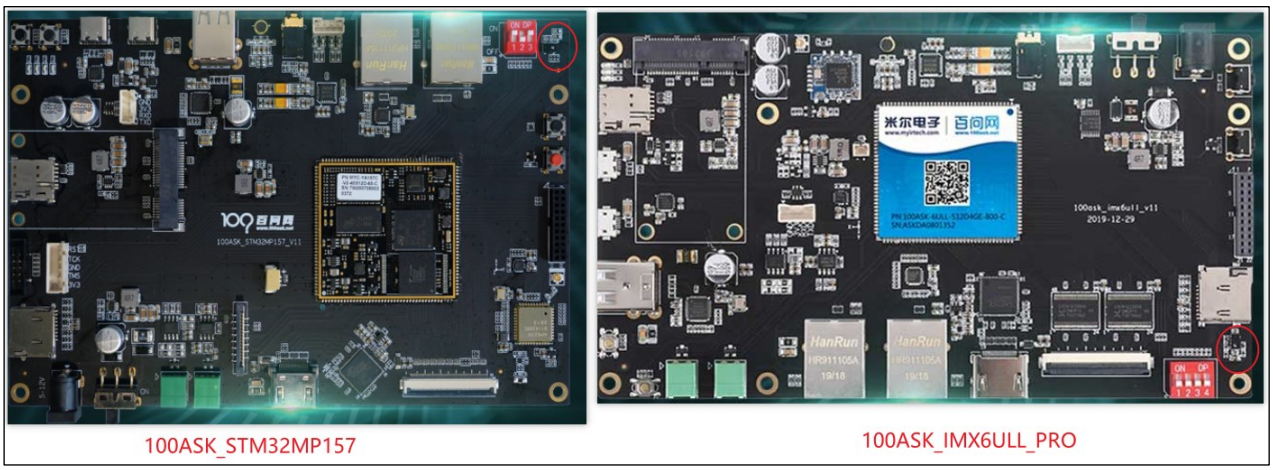
```
# i2ctransfer
Usage: i2ctransfer [-f] [-y] [-v] [-V] [-a] I2CBUS DESC [DATA] [DESC [DATA]]...
  I2CBUS is an integer or an I2C bus name
  DESC describes the transfer in the form: {r|w}LENGTH[@address]
    1) read/write-flag 2) LENGTH (range 0-65535) 3) I2C address (use last one if omit
ted)
  DATA are LENGTH bytes for a write message. They can be shortened by a suffix:
    = (keep value constant until LENGTH)
    + (increase value by 1 until LENGTH)
    - (decrease value by 1 until LENGTH)
    p (use pseudo random generator until LENGTH with value as seed)

Example (bus 0, read 8 byte at offset 0x64 from EEPROM at 0x50):
  # i2ctransfer 0 w1@0x50 0x64 r8
Example (same EEPROM, at offset 0x42 write 0xff 0xfe ... 0xf0):
  # i2ctransfer 0 w17@0x50 0x42 0xff-
```

使用举例：

```
// Example (bus 0, read 8 byte at offset 0x64 from EEPROM at 0x50):
# i2ctransfer -f -y 0 w1@0x50 0x64 r8

// Example (bus 0, write 3 byte at offset 0x64 from EEPROM at 0x50):
# i2ctransfer -f -y 0 w9@0x50 0x64 val1 val2 val3

// Example
// first: (bus 0, write 3 byte at offset 0x64 from EEPROM at 0x50)
// and then: (bus 0, read 3 byte at offset 0x64 from EEPROM at 0x50)
# i2ctransfer -f -y 0 w9@0x50 0x64 val1 val2 val3 r3@0x50
# i2ctransfer -f -y 0 w9@0x50 0x64 val1 val2 val3 r3 //如果设备地址不变,后面的设备地址可
省略
```

4 使用 I2C-Tools 操作传感器 AP3216C

百问网的开发板上有光感芯片 AP3216C：



AP3216C 是红外、光强、距离三合一的传感器，以读出光强、距离值为例，步骤如下：

- 复位：往寄存器 0 写入 0x4
- 使能：往寄存器 0 写入 0x3
- 读光强：读寄存器 0xC、0xD 得到 2 字节的光强
- 读距离：读寄存器 0xE、0xF 得到 2 字节的距离值

AP3216C 的设备地址是 0x1E，假设节在 I2C BUS0 上，操作命令如下：

- 使用 SMBus 协议

```
i2cset -f -y 0 0x1e 0 0x4
i2cset -f -y 0 0x1e 0 0x3
i2cget -f -y 0 0x1e 0xc w
i2cget -f -y 0 0x1e 0xe w
```

- 使用 I2C 协议

```
i2ctransfer -f -y 0 w2@0x1e 0 0x4
i2ctransfer -f -y 0 w2@0x1e 0 0x3
i2ctransfer -f -y 0 w1@0x1e 0xc r2
i2ctransfer -f -y 0 w1@0x1e 0xe r2
```

10.5.2 I2C-Tools 访问 I2C 设备的 2 种方式

I2C-Tools 可以通过 SMBus 来访问 I2C 设备，也可以使用一般的 I2C 协议来访问 I2C 设备。

使用一句话概括 I2C 传输：APP 通过 I2C Controller 与 I2C Device 传输数据。

在 APP 里，有这几个问题：

① 怎么指定 I2C 控制器？

- i2c-dev.c 为每个 I2C 控制器(I2C Bus、I2C Adapter)都生成一个设备节点：/dev/i2c-0、/dev/i2c-1 等等；
- open 某个/dev/i2c-X 节点，就是去访问该 I2C 控制器下的设备；

② 怎么指定 I2C 设备？

通过 ioctl 指定 I2C 设备的地址

- ioctl(file, I2C_SLAVE, address)
  - 如果该设备已经有了对应的设备驱动程序，则返回失败。
- ioctl(file, I2C_SLAVE_FORCE, address)
  - 如果该设备已经有了对应的设备驱动程序但是还是想通过 i2c-dev 驱动来访问它，则使用这个 ioctl 来指定 I2C 设备地址。

③ 怎么传输数据？

两种方式

- 一般的 I2C 方式：ioctl(file, I2C_RDWR, &rdwr)
- SMBus 方式：ioctl(file, I2C_SMBUS, &args)

10.5.3 源码流程分析

1 使用 I2C 方式

示例代码：i2ctransfer.c

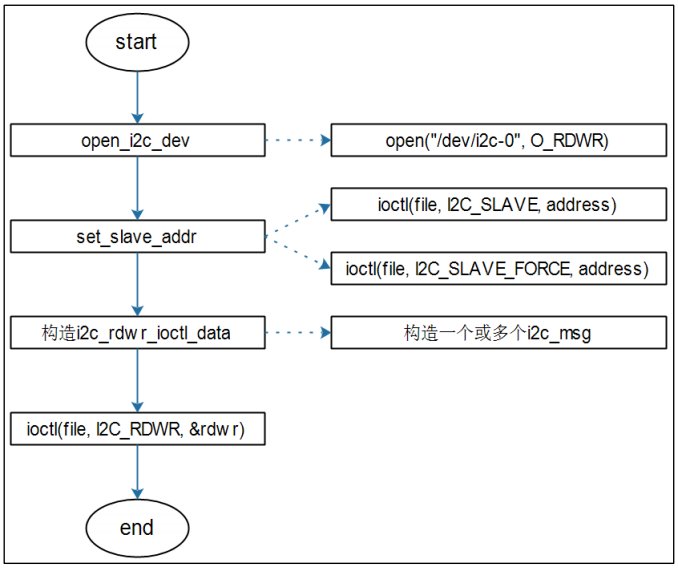


图 10.33 I2C 读写源码流程

2 使用 SMBus 方式

示例代码：i2cget.c、i2cset.c

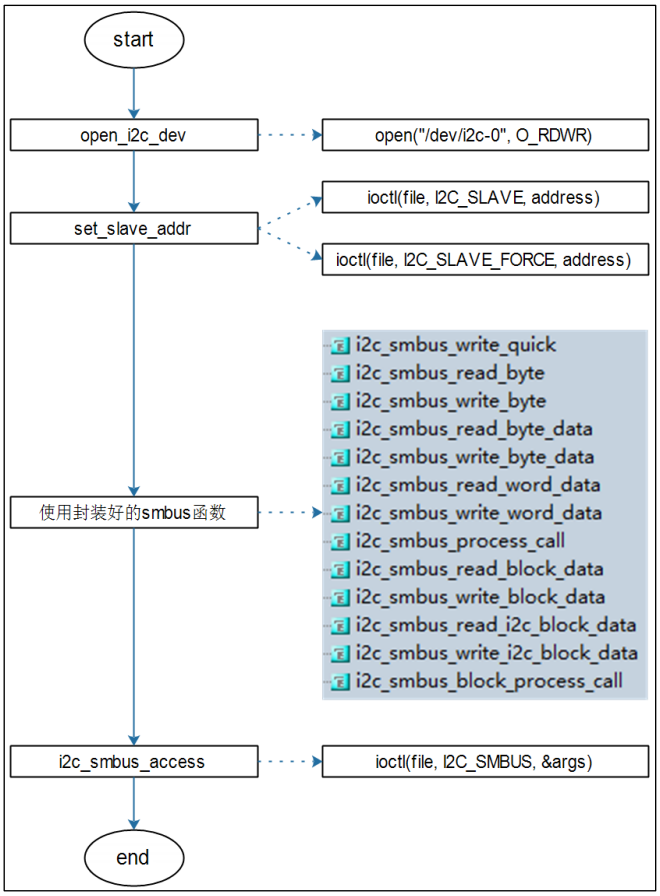


图 10.34 SMBus 读写源码流程

10.6 编写 APP 直接访问 EEPROM

参考资料：

- Linux 驱动程序：drivers/i2c/i2c-dev.c
- 2C-Tools-4.2：

https://mirrors.edge.kernel.org/pub/software/utils/i2c-tools/

- AT24cxx.pdf

10.6.1 硬件连接

IMX6ULL 的 I2C 模块连接方法：

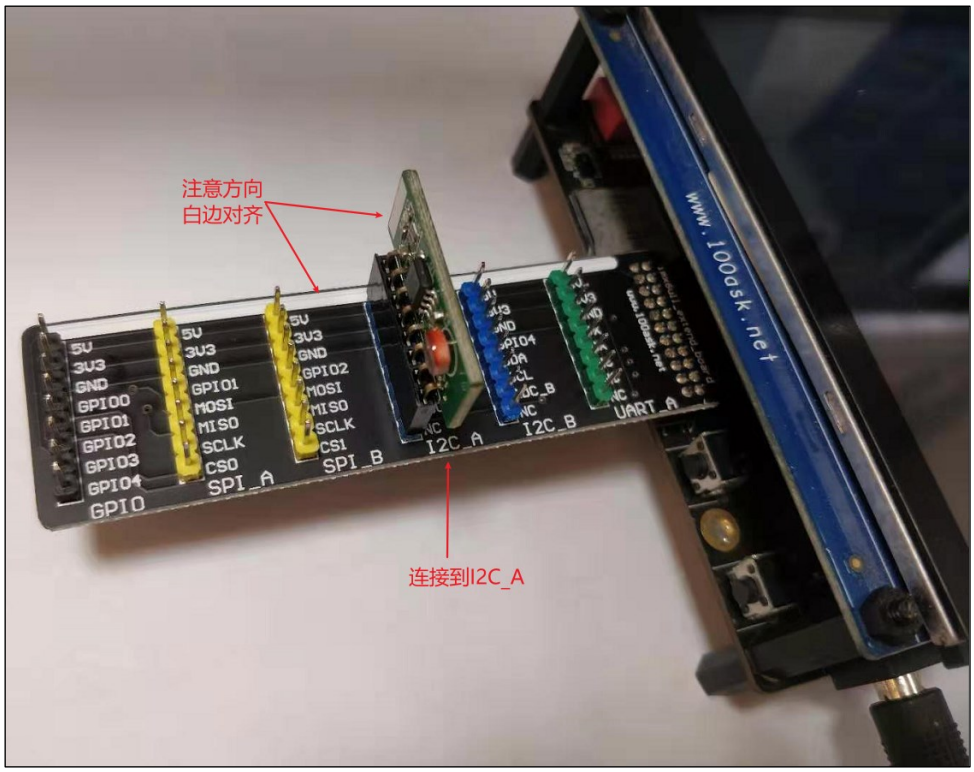


图 10.35

10.6.2 AT24C02 访问方法

1 设备地址

从芯片手册上可以知道，AT24C02 的设备地址跟它的 A2、A1、A0 引脚有关：

Figure 1. Device Address

1K/2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSD						LSB	
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W
16K	1	0	1	0	P2	P1	P0	R/W

图 10.36 AT24C02 设备地址引脚配置

打开 I2C 模块的原理图：

开发板配套网盘资料\04_开发板原理图\04_Extend_modules\通用模块\eeprom.zip\i2c_eeprom_module_v1.0.pdf

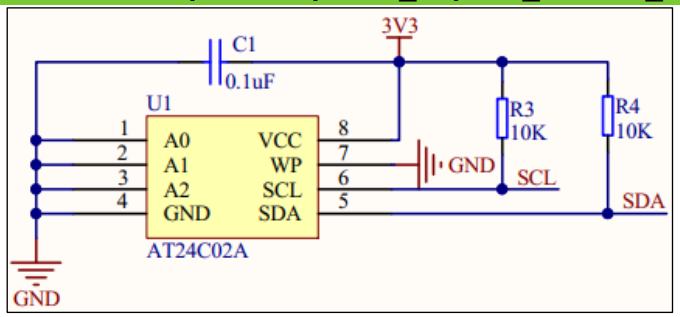


图 10.37

从原理图可知，A2A1A0 都是 0，所以 AT24C02 的设备地址是：0b1010000，即 0x50。

2 写数据

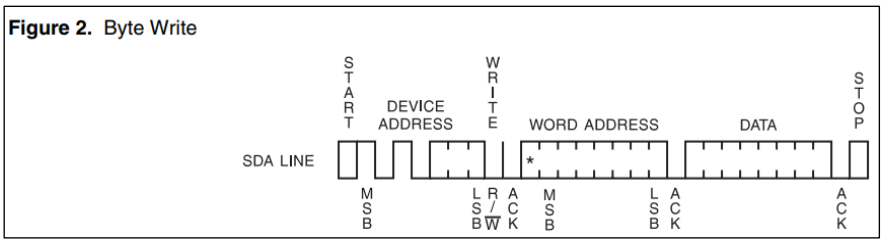


图 10.38 AT24C02 写数据时序

3 读数据

可以读 1 个字节，也可以连续读出多个字节。连续读多个字节时，芯片内部的地址会自动累加。当地址到达存储空间最后一个地址时，会从 0 开始。

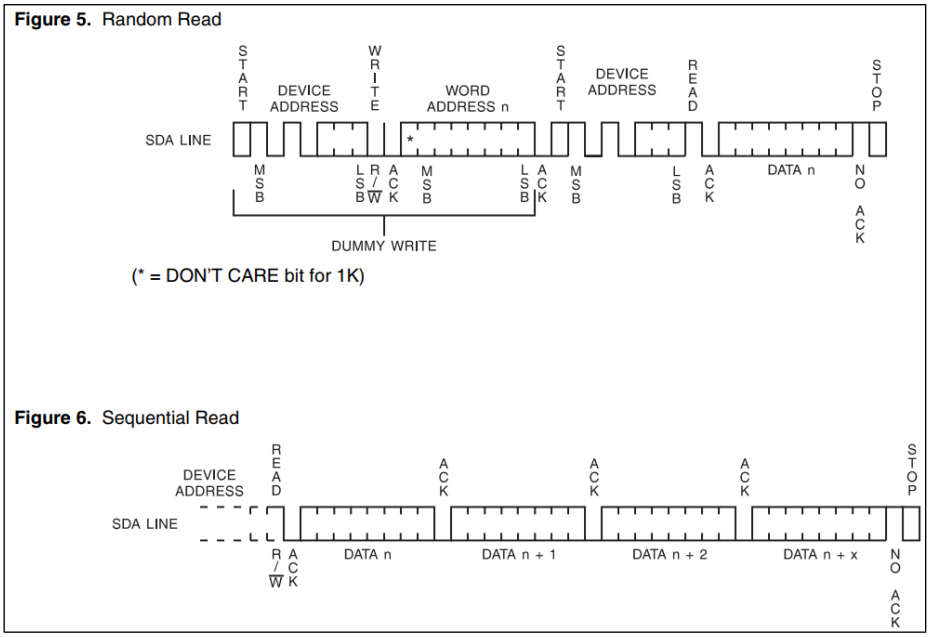


图 10.39 AT24C02 读数据时序

10.6.3 使用 I2C-Tools 的函数编程

I2C_Tools 上一小节已经讲解过，读者可自行学习编写程序。

10.6.4 编译

编译应用程序需要设置交叉编译工具链：`vim ~/.bashrc`

```
export ARCH=arm
export CROSS_COMPILE= arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

1 使用 I2C-Tools 的源码

01_all_series_quickstart
04_嵌入式 Linux 应用开发基础知识\source\15_I2C\01_at24c02_test

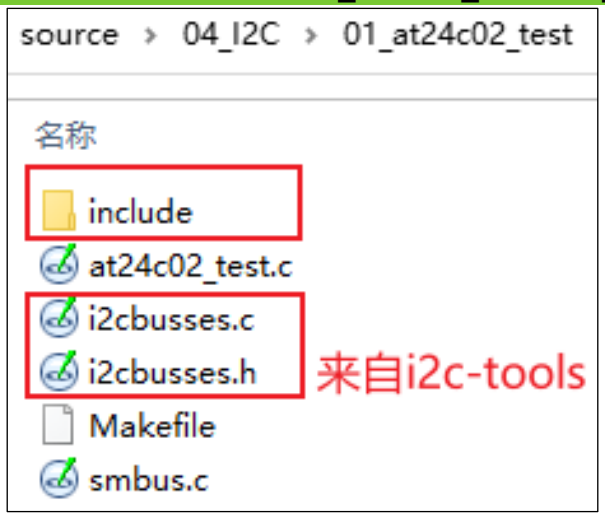


图 10.40 测试源码

2 编译

为 IMX6ULL 编译时，有如下错误：

```
book@book-virtual-machine:~/tmp/01_at24c02_test$ make
arm-linux-gnueabihf-gcc -o at24c02_test at24c02_test.c i2cbusses.c smbus.c
at24c02_test.c:10:23: fatal error: i2c/smbus.h: No such file or directory
 #include <i2c/smbus.h>
                       ^
compilation terminated.
smbus.c:16:23: fatal error: i2c/smbus.h: No such file or directory
 #include <i2c/smbus.h>
                       ^
compilation terminated.
Makefile:2: recipe for target 'all' failed
make: *** [all] Error 1
```

图 10.41 IMX6ULL 编译 I2C 例程错误提示

这是因为 IMX6ULL 的工具链自带的 include 目录中，没有 smbus.h，需要我们自己提供这个头文件，解决方法：

- 提供头文件：

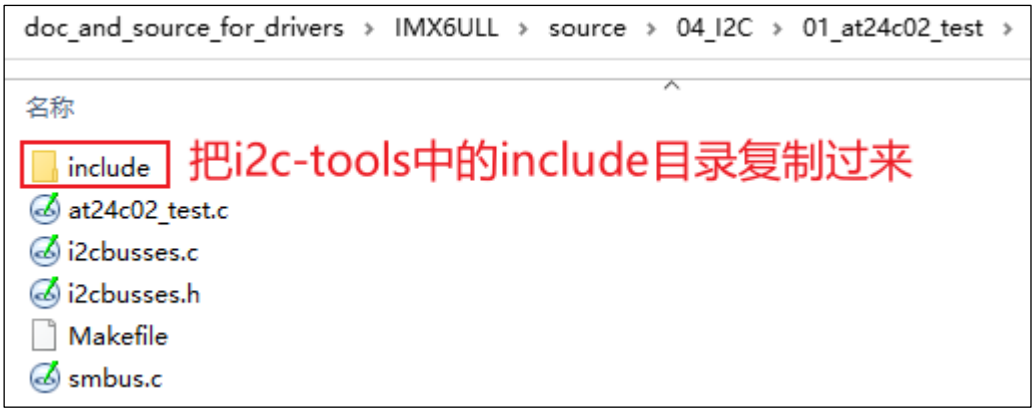


图 10.42 复制头文件

- 修改 Makefile 指定头文件目录

```
all:
  $(CROSS_COMPILE)gcc -I ./include -o at24c02_test at24c02_test.c i2cbusses.c smbus.c
```

10.6.5 上机测试

注意：以下命令在开发板中执行。

- 挂载 NFS

  vmware 使用桥接，或者不使用 vmware 而是直接使用服务器：假设 Ubuntu IP 为 192.168.1.137

```
[root@100ask:~]#  mount -t nfs -o nolock,vers=3 192.168.1.137:/home/book/nfs_rootfs /mnt
```

- 复制、执行程序

```
[root@100ask:~]# cp /mnt/at24c02_test   /bin
[root@100ask:~]# at24c02_test 0 w www.100ask.net
[root@100ask:~]# at24c02_test 0 r
get data: www.100ask.net
```

第11章 Linux 串口应用编程

参考资料：

- Serial Programming Guide for POSIX Operating Systems：
  https://digilander.libero.it/robang/rubrica/serial.htm
- Linux 串口编程：
  https://www.cnblogs.com/feisky/archive/2010/05/21/1740893.html
- Linux 串口—struct termios 结构体
  https://blog.csdn.net/yemingzhu163/article/details/5897156

11.1 串口 API

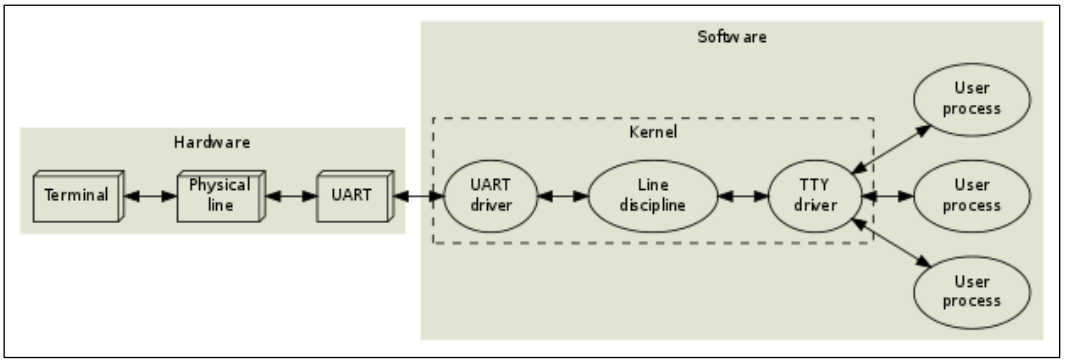


图 11.1 Linux 串口通信

在 Linux 系统中，操作设备的统一接口就是：open/ioctl/read/write。对于 UART，又在 ioctl 之上封装了很多函数，主要是用来设置行规程。所以对于 UART，编程的套路就是：

- open；
- 设置行规程，比如波特率、数据位、停止位、检验位、RAW 模式、一有数据就返回；
- read/write；

怎么设置行规程？行规程的参数用结构体 termios 来表示，可以参考 Linux 串口—struct termios 结构体：https://blog.csdn.net/yemingzhu163/article/details/5897156)

```
typedef unsigned char   cc_t;
typedef unsigned int    speed_t;
typedef unsigned int    tcflag_t;

#define NCCS 19
struct termios {
    tcflag_t c_iflag;       /* input mode flags */
    tcflag_t c_oflag;       /* output mode flags */
    tcflag_t c_cflag;       /* control mode flags */
    tcflag_t c_lflag;       /* local mode flags */
    cc_t c_line;            /* line discipline */
    cc_t c_cc[NCCS];        /* control characters */
};
```

这些函数在名称上有一些惯例：

- tc：terminal contorl
- cf：control flag

下面列出一些函数：

表 11-1 行规程函数

函数名	作用
tcgetattr	get terminal attributes，获得终端的属性
tcsetattr	set terminal attributes，修改终端参数
tcflush	清空终端未完成的输入/输出请求及数据
cfsetispeed	sets the input baud rate，设置输入波特率
cfsetospeed	sets the output baud rate，设置输出波特率
cfsetspeed	同时设置输入、输出波特率

函数不多，主要是需要设置好 termios 中的参数，这些参数很复杂，可以参考 Linux 串口—struct termios 结构体。

11.2 串口收发实验

本实验通过把串口的发送、接收引脚短接，实现自发自收：使用 write 函数发出字符，使用 read 函数读取字符。

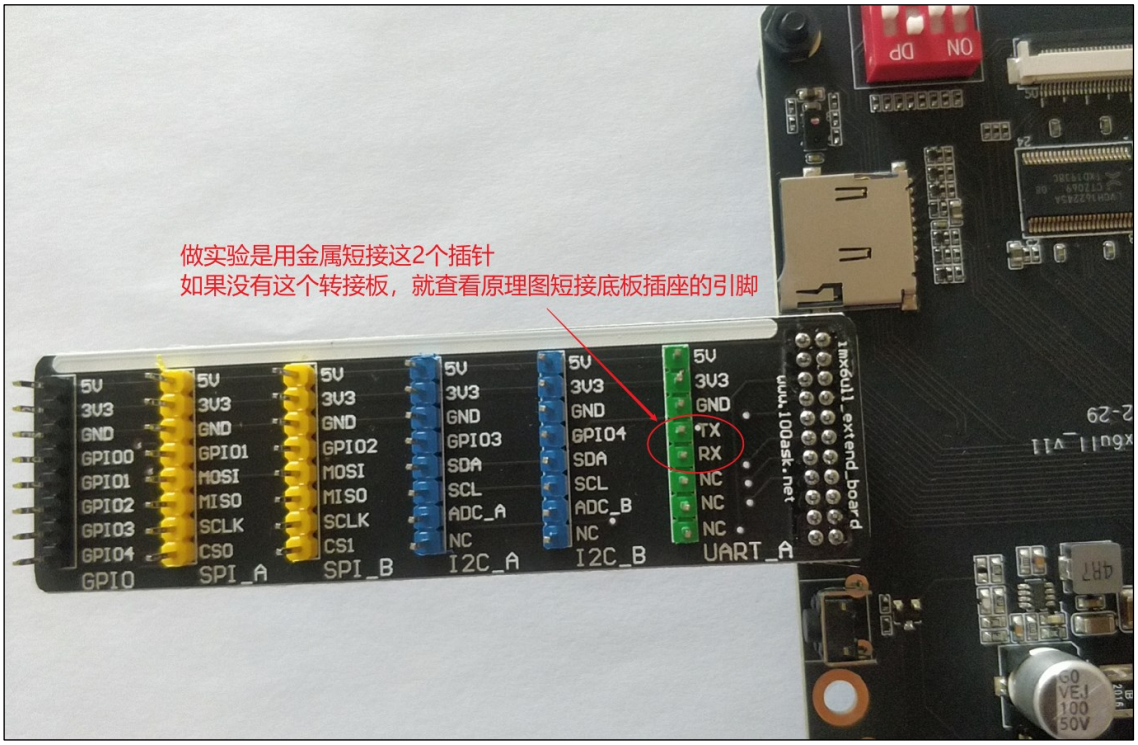


图 11.2 IMX6ULL 扩展板串口硬件连接

11.2.1 上机实验

1 设置工具链：vim ~/.bashrc

```
export ARCH=arm
export CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueab
```

```
ihf_sdk-buildroot/bin
```

2 编译、执行程序

```
1. Ubuntu 上
arm-buildroot-linux-gnueabihf-gcc -o serial_send_recv serial_send_recv.c
2. 板子上
/mnt/serial_send_recv  /dev/ttymxc5
```

11.2.2 GPS 模块实验

1 3.1 GPS 简介

全球定位系统(Global Positioning System，GPS)是一种以空中卫星为基础的高精度无线电导航的定位系统，它在全球任何地方以及近地空间都能够提供准确的地理位置、车行速度及精确的时间信息。GPS 主要由三大组成部分：空间部分、地面监控部分和用户设备部分。GPS 系统具有高精度、全天候、用广泛等特点。

太空卫星部分由多颗卫星组成，分成多个轨道，绕行地球一周约 12 小时。每个卫星均持续发射载有卫星轨道数据及时间的无线电波，提供地球上的各种接收机来应用。

地面管制部分，这是为了追踪及控制太空卫星运行所设置的地面管制站，主要工作为负责修正与维护每个卫星能够正常运转的各项参数数据，以确保每个卫星都能够提供正确的讯息给使用者接收机来接收

使用者接收机（即用户设备），追踪所有的 GPS 卫星，并实时的计算出接收机所在位置的坐标、移动速度及时间。我们日常接触到的是用户设备部分，这里使用到的 GPS 模块即为用户设备接收机部分。

2 3.2 GPS 模块硬件

GPS 模块与外部控制器的通讯接口有多种方式,这里我们使用串口进行通讯，波特率为 9600bps,1bit 停止位，无校验位，无流控，默认每秒输出一次标准格式数据。

GPS 模块外观如下图所示，通过排线与控制器进行供电和通讯。该模块为集成模块，没有相关原理图。

图 11.3 GPS 模块

3 GPS 模块数据格式

GPS 使用多种标准数据格式，目前最通用的 GNSS 格式是 NMEA0183 格式。NMEA0183 是最终定位格式，即将二进制定位格式转为统一标准定位格式，与卫星类型无关。这是一套定义接收机输出的标准信息，有几种不同的格式，每种都是独立相关的 ASCII 格式，逗点隔开数据流，数据流长度从 30-100 字符不等，通常以每秒间隔持续输出。

NVMEA0183 格式主要针对民用定位导航，与专业 RTCM2.3/3.0 和 CMR+的 GNSS 数据格式不同。通过 NMEA0183 格式，可以实现 GNSS 接收机与 PC 或 PDA 之间的数据交换，可以通过 USB 和 COM 口等通用数据接口进行数据传输，其兼容性高，数据传输稳定。这里我们使用串口进行是通讯，通信框图如下图所示。

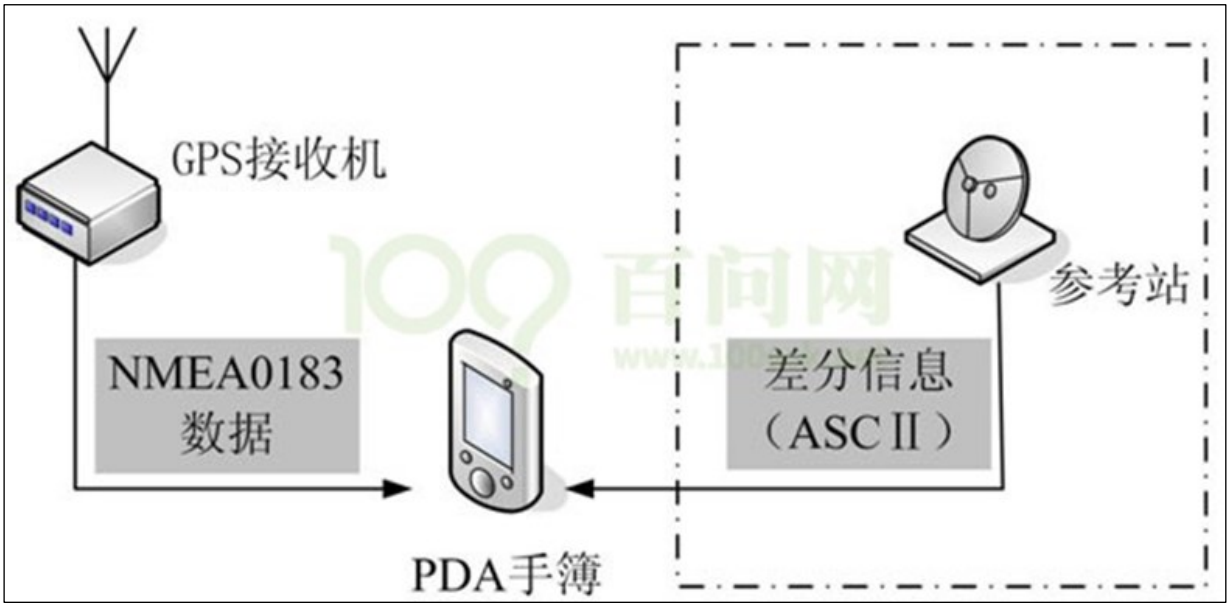


图 11.4 NVMEA0183 格式

我们使用串口接收数据，收到的数据包含：$GPGGA（GPS 定位数据）、$GPGLL（地理定位信息）、$GPGSA（当前卫星信息）、$GPGSV（可见卫星状态信息）、$GPRMC（推荐最小定位信息）、$GPVTG（地面速度信息）。

这里我们只分析$GPGGA (Global Positioning System Fix Data)即可，它包含了 GPS 定位经纬度、质量因子、HDOP、高程、参考站号等字段。其标准格式如下：

$GPGGA，<1>，<2>，<3>，<4>，<5>，<6>，<7>，<8>，<9>，M，<10>，M，<11>，<12>*hh<CR><LF>

$XXGGA 语句各字段的含义和取值范围各字段的含义和取值范围见下表所示，XX 取值有：

- GPGGA：单 GPS
- BDGGA：单北斗
- GLGGA：单 GLONASS
- GNGGA：多星联合定位

表 11-2 $XXGGA 字段

字段	含义	取值范围
<1>	UTC 时间 hhmmss.ss	000000.00~235959.99
<2>	纬度，格式：ddmm.mmmm	000.00000~8959.9999
<3>	南北半球	N 北纬 S 南纬
<4>	经度格式 dddmm.mmmm	00000.0000~17959.9999
<5>	东西半球	E 表示东经 W 表示西经
<6>	GPS 状态	0=未定位 1=GPS 单点定位固定解 2=差分定位 3=PPS 解 4=RTK 固定解 5=RTK 浮点解 6=估计值 7=手工输入模式 8=模拟模式
<7>	应用解算位置的卫星数	00~12
<8>	HDOP 水平图形强度因子	0.500~99.000（大于 6 不可用)
<9>	海拔高度	-9999.9~99999.9
<10>	地球椭球面相对大地水准面的高度（高程异常）	-9999.9~99999.9
<11>	差分时间	从最近一次接收到差分信号开始的秒数，如果不是差分定位将为空
<12>	参考站号	0000~1023；不使用 DGPS 时为空

例子：$GPGGA，074529.82，2429.6717，N，11804.6973，E，1，8，1.098，42.110，，，M，，*76。

11.2.3 编程

参考例程：

01_all_series_quickstart
04_嵌入式 Linux 应用开发基础知识\source\14_UART\02_gps\gps_read.c

11.2.4 接线

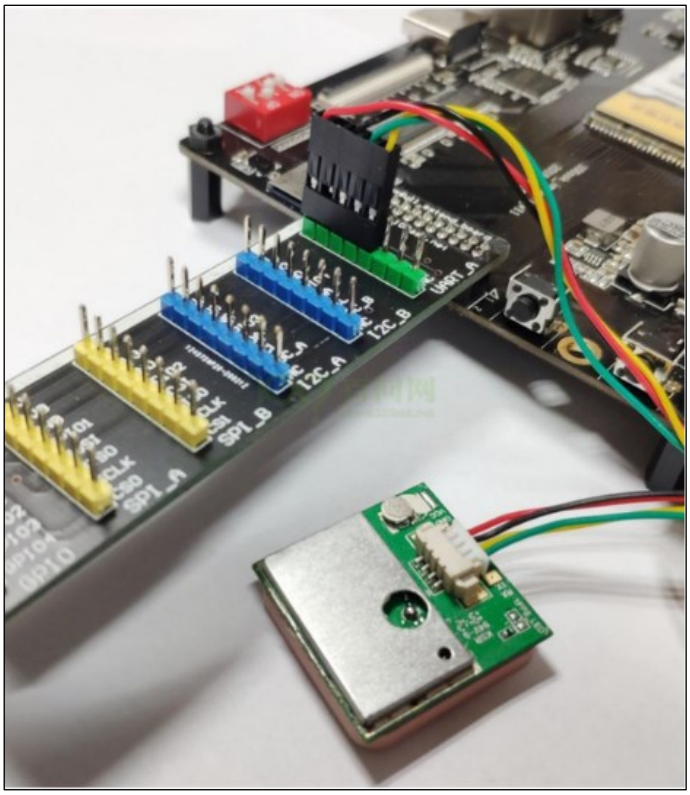


图 11.5 GPS 模块硬件连接

11.2.5 上机实验

设置工具链：

```
export ARCH=arm
export CROSS_COMPILE=arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

编译、执行程序：

```
1. Ubuntu 上
arm-buildroot-linux-gnueabihf-gcc -o gps_read gps_read.c
2. 板子上
/mnt/gps_read  /dev/ttymxc5
```

>>>>>>>第五篇 嵌入式 Linux 驱动开发基础知识<<<<<<<

具体操作视频链接：https://www.bilibili.com/video/BV14f4y1Q7ti

第1章 Hello 驱动(不涉及硬件操作)

我们选用的内核都是 4.x 版本，操作都是类似的：

- rk3399　linux 4.4.154
- rk3288　linux 4.4.154
- imx6ul　linux 4.9.88
- am3358　linux 4.9.168

1.1 APP 打开的文件在内核中如何表示

APP 打开文件时，可以得到一个整数，这个整数被称为文件句柄。对于 APP 的每一个文件句柄，在内核里面都有一个“struct file”与之对应。

```
894: struct file {
895:     union {
896:         struct llist_node   fu_llist;
897:         struct rcu_head     fu_rcuhead;
898:     } f_u;
899:     struct path     f_path;
900:     struct inode        *f_inode;   /* cached value */
901:     const struct file_operations    *f_op;
902:
903:     /*
904:      * Protects f_ep_links, f_flags.
905:      * Must not be taken from IRQ context.
906:      */
907:     spinlock_t      f_lock;
908:     atomic_long_t       f_count;
909:     unsigned int        f_flags;
910:     fmode_t         f_mode;
911:     struct mutex        f_pos_lock;
912:     loff_t          f_pos;
913:     struct fown_struct  f_owner;
914:     const struct cred   *f_cred;
915:     struct file_ra_state    f_ra;
916:
```

图 1.1 struct file

可以猜测，我们使用 open 打开文件时，传入的 flags、mode 等参数会被记录在内核中对应的 struct file 结构体里(f_flags、f_mode)：

```
int open(const char *pathname, int flags, mode_t mode);
```

去读写文件时，文件的当前偏移地址也会保存在 struct file 结构体的 f_pos 成员里。

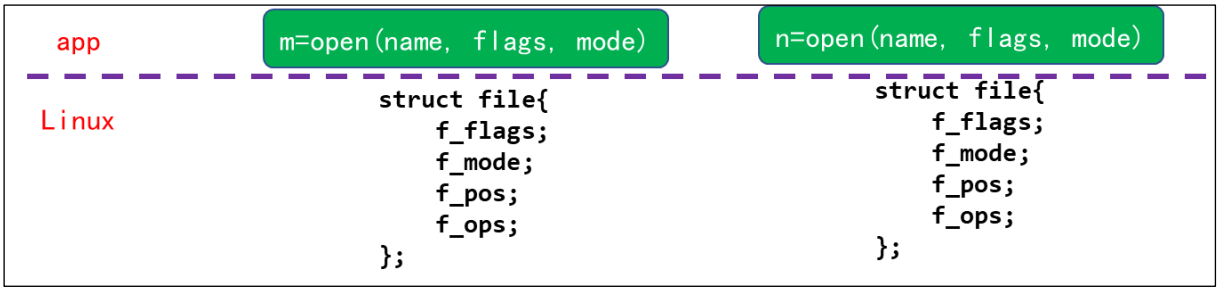


图 1.2 open->struct file

1.2 打开字符设备节点时，内核中也有对应的 struct file

注意这个结构体中的结构体：struct file_operations *f_op，这是由驱动程序提供的。

```
894: struct file {
895:     union {
896:         struct llist_node    fu_llist;
897:         struct rcu_head      fu_rcuhead;
898:     } f_u;
899:     struct path      f_path;
900:     struct inode         *f_inode;    /* cached value */
901:     const struct file_operations    *f_op;
902:
```

图 1.3 驱动程序的 struct file

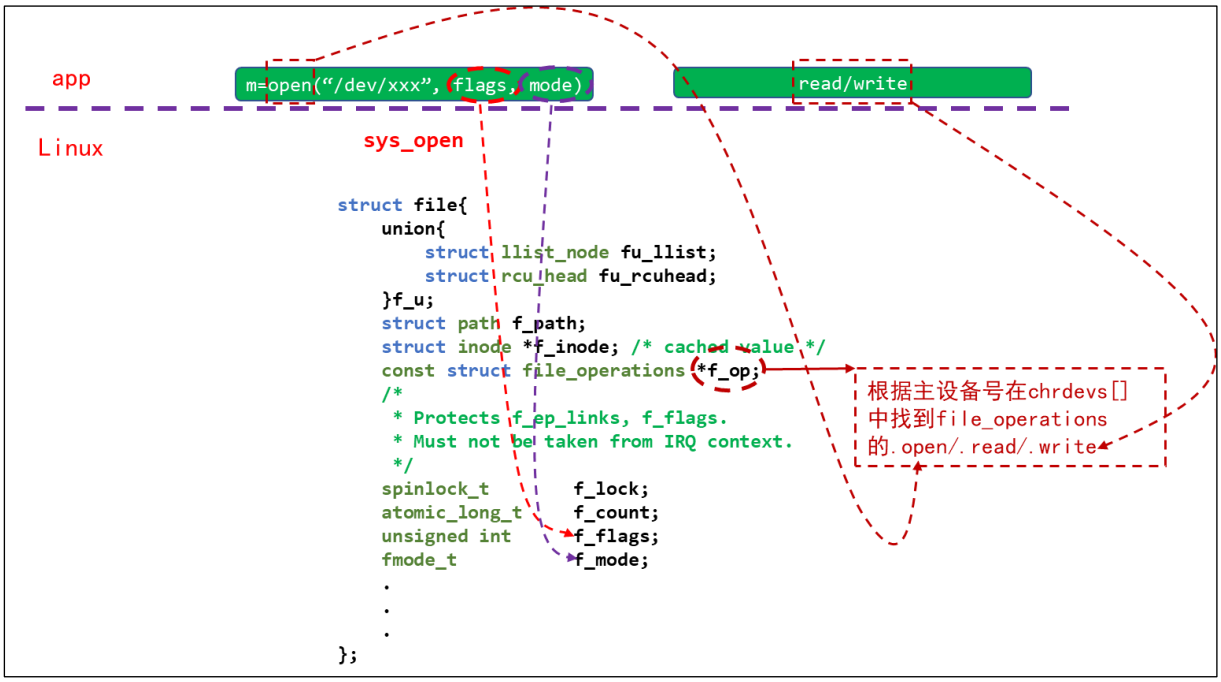


图 1.4 驱动程序的 open/read/write

结构体 struct file_operations 的定义如下：

```
1674: struct file_operations {
1675:     struct module *owner;
1676:     loff_t (*llseek) (struct file *, loff_t, int);
1677:     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1678:     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1679:     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1680:     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1681:     int (*iterate) (struct file *, struct dir_context *);
1682:     unsigned int (*poll) (struct file *, struct poll_table_struct *);
1683:     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1684:     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1685:     int (*mmap) (struct file *, struct vm_area_struct *);
1686:     int (*open) (struct inode *, struct file *);
1687:     int (*flush) (struct file *, fl_owner_t id);
1688:     int (*release) (struct inode *, struct file *);
1689:     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
```

图 1.5 struct file_operations 的定义

1.3 请猜猜怎么编写驱动程序

① 确定主设备号，也可以让内核分配
② 定义自己的 file_operations 结构体
③ 实现对应的 drv_open/drv_read/drv_write 等函数，填入 file_operations 结构体
④ 把 file_operations 结构体告诉内核：register_chrdev
⑤ 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数
⑥ 有入口函数就应该有出口函数：卸载驱动程序时，出口函数调用 unregister_chrdev
⑦ 其他完善：提供设备信息，自动创建设备节点：class_create, device_create

1.4 编写代码

1.4.1 写驱动程序

参考 driver/char 中的程序，包含头文件，写框架，传输数据：

- 驱动中实现 open，read，write，release，APP 调用这些函数时，都打印内核信息
- APP 调用 write 函数时，传入的数据保存在驱动中
- APP 调用 read 函数时，把驱动中保存的数据返回给 APP

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\01_hello_drv\hello_drv.c
```

hello_drv.c 源码如下：

```
01 #include <linux/module.h>
02
03 #include <linux/fs.h>
04 #include <linux/errno.h>
05 #include <linux/miscdevice.h>
06 #include <linux/kernel.h>
07 #include <linux/major.h>
08 #include <linux/mutex.h>
09 #include <linux/proc_fs.h>
10 #include <linux/seq_file.h>
11 #include <linux/stat.h>
12 #include <linux/init.h>
13 #include <linux/device.h>
14 #include <linux/tty.h>
15 #include <linux/kmod.h>
16 #include <linux/gfp.h>
17
18 /* 1. 确定主设备号 */
19 static int major = 0;
```

```
20 static char kernel_buf[1024];
21 static struct class *hello_class;
22
23
24 #define MIN(a, b) (a < b ? a : b)
25
26 /* 3. 实现对应的 open/read/write 等函数，填入 file_operations 结构体 */
27 static ssize_t hello_drv_read (struct file *file, char __user *buf, size_t size, l
off_t *offset)
28 {
29     int err;
30     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
31     err = copy_to_user(buf, kernel_buf, MIN(1024, size));
32     return MIN(1024, size);
33 }
34
35 static ssize_t hello_drv_write (struct file *file, const char __user *buf, size_t
size, loff_t *offset)
36 {
37     int err;
38     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
39     err = copy_from_user(kernel_buf, buf, MIN(1024, size));
40     return MIN(1024, size);
41 }
42
43 static int hello_drv_open (struct inode *node, struct file *file)
44 {
45     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
46     return 0;
47 }
48
49 static int hello_drv_close (struct inode *node, struct file *file)
50 {
51     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
52     return 0;
53 }
54
55 /* 2. 定义自己的 file_operations 结构体 */
56 static struct file_operations hello_drv = {
57     .owner   = THIS_MODULE,
58     .open    = hello_drv_open,
59     .read    = hello_drv_read,
60     .write   = hello_drv_write,
61     .release = hello_drv_close,
62 };
63
64 /* 4. 把 file_operations 结构体告诉内核：注册驱动程序 */
65 /* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数 */
66 static int __init hello_init(void)
67 {
68     int err;
69
70     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
71     major = register_chrdev(0, "hello", &hello_drv);  /* /dev/hello */
72
73
```

```
74      hello_class = class_create(THIS_MODULE, "hello_class");
75      err = PTR_ERR(hello_class);
76      if (IS_ERR(hello_class)) {
77              printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
78              unregister_chrdev(major, "hello");
79              return -1;
80      }
81
82      device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello"); /* /dev/hel
lo */
83
84      return 0;
85 }
86
87 /* 6. 有入口函数就有出口函数：卸载驱动程序时就会去调用这个出口函数 */
88 static void __exit hello_exit(void)
89 {
90      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
91      device_destroy(hello_class, MKDEV(major, 0));
92      class_destroy(hello_class);
93      unregister_chrdev(major, "hello");
94 }
95
96
97 /* 7. 其他完善：提供设备信息，自动创建设备节点 */
98
99 module_init(hello_init);
100 module_exit(hello_exit);
101
102 MODULE_LICENSE("GPL");
103
```

阅读一个驱动程序，从它的入口函数开始，第 66 行就是入口函数。它的主要工作就是第 71 行，向内核注册一个 file_operations 结构体：hello_drv，这就是字符设备驱动程序的核心。

file_operations 结构体 hello_drv 在第 56 行定义，里面提供了 open/read/write/release 成员，应用程序调用 open/read/write/close 时就会导致这些成员函数被调用。

file_operations 结构体 hello_drv 中的成员函数都比较简单，大多数只是打印而已。要注意的是，驱动程序和应用程序之间传递数据要使用 copy_from_user/copy_to_user 函数。

1.4.2 写测试程序

测试程序要实现写、读功能：

```
./hello_drv_test  -w  www.100ask.net  // 把字符串“www.100ask.net”发给驱动程序
./hello_drv_test  -r                  // 把驱动中保存的字符串读回来
```

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\01_hello_drv\hello_drv_test.c
```

hello_drv_test.c 源码如下：

```
02 #include <sys/types.h>
03 #include <sys/stat.h>
04 #include <fcntl.h>
05 #include <unistd.h>
06 #include <stdio.h>
07 #include <string.h>
08
09 /*
10  * ./hello_drv_test -w abc
11  * ./hello_drv_test -r
12  */
13 int main(int argc, char **argv)
14 {
15     int fd;
16     char buf[1024];
17     int len;
18
19     /* 1. 判断参数 */
20     if (argc < 2)
21     {
22             printf("Usage: %s -w <string>\n", argv[0]);
23             printf("       %s -r\n", argv[0]);
24             return -1;
25     }
26
27     /* 2. 打开文件 */
28     fd = open("/dev/hello", O_RDWR);
29     if (fd == -1)
30     {
31             printf("can not open file /dev/hello\n");
32             return -1;
33     }
34
35     /* 3. 写文件或读文件 */
36     if ((0 == strcmp(argv[1], "-w")) && (argc == 3))
37     {
38             len = strlen(argv[2]) + 1;
39             len = len < 1024 ? len : 1024;
40             write(fd, argv[2], len);
41     }
42     else
43     {
44             len = read(fd, buf, 1024);
45             buf[1023] = '\0';
46             printf("APP read : %s\n", buf);
47     }
48
49     close(fd);
50
51     return 0;
52 }
```

1.4.3 测试

- 编写驱动程序的 Makefile

驱动程序中包含了很多头文件，这些头文件来自内核，不同的 ARM 板它的某些头文件可能不同。所以编译驱动程序时，需要指定板子所用的内核的源码路径。

要编译哪个文件？这也需要指定，设置 obj-m 变量即可

怎么把.c 文件编译为驱动程序.ko？这要借助内核的顶层 Makefile。

本驱动程序的 Makefile 内容如下：

```
# 1. 使用不同的开发板内核时，一定要修改 KERN_DIR
# 2. KERN_DIR 中的内核要事先配置、编译，为了能编译内核，要先设置下列环境变量:
# 2.1 ARCH,          比如: export ARCH=arm64
# 2.2 CROSS_COMPILE, 比如: export CROSS_COMPILE=aarch64-linux-gnu-
# 2.3 PATH,          比如: export PATH=$PATH:/home/book/100ask_roc-rk3399-pc/ToolChain-6.3.1/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin
# 注意: 不同的开发板不同的编译器上述 3 个环境变量不一定相同,
#       请参考各开发板的高级用户使用手册

KERN_DIR = /home/book/100ask_roc-rk3399-pc/linux-4.4

all:
	make -C $(KERN_DIR) M=`pwd` modules
	$(CROSS_COMPILE)gcc -o hello_drv_test hello_drv_test.c

clean:
	make -C $(KERN_DIR) M=`pwd` modules clean
	rm -rf modules.order
	rm -f hello_drv_test

obj-m	+= hello_drv.o
```

先设置好交叉编译工具链，编译好你的板子所用的内核，然后修改 Makefile 指定内核源码路径，最后即可执行 make 命令编译驱动程序和测试程序。

- 上机实验

注意：我们是在 Ubuntu 中编译程序，但是需要在 ARM 板子上测试。所以需要把程序放到 ARM 板子上。

启动单板后，可以通过 NFS 挂载 Ubuntu 的某个目录，访问该目录中的程序。

- 测试示例：
  - 在 Ubuntu 上编译好驱动，并它复制到 NFS 目录：

```
cp *.ko hello_drv_test ~/nfs_rootfs/
```

  - 在 ARM 板上测试：

```
[root@100ask:~]# echo "7 4 1 7" > /proc/sys/kernel/printk  // 打开内核的打印信息，有些板子默认打开了
[root@100ask:~]# ifconfig eth0 192.168.5.9   // 配置 ARM 板 IP，下面是挂载 NFS 文件系统
// 2.如果使用 VMware 桥接网络，假设 Ubuntu IP 为 192.168.5.11，使用下面命令挂载 NFS
[root@100ask:~]# mount -t nfs -o nolock,vers=3  192.168.5.11:/home/book/nfs_rootfs  /mnt
[root@100ask:~]# cd  /mnt
[root@100ask:~]# insmod hello_drv.ko    // 安装驱动程序
[  293.594910] hello_drv: loading out-of-tree module taints kernel.
[  293.616051] /home/book/source/01_hello_drv/hello_drv.c hello_init line 70
[root@100ask:~]# ls /dev/hello -l        // 驱动程序会生成设备节点
crw-------    1 root     root      236,   0 Jan 18 08:55 /dev/hello
[root@100ask:~]#./hello_drv_test        // 查看测试程序的用法
```

```
Usage: ./hello_drv_test -w <string>
       ./hello_drv_test -r
[root@100ask:~]#./hello_drv_test -w www.100ask.net    // 往驱动程序中写入字符串
[  318.360800] /home/book/source/01_hello_drv/hello_drv.c hello_drv_open line 45
[  318.372570] /home/book/source/01_hello_drv/hello_drv.c hello_drv_write line 38
[  318.382854] /home/book/source/01_hello_drv/hello_drv.c hello_drv_close line 51
[root@100ask:~]#./hello_drv_test -r                // 从驱动程序中读出字符串
[  326.177890] /home/book/source/01_hello_drv/hello_drv.c hello_drv_open line 45
[  326.198304] /home/book/source/01_hello_drv/hello_drv.c hello_drv_read line 30
[root@100ask:~]# APP read : www.100ask.net
[  326.214782] /home/book/source/01_hello_drv/hello_drv.c hello_drv_close line 51
```

注意：如果安装驱动时提示 version magic 不匹配，或是污染内核(taint)，请参考这些章节更新内核：《>>>>>>>>第三篇第 5 章开发板的第 1 个驱动实验》。

1.5 Hello 驱动中的一些补充知识

1.5.1 module_init/module_exit 的实现

一个驱动程序有入口函数、出口函数，代码如下：

```
module_init(hello_init);
module_exit(hello_exit);
```

驱动程序可以被编进内核里，也可以被编译为 ko 文件后手工加载。对于这两种形式，“module_init/module_exit”这 2 个宏是不一样的。在内核文件“include\linux\module.h”中可以看到这 2 个宏：

```
01 #ifndef MODULE
02
03 #define module_init(x)    __initcall(x);
04 #define module_exit(x)    __exitcall(x);
05
06 #else /* MODULE */
07
08 #define module_init(initfn)                \
09
10 /* Each module must use one module_init(). */
11     static inline initcall_t __inittest(void)      \
12     { return initfn; }                \
13     int init_module(void) __attribute__((alias(#initfn)));
14
15 /* This is only required if you want to be unloadable. */
16 #define module_exit(exitfn)                \
17     static inline exitcall_t __exittest(void)      \
18     { return exitfn; }                \
19     void cleanup_module(void) __attribute__((alias(#exitfn)));
20
21 #endif
```

编译驱动程序时，我们执行“make modules”这样的命令，它在编译 c 文件时会定义宏 MODULE，比如：

```
arm-buildroot-linux-gnueabihf-gcc  -DMODULE  -c -o hello_drv.o hello_drv.c
```

在编译内核时，并不会定义宏 MODULE。所以，“module_init/module_exit”

这 2 个宏在驱动程序被编进内核时，如上面代码中第 3、4 行那样定义；在驱动程序被编译为 ko 文件时，如上面代码中第 11~19 行那样定义。

把上述代码里的宏全部展开后，得到如下代码：

```
01 #ifndef MODULE
01
01 #define module_init(fn)
02     static initcall_t __initcall_##fn##id __used \
03 __attribute__((__section__(".initcall" #6 ".init"))) = fn;
04
05 #define module_exit(x)    static exitcall_t __exitcall_##x __used __section(.exitc
all.exit) = x;
06
07 #else /* MODULE */
08
09 #define module_init(initfn)                \
10
11 /* Each module must use one module_init(). */
12     static inline initcall_t __inittest(void)       \
13     { return initfn; }                  \
14     int init_module(void) __attribute__((alias(#initfn)));
15
16 /* This is only required if you want to be unloadable. */
17 #define module_exit(exitfn)                \
18     static inline exitcall_t __exittest(void)       \
19     { return exitfn; }                  \
20     void cleanup_module(void) __attribute__((alias(#exitfn)));
21
22 #endif
```

驱动程序被编进内核时，把“module_init(hello_init)”、“module_exit(hello_exit)”展开，得到如下代码：

```
static initcall_t __initcall_hello_init6 __used \
__attribute__((__section__(".initcall6.init"))) = hello_init;

static exitcall_t __exitcall_hello_exit  __used __section(.exitcall.exit) = hello_exi
t;
```

其中的“initcall_t”、“exitcall_t”就是函数指针类型，所以上述代码就是定义了两个函数指针：第 1 个函数指针名为__initcall_hello_init6，放在段".initcall6.init"里；第 2 个函数指针名为__exitcall_hello_exit，放在段“.exitcall.exit”里。

内核启动时，会去段".initcall6.init"里取出这些函数指针来执行，所以驱动程序的入口函数就被执行了。

一个驱动被编进内核后，它是不会被卸载的，所以段“.exitcall.exit”不会被用到，内核启动后会释放这块段空间。

驱动程序被编译为 ko 文件时，把“module_init(hello_init)”、“module_exit(hello_exit)”展开，得到如下代码：

```
static inline initcall_t __inittest(void)       \
{ return hello_init; }                  \
int init_module(void) __attribute__((alias("hello_init")));
```

```
static inline exitcall_t __exittest(void)      \
{ return hello_exit; }                  \
void cleanup_module(void) __attribute__((alias("hello_exit")));
```

分别定义了 2 个函数：第 1 个函数名为 init_module，它是 hello_init 函数的别名；第 2 个函数名为 cleanup_module，它是 hello_exit 函数的别名。

以后我们使用 insmod 命令加载驱动时，内核都是调用 init_module 函数，实际上就是调用 hello_init 函数；使用 rmmod 命令卸载驱动时，内核都是调用 cleanup_module 函数，实际上就是调用 hello_exit 函数。

1.5.2 register_chrdev 的内部实现

register_chrdev 函数源码如下：

```
static inline int register_chrdev(unsigned int major, const char *name,
                  const struct file_operations *fops)
{
    return __register_chrdev(major, 0, 256, name, fops);
}
```

它调用__register_chrdev 函数，这个函数的代码精简如下：

```
01 int __register_chrdev(unsigned int major, unsigned int baseminor,
02              unsigned int count, const char *name,
03              const struct file_operations *fops)
04 {
05     struct char_device_struct *cd;
06     struct cdev *cdev;
07     int err = -ENOMEM;
08
09     cd = __register_chrdev_region(major, baseminor, count, name);
10
11     cdev = cdev_alloc();
12
13     cdev->owner = fops->owner;
14     cdev->ops = fops;
15     kobject_set_name(&cdev->kobj, "%s", name);
16
17     err = cdev_add(cdev, MKDEV(cd->major, baseminor), count);
18 }
```

这个函数主要的代码是第 09 行、第 11~15 行、第 17 行。

第 09 行，调用__register_chrdev_region 函数来“注册字符设备的区域”，它仅仅是查看设备号(major, baseminor)到(major, baseminor+count-1)有没有被占用，如果未被占用的话，就使用这块区域。

在前面的课程里，在引入驱动程序时为了便于理解，我们说内核里有一个 chrdevs 数组，根据主设备号 major 在 chrdevs[major]中放入 file_operations 结构体，以后 open/read/write 某个设备文件时，就是根据主设备号从 chrdevs[major]中取出 file_operations 结构体，调用里面的 open/read/write 函数指针。

上述说法并不准确，内核中确实有一个 chrdevs 数组：

```
static struct char_device_struct {
    struct char_device_struct *next;
```

```
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    char name[64];
    struct cdev *cdev;      /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

去访问它的时候，并不是直接使用主设备号 major 来确定数组项，而是使用如下函数来确定数组项：

```
/* index in the above */
static inline int major_to_index(unsigned major)
{
    return major % CHRDEV_MAJOR_HASH_SIZE;
}
```

上述代码中，CHRDEV_MAJOR_HASH_SIZE 等于 255。比如主设备号 1、256，都会使用 chardevs[1]。chardevs[1]是一个链表，链表里有多个 char_device_struct 结构体，某个结构体表示主设备号为 1 的设备，某个结构体表示主设备号为 256 的设备。

chardevs 的结构图如图 2.6 所示：

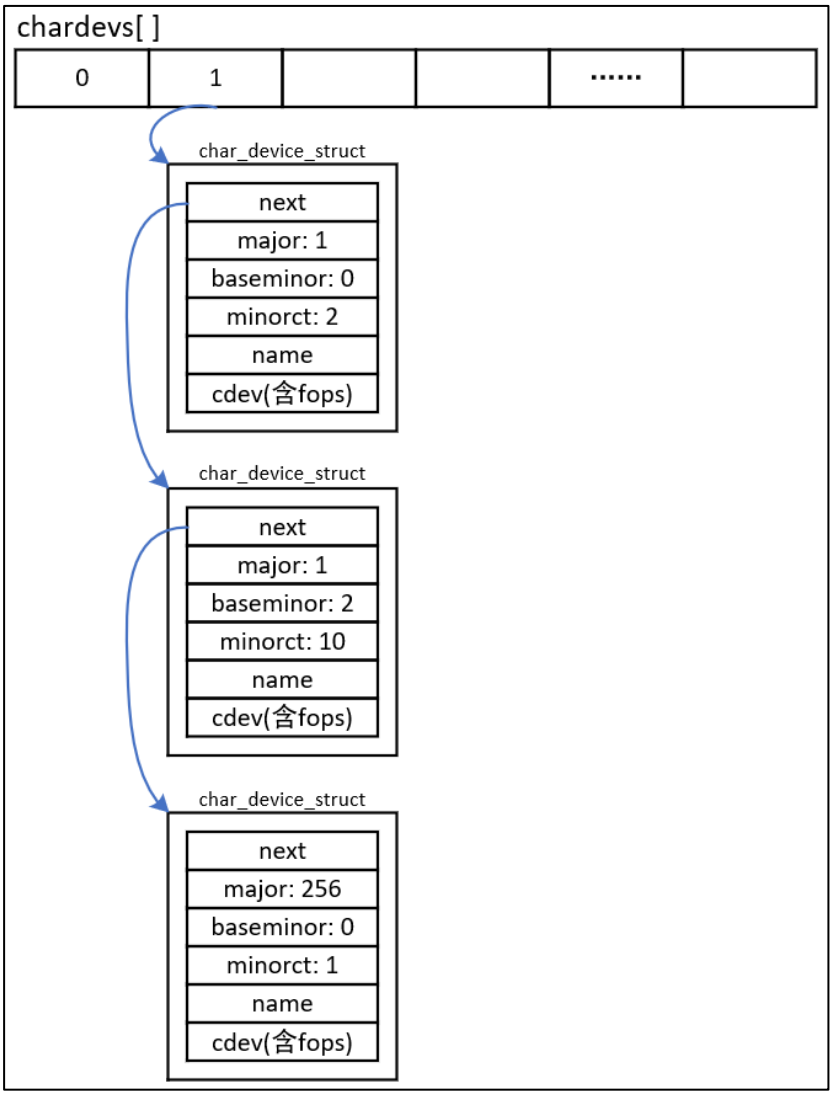


图 1.6 chardevs 结构图

从图 2.6 可以得出如下结论：

① chrdevs[i]数组项是一个链表头

链表里每一个元素都是一个 char_device_struct 结构体，每个元素表示一个驱动程序。

char_device_struct 结构体内容如下：

```
struct char_device_struct {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    char name[64];
    struct cdev *cdev;      /* will die */
}
```

它指定了主设备号 major、次设备号 baseminor、个数 minorct，在 cdev 中含有 file_operations 结构体。

char_device_struct 结构体的含义是：主次设备号为(major, baseminor)、(major, baseminor+1)、(major, baseminor+2)、(major, baseminor+ minorct-1)的这些设备，都使用同一个 file_operations 来操作。

以前为了更容易理解驱动程序时，说“内核通过主设备号找到对应的 file_operations 结构体”，这并不准确。应该改成：“内核通过主、次设备号，找到对应的 file_operations 结构体”。

② 在图 2.6 中，chardevs[1]中有 3 个驱动程序

第 1 个 char_device_struct 结构体对应主次设备号(1, 0)、(1, 1)，这是第 1 个驱动程序。

第 2 个 char_device_struct 结构体对应主次设备号(1, 2)、(1, 2)、……、(1, 11)，这是第 2 个驱动程序。

第 3 个 char_device_struct 结构体对应主次设备号(256, 0)，这是第 3 个驱动程序。

第 11~15 行分配一个 cdev 结构体，并设置它：它含有 file_operations 结构体。

第 17 行调用 cdev_add 把 cdev 结构体注册进内核里，cdev_add 函数代码如下：

```
01 int cdev_add(struct cdev *p, dev_t dev, unsigned count)
02 {
03     int error;
04
05     p->dev = dev;
06     p->count = count;
07
08     error = kobj_map(cdev_map, dev, count, NULL,
09              exact_match, exact_lock, p);
10     if (error)
```

```
11          return error;
12
13      kobject_get(p->kobj.parent);
14
15      return 0;
16  }
```

这个函数涉及 kobj 的操作，这是一个通用的链表操作函数。它的作用是：把 cdev 结构体放入 cdev_map 链表中，对应的索引值是“dev”到“dev+count-1”。以后可以从 cdev_map 链表中快速地使用索引值取出对应的 cdev。

比如执行以下代码：

```
err = cdev_add(cdev, MKDEV(1, 2), 10);
```

其中的 MKDEV(1,2)构造出一个整数“1<<8 | 2”，即 0x102；上述代码将 cdev 放入 cdev_map 链表中，对应的索引值是 0x102 到 0x10c(即 0x102+10)。以后根据这 10 个数值(0x102、0x103、0x104、……、0x10c)中任意一个，都可以快速地从 cdev_map 链表中取出 cdev 结构体。

APP 打开某个字符设备节点时，进入内核。在内核里根据字符设备节点的主、次设备号，计算出一个数值(major<<8 | minor，即 inode->i_rdev)，然后使用这个数值从 cdev_map 中快速得到 cdev，再从 cdev 中得到 file_operations 结构体。

关键函数如下：

```
348: /*
349:  * Called every time a character special file is opened
350:  */
351: static int chrdev_open(struct inode *inode, struct file *filp)
352: {
353:     const struct file_operations *fops;
354:     struct cdev *p;
355:     struct cdev *new = NULL;
356:     int ret = 0;
357:
358:     spin_lock(&cdev_lock);
359:     p = inode->i_cdev;
360:     if (!p) {
361:         struct kobject *kobj;
362:         int idx;
363:         spin_unlock(&cdev_lock); 1.根据设备号从cdev_map得到kobj
364:         kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
365:         if (!kobj)
366:             return -ENXIO;          2.把kobj转换为cdev
367:         new = container_of(kobj, struct cdev, kobj);
368:         spin_lock(&cdev_lock);
369:         /* Check i_cdev again in case somebody beat us to it while
370:            we dropped the lock. */
```

图 1.7 找到驱动程序的关键函数

在打开文件的过程中，可以看到并未涉及 chrdevs，都是使用 cdev_map。所以可以看到在 chrdevs 的定义中看到如下注释：

```
31: static struct char_device_struct {
32:     struct char_device_struct *next;
33:     unsigned int major;
34:     unsigned int baseminor;
35:     int minorct;
36:     char name[64];
37:     struct cdev *cdev;
38: } *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
39:
```

以后可以去掉它

/* will die */

图 1.8 chardevs 中 cdev 的作用并不大

1.5.3 class_destroy/device_create 浅析

驱动程序的核心是 file_operations 结构体：分配、设置、注册它。“class_destroy/device_create”函数知识起一些辅助作用：在/sys 目录下创建一些目录、文件，这样 Linux 系统中的 APP(比如 udev、mdev)就可以根据这些目录或文件来创建设备节点。

以下代码将会在“/sys/class”目录下创建一个子目录“hello_class”：

```
hello_class = class_create(THIS_MODULE, "hello_class");
```

以下代码将会在“/sys/class/hello_class”目录下创建一个文件“hello”：

```
device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello");
```

更详细的信息请看图 2.9：

```
[root@100ask:/sys/class/hello_class]# ls -l 1.在/sys/class目录下创建了一个hello_class
total 0                                  2. 在hello_class下有链接文件hello，表示设备
lrwxrwxrwx 1 root root 0 Aug  2 03:53 hello -> ../../devices/virtual/hello_class/hello
[root@100ask:/sys/class/hello_class]#
[root@100ask:/sys/class/hello_class]# ls -l ../../devices/virtual/hello_class/hello
total 0                                                  3.这个目录下有更详细的设备信息
-r--r--r-- 1 root root 4096 Aug  2 03:56 dev
drwxr-xr-x 2 root root    0 Aug  2 03:56 power
lrwxrwxrwx 1 root root    0 Aug  2 03:56 subsystem -> ../../../../class/hello_class
-rw-r--r-- 1 root root 4096 Aug  2 03:52 uevent   4.比如dev文件里含有主、次设备号，
[root@100ask:/sys/class/hello_class]#                APP根据它创建设备节点
[root@100ask:/sys/class/hello_class]# cat ../../devices/virtual/hello_class/hello/dev
240:0 主、次设备号
```

图 1.9 class 和 device

第2章 硬件知识_LED 原理图

当我们学习 C 语言的时候，我们会写个 Hello 程序。

那当我们写 ARM 程序，也该有一个简单的程序引领我们入门，这个程序就是点亮 LED。

我们怎样去点亮一个 LED 呢？分为三步：

第1步 看原理图，确定控制 LED 的引脚；

第2步 看主芯片的芯片手册，确定如何设置控制这个引脚；

第3步 写程序；

2.1 先来讲讲怎么看原理图

LED 样子有很多种，像插脚的，贴片的。

图 2.1 市面 LED 产品

它们长得完全不一样，因此我们在原理图中将它抽象出来。

点亮 LED 需要通电源，同时为了保护 LED，加个电阻减小电流。

控制 LED 灯的亮灭，可以手动开关 LED，但在电子系统中，不可能让人来控制开关，通过编程，利用芯片的引脚去控制开关。

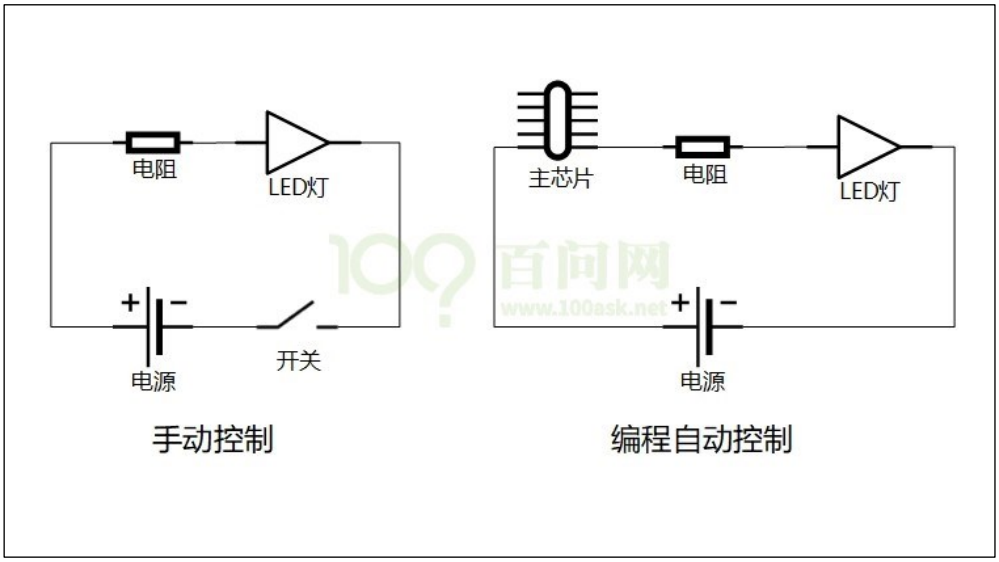


图 2.2 LED 控制电路示意图

LED 的驱动方式，常见的有四种。

① 使用引脚输出 3.3V 点亮 LED，输出 0V 熄灭 LED。

② 使用引脚拉低到 0V 点亮 LED，输出 3.3V 熄灭 LED。

有的芯片为了省电等原因，其引脚驱动能力不足，这时可以使用三极管驱动。

③ 使用引脚输出 1.2V 点亮 LED，输出 0V 熄灭 LED。

④ 使用引脚输出 0V 点亮 LED，输出 1.2V 熄灭 LED。

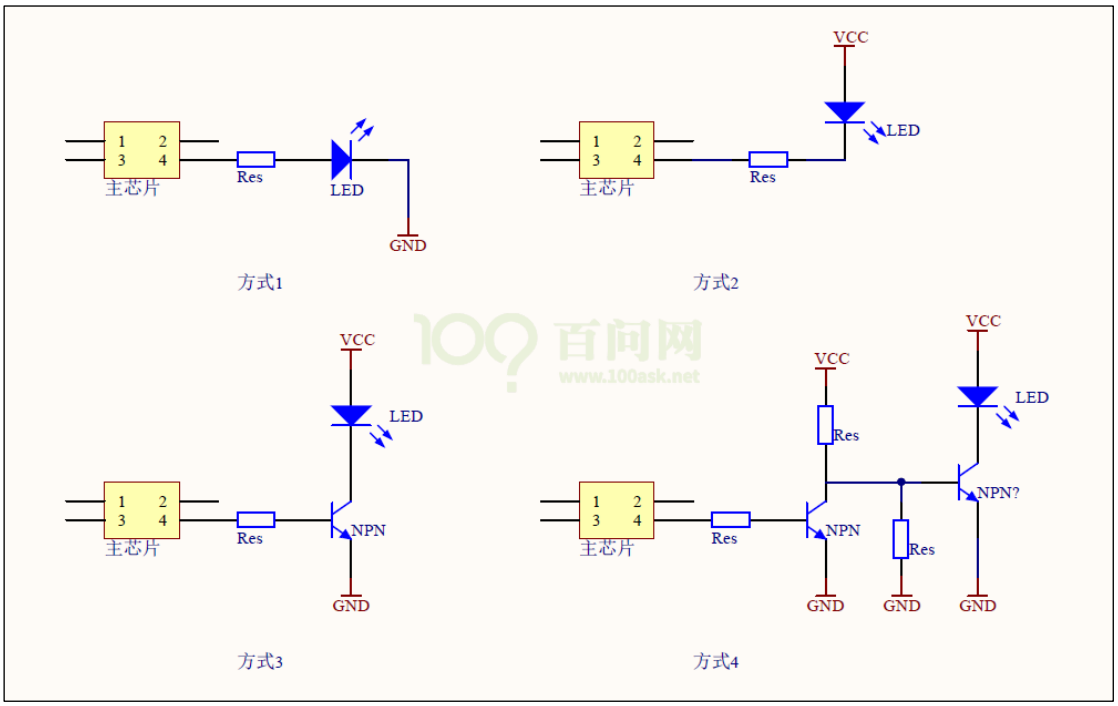


图 2.3 LED 的 4 中驱动方式

由此，主芯片引脚输出高电平/低电平，即可改变 LED 状态，而无需关注 GPIO 引脚输出的是 3.3V 还是 1.2V。所以简称输出 1 或 0：

- 逻辑 1-->高电平
- 逻辑 0-->低电平

第3章 普适的 GPIO 引脚操作方法

GPIO: General-purpose input/output，通用的输入输出口

3.1 GPIO 模块一般结构

- 有多组 GPIO，每组有多个 GPIO
- 使能：电源/时钟
- 模式(Mode)：引脚可用于 GPIO 或其他功能
- 方向：引脚 Mode 设置为 GPIO 时，可以继续设置它是输出引脚，还是输入引脚
- 数值：
  - 对于输出引脚，可以设置寄存器让它输出高、低电平
  - 对于输入引脚，可以读取寄存器得到引脚的当前电平

3.2 GPIO 寄存器操作

- 芯片手册一般有相关章节，用来介绍：power/clock
  - 可以设置对应寄存器使能某个 GPIO 模块(Module)
  - 有些芯片的 GPIO 是没有使能开关的，即它总是使能的
- 一个引脚可以用于 GPIO、串口、USB 或其他功能，
  - 有对应的寄存器来选择引脚的功能
- 对于已经设置为 GPIO 功能的引脚，有方向寄存器用来设置它的方向：输出、输入
- 对于已经设置为 GPIO 功能的引脚，有数据寄存器用来写、读引脚电平状态

GPIO 寄存器的 2 种操作方法：**原则：不能影响到其他位**

① 直接读写：读出、修改对应位、写入

a) 要设置 bit n：

```
val = data_reg;
val = val | (1<<n);
data_reg = val;
```

b) 要清除 bit n：

```
val = data_reg;
val = val & ~(1<<n);
data_reg = val;
```

② set-and-clear protocol：

set_reg，clr_reg，data_reg 三个寄存器对应的是同一个物理寄存器，

a) 要设置 bit n：set_reg = (1<<n);

b) 要清除 bit n：clr_reg = (1<<n);

第4章 具体单板的 GPIO 操作方法

请使用 GIT 下载文档后，看图 4.1 红框所示目录中各板子对应的文档及图片。

网盘中相同名字的目录下也有对应的视频。

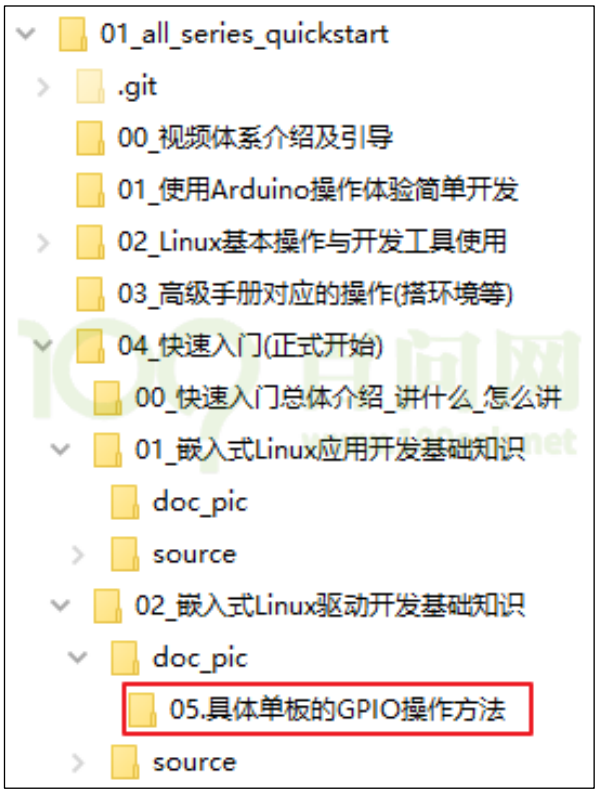


图 4.1 GPIO 操作文档在 git 仓库所在位置

为方便学习，在本文档中也把上述 GIT 目录中的文档添加进来了。

4.1 IMX6ULL 的 GPIO 操作方法

表 4-1 GPIO 操作相关名词解释

名词	解释
CCM	Clock Controller Module (时钟控制模块)
IOMUXC	IOMUX Controller，IO 复用控制器
GPIO	General-purpose input/output，通用的输入输出口

4.1.1 IMX6ULL 的 GPIO 模块结构

参考资料：芯片手册《Chapter 28: General Purpose Input/Output (GPIO)》

有 5 组 GPIO（GPIO1～GPIO5），每组引脚最多有 32 个，但是可能实际上并没有那么多。

- GPIO1 有 32 个引脚：GPIO1_IO0~GPIO1_IO31；
- GPIO2 有 22 个引脚：GPIO2_IO0~GPIO2_IO21；

- GPIO3 有 29 个引脚：GPIO3_IO0~GPIO3_IO28；
- GPIO4 有 29 个引脚：GPIO4_IO0~GPIO4_IO28；
- GPIO5 有 12 个引脚：GPIO5_IO0~GPIO5_IO11；
- GPIO 的控制涉及 4 大模块：CCM、IOMUXC、GPIO 模块本身，框图如图 4.2

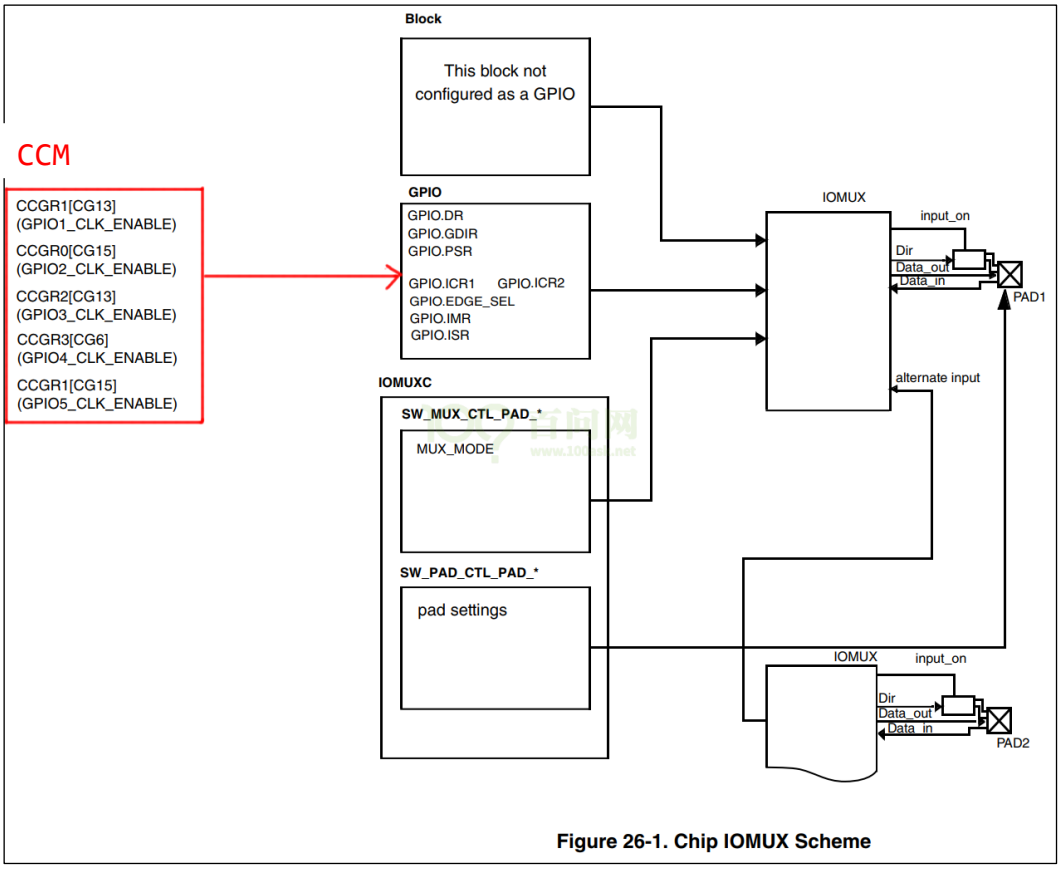


图 4.2 CCM/IOMUXC/GPIO 模块图

4.1.2 CCM 用于设置是否向 GPIO 模块提供时钟

参考资料：芯片手册《Chapter 18: Clock Controller Module (CCM)》

GPIOx 要用 CCM_CCGRy 寄存器中的 2 位来决定该组 GPIO 是否使能。哪组 GPIO 用哪个 CCM_CCGR 寄存器来设置，请看上图红框部分。

CCM_CCGR 寄存器中某 2 位的取值含义如下：

CGR value	Clock Activity Description
00	Clock is off during all modes. Stop enter hardware handshake is disabled.
01	Clock is on in run mode, but off in WAIT and STOP modes
10	Not applicable (Reserved).
11	Clock is on during all modes, except STOP mode.

图 4.3 CCM_CCGR

① 00：该 GPIO 模块全程被关闭

② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭

③ 10：保留

④ 11：该 GPIO 模块全程使能

- GPIO2 时钟控制：

Address: 20C_4000h base + 68h offset = 20C_4068h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR0 field descriptions

Field	Description
31–30 CG15	gpio2_clocks (gpio2_clk_enable)

图 4.4 GPIO2 的时钟控制寄存器

- GPIO1、GPIO5 时钟控制：

Address: 20C_4000h base + 6Ch offset = 20C_406Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR1 field descriptions

Field	Description
31–30 CG15	gpio5 clock (gpio5_clk_enable)
29–28 CG14	csu clock (csu_clk_enable)
27–26 CG13	gpio1 clock (gpio1_clk_enable)

图 4.5 GPIO1/5 的时钟控制寄存器

- GPIO3 时钟控制：

Address: 20C_4000h base + 70h offset = 20C_4070h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR2 field descriptions

Field	Description
31–30 CG15	pxp clocks (pxp_clk_enable)
29–28 CG14	lcd clocks (lcd_clk_enable)
27–26 CG13	gpio3 clock (gpio3_clk_enable)

图 4.6 GPIO3 的时钟控制寄存器

- GPIO4 时钟控制：

Address: 20C_4000h base + 74h offset = 20C_4074h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

13–12 CG6	gpio4 clock (gpio4_clk_enable)

图 4.7 GPIO4 的时钟控制寄存器

4.1.3 IOMUXC：引脚的模式(Mode、功能)

参考资料：芯片手册《Chapter 32: IOMUX Controller (IOMUXC)》。

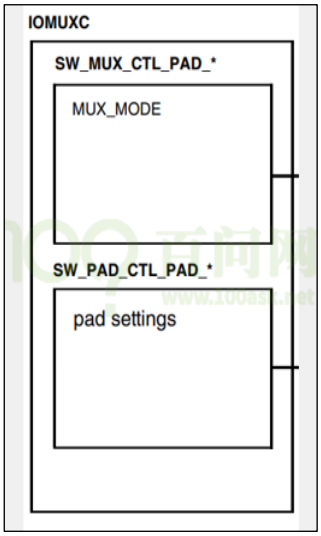


图 4.8 IOMUXC

对于某个/某组引脚，IOMUXC 中有 2 个寄存器用来设置它：

① 选择功能：

a) IOMUXC_SW_MUX_CTL_PAD_<PADNAME> ：Mux pad xxx，选择某个 pad 的功能

b) IOMUXC_SW_MUX_CTL_GRP_<GROUP NAME>：Mux grp xxx，选择某组引脚的功能

某个引脚，或是某组预设的引脚，都有 8 个可选的模式(alternate (ALT) MUX_MODE)。

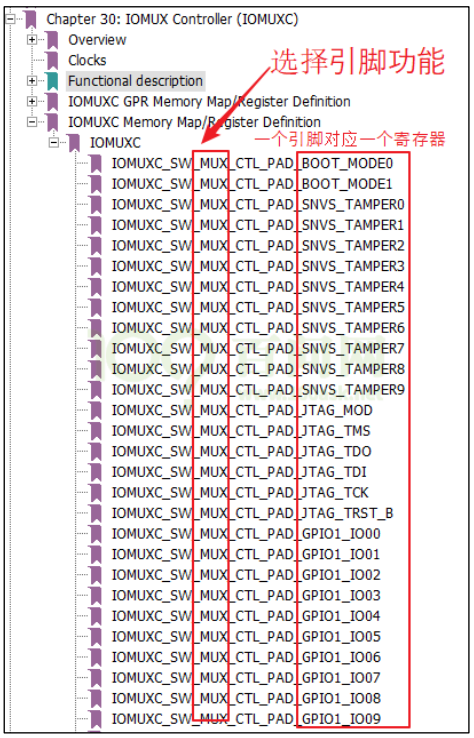


图 4.9 选择引脚功能

比如：

Address: 20E_0000h base + 5Ch offset = 20E_005Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00 field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. loopback回环模式，用于测试 Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad GPIO1_IO00 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	MUX Mode Select Field. Select 1 of 9 iomux modes to be used for pad: GPIO1_IO00. 0000 **ALT0** — Select mux mode: ALT0 mux port: I2C2_SCL of instance: i2c2 0001 **ALT1** — Select mux mode: ALT1 mux port: GPT1_CAPTURE1 of instance: gpt1 0010 **ALT2** — Select mux mode: ALT2 mux port: ANATOP_OTG1_ID of instance: anatop 0011 **ALT3** — Select mux mode: ALT3 mux port: ENET1_REF_CLK1 of instance: enet1 0100 **ALT4** — Select mux mode: ALT4 mux port: MQS_RIGHT of instance: mqs 0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO1_IO00 of instance: gpio1 用作GPIO时 0110 **ALT6** — Select mux mode: ALT6 mux port: ENET1_1588_EVENT0_IN of instance: enet1 0111 **ALT7** — Select mux mode: ALT7 mux port: SRC_SYSTEM_RESET of instance: src 1000 **ALT8** — Select mux mode: ALT8 mux port: WDOG3_WDOG_B of instance: wdog3

图 4.10 SCON 的回环模式

② 设置上下拉电阻等参数：

a) IOMUXC_SW_PAD_CTL_PAD_<PAD_NAME>：pad pad xxx，设置某个 pad 的参数

b) IOMUXC_SW_PAD_CTL_GRP_<GROUP NAME>：pad grp xxx，设置某组引脚的参数

图 4.11 设置引脚参数

比如：

IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 field descriptions

Field	Description
31–17 -	This field is reserved. Reserved
16 HYS	Hyst. Enable Field Select one out of next values for pad: GPIO1_IO00 0 **HYS_0_Hysteresis_Disabled** — Hysteresis Disabled 1 **HYS_1_Hysteresis_Enabled** — Hysteresis Enabled
15–14 PUS	Pull Up / Down Config. Field Select one out of next values for pad: GPIO1_IO00 00 **PUS_0_100K_Ohm_Pull_Down** — 100K Ohm Pull Down 01 **PUS_1_47K_Ohm_Pull_Up** — 47K Ohm Pull Up 10 **PUS_2_100K_Ohm_Pull_Up** — 100K Ohm Pull Up 11 **PUS_3_22K_Ohm_Pull_Up** — 22K Ohm Pull Up
13 PUE	Pull / Keep Select Field Select one out of next values for pad: GPIO1_IO00 0 **PUE_0_Keeper** — Keeper 1 **PUE_1_Pull** — Pull
12 PKE	Pull / Keep Enable Field Select one out of next values for pad: GPIO1_IO00 0 **PKE_0_Pull_Keeper_Disabled** — Pull/Keeper Disabled 1 **PKE_1_Pull_Keeper_Enabled** — Pull/Keeper Enabled
11 ODE	Open Drain Enable Field Select one out of next values for pad: GPIO1_IO00 0 **ODE_0_Open_Drain_Disabled** — Open Drain Disabled 1 **ODE_1_Open_Drain_Enabled** — Open Drain Enabled
10–8 -	This field is reserved. Reserved
7–6 SPEED	Speed Field Select one out of next values for pad: GPIO1_IO00 00 **SPEED_0_low_50MHz_** — low(50MHz) 01 **SPEED_1_medium_100MHz_** — medium(100MHz) 10 **SPEED_2_medium_100MHz_** — medium(100MHz) 11 **SPEED_3_max_200MHz_** — max(200MHz)
5–3 DSE	Drive Strength Field Select one out of next values for pad: GPIO1_IO00 000 **DSE_0_output_driver_disabled_** — output driver disabled; 001 **DSE_1_R0_260_Ohm__3_3V__150_Ohm_1_8V__240_Ohm_for_DDR_** — R0(260 Ohm @ 3.3V, 150 Ohm@1.8V, 240 Ohm for DDR) 010 **DSE_2_R0_2** — R0/2 011 **DSE_3_R0_3** — R0/3 100 **DSE_4_R0_4** — R0/4 101 **DSE_5_R0_5** — R0/5 110 **DSE_6_R0_6** — R0/6 111 **DSE_7_R0_7** — R0/7
2–1 -	This field is reserved. Reserved
0 SRE	Slew Rate Field Select one out of next values for pad: GPIO1_IO00 0 **SRE_0_Slow_Slew_Rate** — Slow Slew Rate 1 **SRE_1_Fast_Slew_Rate** — Fast Slew Rate

①输入方式：施密特触发器还是CMOS输入模式

④上拉或下拉的电阻值是多少?

③使能的是keeper还是pul?

②是否使能?

用于设置输入参数

a.输出方式

c.输出速率

b.驱动能力

用于设置输出参数

d.压摆率/转速速率：0->波形缓和，低干扰;1->高速电路

图 4.12 参数解释

4.1.4 GPIO 模块内部

框图如下：

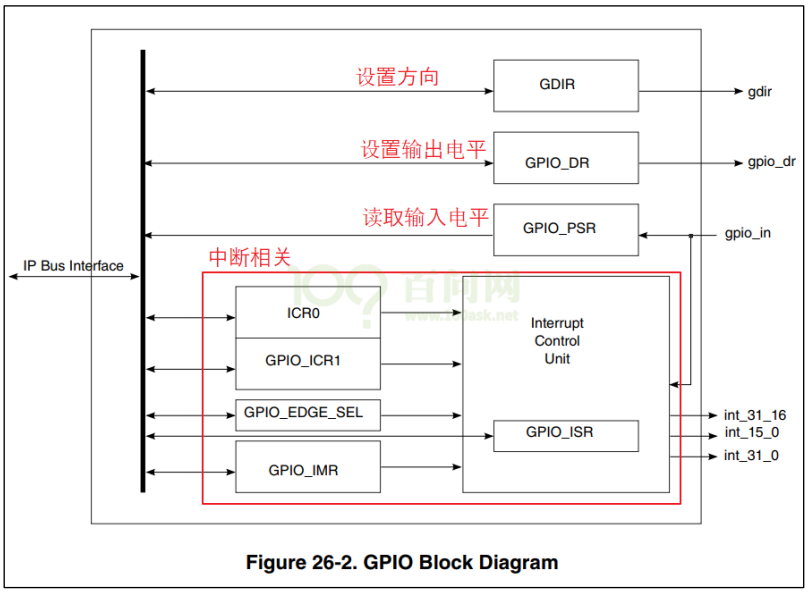
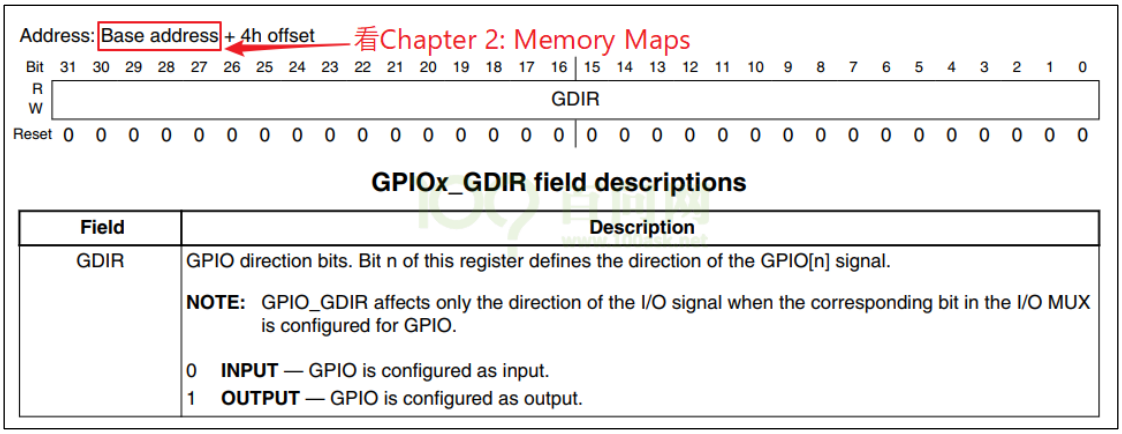


Figure 26-2. GPIO Block Diagram

图 4.13 GPIO 内部模块图

我们暂时只需要关心 3 个寄存器：

① GPIOx_GDIR：设置引脚方向，每位对应一个引脚，1-output，0-input

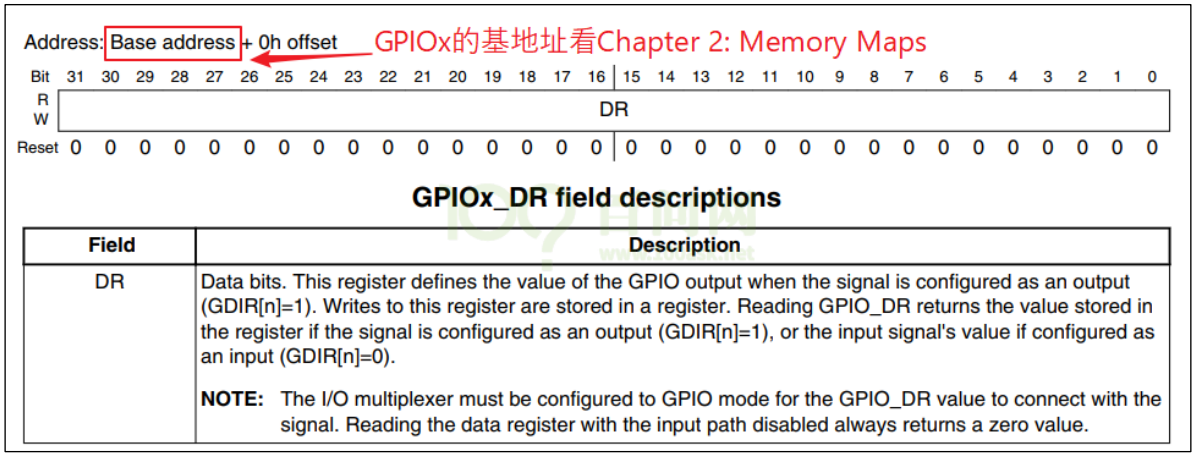


GPIOx_GDIR field descriptions

Field	Description
GDIR	GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal. **NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO. 0 **INPUT** — GPIO is configured as input. 1 **OUTPUT** — GPIO is configured as output.

图 4.14 GPIOx_GDIR

② GPIOx_DR：设置输出引脚的电平，每位对应一个引脚，1-高电平，0-低电平

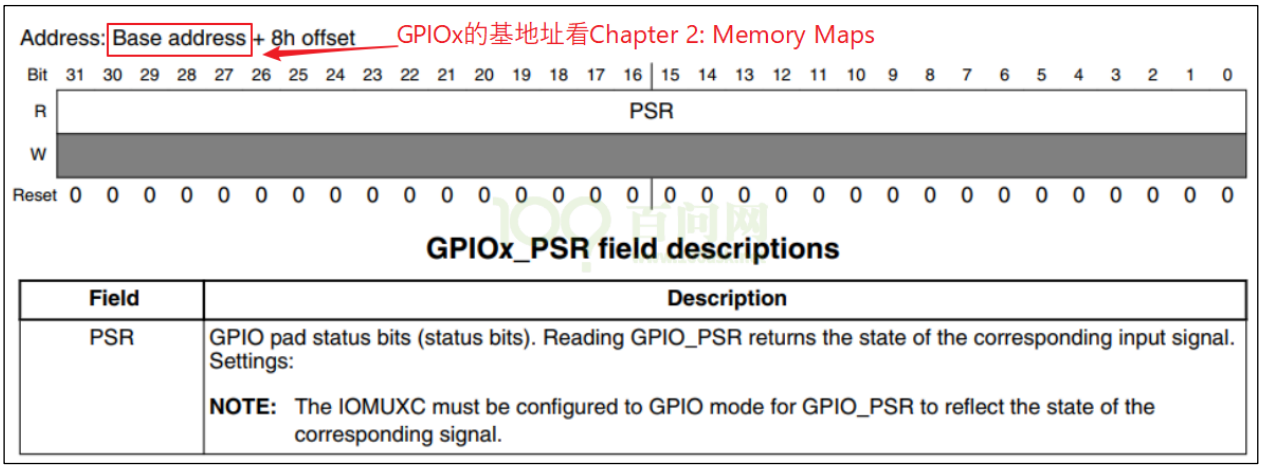


GPIOx_DR field descriptions

Field	Description
DR	Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0). **NOTE:** The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value.

图 4.15 GPIOx_DR

③ GPIOx_PSR：读取引脚的电平，每位对应一个引脚，1-高电平，0-低电平



GPIOx_PSR field descriptions

Field	Description
PSR	GPIO pad status bits (status bits). Reading GPIO_PSR returns the state of the corresponding input signal. Settings: **NOTE:** The IOMUXC must be configured to GPIO mode for GPIO_PSR to reflect the state of the corresponding signal.

图 4.16 GPIOx_PSR

4.1.5 读 GPIO

26.4.3.1 GPIO Read Mode

The programming sequence for reading input signals should be as follows:

1. Configure IOMUX to select GPIO mode (Via IOMUX Controller (IOMUXC)).
2. Configure GPIO direction register to input (GPIO_GDIR[GDIR] set to 0b).
3. Read value from data register/pad status register.

A pseudocode description to read [input3:input0] values is as follows:

```
// SET INPUTS TO GPIO MODE.
write sw_mux_ctl_<input0>_<input1>_<input2>_<input3>, 32'h00000000
// SET GDIR TO INPUT.
write GDIR[31:4,input3_bit, input2_bit, input1_bit, input0_bit,] 32'hxxxxxxx0
// READ INPUT VALUE FROM DR.
read DR
// READ INPUT VALUE FROM PSR.
read PSR
```

NOTE

While the GPIO direction is set to input (GPIO_GDIR = 0), a read access to GPIO_DR does not return GPIO_DR data. Instead, it returns the GPIO_PSR data, which is the corresponding input signal value.

图 4.17 如何读取 GPIO

翻译一下：

① 设置 CCM_CCGRx 寄存器中某位使能对应的 GPIO 模块 // 默认是使能的，上图省略了

② 设置 IOMUX 来选择引脚用于 GPIO

③ 设置 GPIOx_GDIR 中某位为 0，把该引脚设置为输入功能

④ 读 GPIOx_DR 或 GPIOx_PSR 得到某位的值（读 GPIOx_DR 返回的是 GPIOx_PSR 的值）

4.1.6 写 GPIO

26.4.3.2 GPIO Write Mode

The programming sequence for driving output signals should be as follows:

1. Configure IOMUX to select GPIO mode (Via IOMUXC), also enable SION if need to read loopback pad value through PSR
2. Configure GPIO direction register to output (GPIO_GDIR[GDIR] set to 1b).
3. Write value to data register (GPIO_DR).

A pseudocode description to drive 4'b0101 on [output3:output0] is as follows:

```
// SET PADS TO GPIO MODE VIA IOMUX.
write sw_mux_ctl_pad_<output[0-3]>.mux_mode, <GPIO_MUX_MODE>
// Enable loopback so we can capture pad value into PSR in output mode
write sw_mux_ctl_pad_<output[0-3]>.sion, 1
// SET GDIR=1 TO OUTPUT BITS.
write GDIR[31:4,output3_bit,output2_bit, output1_bit, output0_bit,] 32'hxxxxxxxF
// WRITE OUTPUT VALUE=4'b0101 TO DR.
write DR, 32'hxxxxxxx5
// READ OUTPUT VALUE FROM PSR ONLY.
read_cmp PSR, 32'hxxxxxxx5
```

图 4.18 如何写 GPIO

翻译一下：

① 设置 CCM_CCGRx 寄存器中某位使能对应的 GPIO 模块 // 默认是使能的，上图省略了

② 设置 IOMUX 来选择引脚用于 GPIO

③ 设置 GPIOx_GDIR 中某位为 1，把该引脚设置为输出功能

④ 写 GPIOx_DR 某位的值

需要注意的是，你可以设置该引脚的 loopback 功能，这样就可以从 GPIOx_PSR 中读到引脚的有实电平；你从 GPIOx_DR 中读回的只是上次设置的值，它并不能反应引脚的真实电平，比如可能因为硬件故障导致该引脚跟地短路了，你通过设置 GPIOx_DR 让它输出高电平并不会起效果。

第5章 最简单的 LED 驱动程序

这是补录的视频，它的章节序号是 6A，表示看下面的第六章之前先看第 6A 章。后面的 LED 驱动程序为了容易扩展，引入了很多数据结构。对 C 语言的要求有点高，所以我们基于 Hello 驱动程序先写出最简单的 LED 驱动程序。

5.1 LED 操作方法_基于 IMX6ULL

视频中的文档，放在 GIT 仓库中：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
doc_pic\pic\6A.最简单的 LED 驱动程序\03_IMX6ULL 的 LED 操作方法.pptx
```

5.2 最简单的 LED 驱动程序编程_基于 IMX6ULL

视频中的源码文档，放在 GIT 仓库中：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\02_led_drv\00_led_simple\imx6ull
```

5.2.1 字符设备驱动程序框架

字符设备驱动程序的框架：

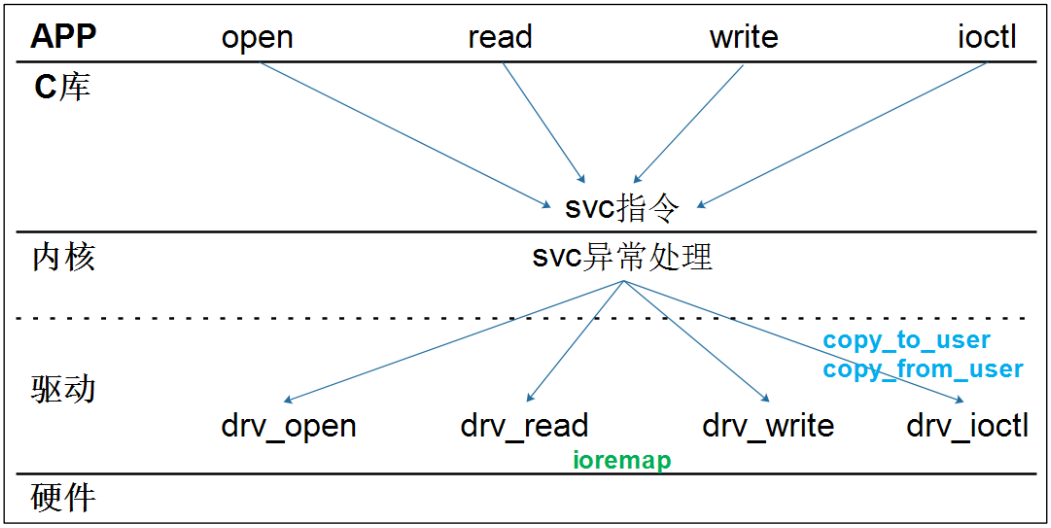


图 5.1 字符设备驱动框架

编写驱动程序的套路：

① 确定主设备号，也可以让内核分配

② 定义自己的 file_operations 结构体

③ 实现对应的 drv_open/drv_read/drv_write 等函数，填入 file_operations 结构体

④ 把 file_operations 结构体告诉内核：register_chrdev

⑤ 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数
⑥ 有入口函数就应该有出口函数：卸载驱动程序时，出口函数调用 unregister_chrdev
⑦ 其他完善：提供设备信息，自动创建设备节点：class_create, device_create

- 驱动怎么操作硬件？
  - 通过 ioremap 映射寄存器的物理地址得到虚拟地址，读写虚拟地址。
- 驱动怎么和 APP 传输数据？
  - 通过 copy_to_user、copy_from_user 这 2 个函数。

5.2.2 实现什么功能

先编写驱动程序：

- 实现 led_open 函数，在里面初始化 LED 引脚。
- 实现 led_write 函数，在里面根据 APP 传来的值控制 LED。
- 再编写测试程序。

5.2.3 上机实验

① 先设置工具链，参考《>>>>>>>>第三篇 2.6.3 配置交叉编译工具链》的内容。
② 再编译程序，把代码上传代服务器后执行 make 命令。
③ 接着在开发板上挂载 NFS，参考第 2 篇第六章的内容。
④ 最后在开发板上加载驱动程序，执行测试程序，如下：

```
[root@100ask:~]# echo "7 4 1 7" > /proc/sys/kernel/printk  // 打开内核的打印信息，有些板子默认打开了
[root@100ask:~]# insmod /mnt/led_drv.ko
/mnt/ledtest /dev/myled on                    // 点灯
/mnt/ledtest /dev/myled off                   // 关灯
```

第6章 LED 驱动程序框架

注意：如果做实验安装驱动时提示 version magic 不匹配，请看本文档最后的“常见问题”。

6.1 回顾字符设备驱动程序框架

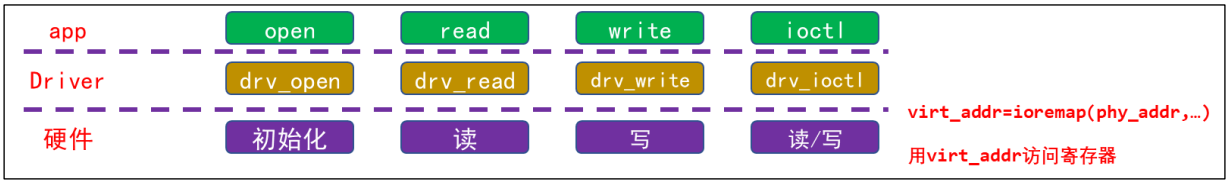


图 6.1 字符设备驱动程序框架

图 6.1 中驱动层访问硬件外设寄存器依靠的是 ioremap 函数去映射到寄存器地址，然后开始控制寄存器。

那么该如何编写驱动程序？

① 确定主设备号，也可以让内核分配；

② 定义自己的 file_operations 结构体，这是核心；

③ 实现对应的 drv_open/drv_read/drv_write 等函数，填入 file_operations 结构体；

④ 把 file_operations 结构体告诉内核：通过 register_chrdev 函数；

⑤ 谁来注册驱动程序？需要一个入口函数：安装驱动程序时，就会去调用这个入口函数；

⑥ 有入口函数就应该有出口函数：卸载驱动程序是，出口函数调用 unregister_chrdev;

⑦ 其它完善：提供设备信息，自动创建设备节点：class_create,device_create;

6.2 对于 LED 驱动，我们想要什么样的接口？

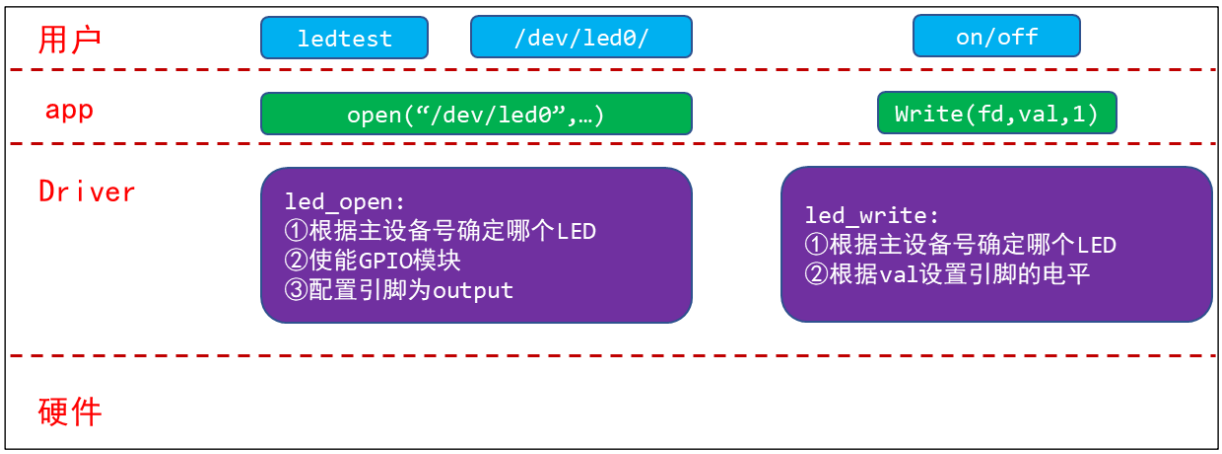


图 6.2 LED 驱动程序接口定义构思

6.3 LED 驱动能支持多个板子的基础：分层思想

① 把驱动拆分为通用的框架(leddrv.c)、具体的硬件操作(board_X.c)：

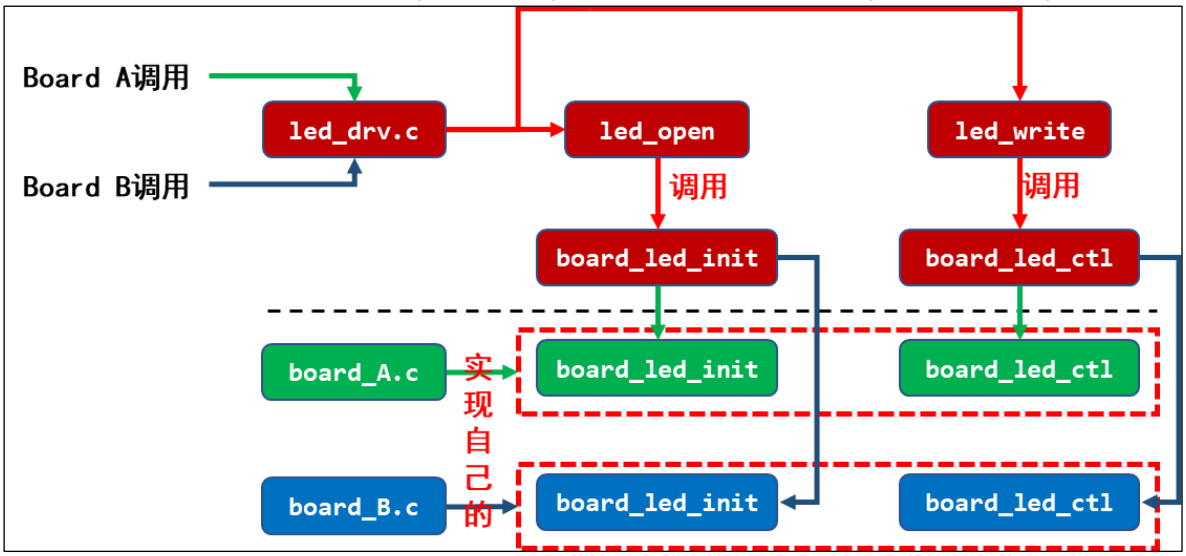


图 6.3 拆分驱动构成通用框架

② 以面向对象的思想，改进代码，抽象出一个结构体：

```
struct led_operations {
    int (*init) (int which); /* 初始化LED, which-哪个LED */
    int (*ctl) (int which, int status); /* 控制LED, which-哪个LED, status:1-亮,0-灭 */
};
```

图 6.4 面向对象的思想抽象出结构体

每个单板相关的 board_X.c 实现自己的 led_operations 结构体，供上层的 leddrv.c 调用：

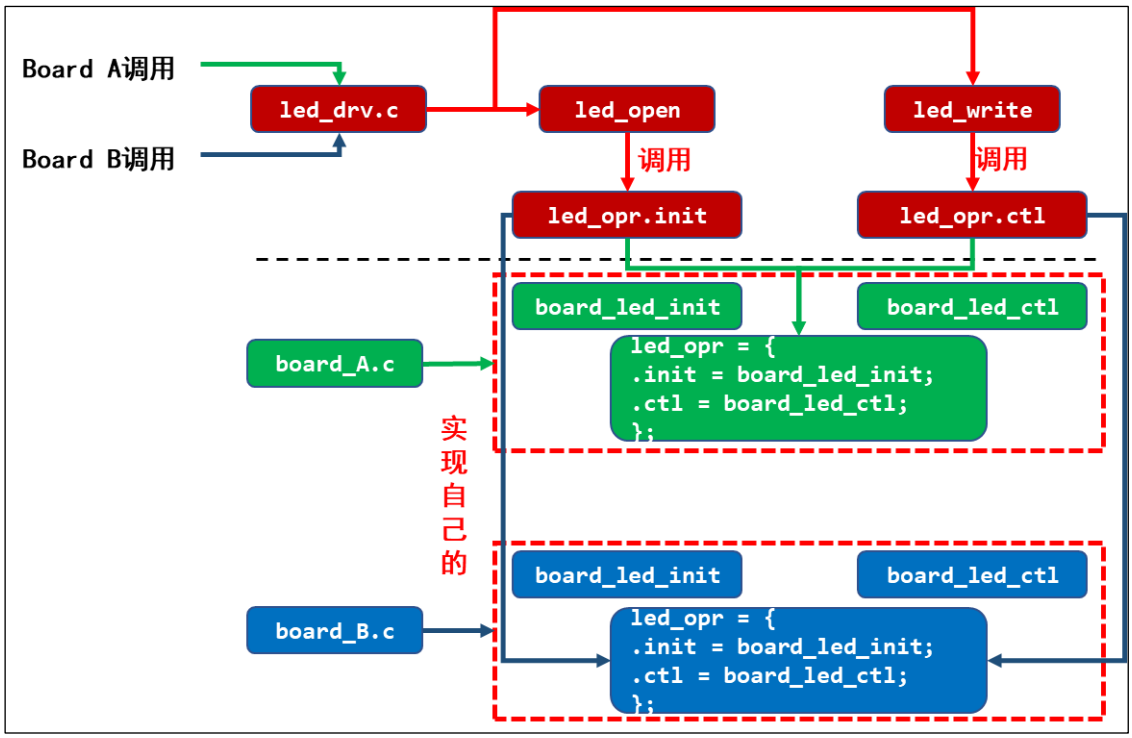


图 6.5 构建实现自己的结构体

6.4 写代码

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\02_led_drv\01_led_drv_template
```

6.4.1 驱动程序

驱动程序分为上下两层：leddrv.c、board_demo.c。leddrv.c 负责注册 file_operations 结构体，它的 open/write 成员会调用 board_demo.c 中提供的硬件 led_opr 中的对应函数。

1 把 LED 的操作抽象出一个 led_operations 结构体

首先看看 led_opr.h，它定义了一个 led_operations 结构体，把 LED 的操作抽象为这个结构体：

```
01 #ifndef _LED_OPR_H
02 #define _LED_OPR_H
03
04 struct led_operations {
05      int (*init) (int which); /* 初始化 LED，which-哪个 LED */
06      int (*ctl) (int which, char status); /* 控制 LED，which-哪个 LED，status:1-亮,0
-灭 */
07 };
08
09 struct led_operations *get_board_led_opr(void);
10
11
12 #endif
```

2 驱动程序的上层：file_operations 结构体

上层是 leddrv.c，它的核心是 file_operations 结构体，首先看看入口函数：

```
80 /* 4. 把 file_operations 结构体告诉内核：注册驱动程序 */
81 /* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数 */
82 static int __init led_init(void)
83 {
84      int err;
85      int i;
86
87      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
88      major = register_chrdev(0, "100ask_led", &led_drv);  /* /dev/led */
89
90
91      led_class = class_create(THIS_MODULE, "100ask_led_class");
92      err = PTR_ERR(led_class);
93      if (IS_ERR(led_class)) {
94              printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
95              unregister_chrdev(major, "100ask_led");
96              return -1;
97      }
98
```

```
99      for (i = 0; i < LED_NUM; i++)
100             device_create(led_class, NULL, MKDEV(major, i), NULL, “100ask_led%d”,
i); /* /dev/100ask_led0,1,… */
101
102     p_led_opr = get_board_led_opr();
103
104     return 0;
105 }
```

第 88 行向内核注册一个 file_operations 结构体。

第 102 行从底层硬件相关的代码 board_demo.c 中获得 led_operaions 结构体。

再来看看 leddrv.c 中 file_operations 结构体的成员函数：

```
37 /* write(fd, &val, 1); */
38 static ssize_t led_drv_write (struct file *file, const char __user *buf, size_t si
ze, loff_t *offset)
39 {
40      int err;
41      char status;
42      struct inode *inode = file_inode(file);
43      int minor = iminor(inode);
44
45      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
46      err = copy_from_user(&status, buf, 1);
47
48      /* 根据次设备号和 status 控制 LED */
49      p_led_opr->ctl(minor, status);
50
51      return 1;
52 }
53
54 static int led_drv_open (struct inode *node, struct file *file)
55 {
56      int minor = iminor(node);
57
58      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
59      /* 根据次设备号初始化 LED */
60      p_led_opr->init(minor);
61
62      return 0;
63 }
64
65 static int led_drv_close (struct inode *node, struct file *file)
66 {
67      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
68      return 0;
69 }
70
71 /* 2. 定义自己的 file_operations 结构体 */
72 static struct file_operations led_drv = {
73      .owner   = THIS_MODULE,
74      .open    = led_drv_open,
75      .read    = led_drv_read,
76      .write   = led_drv_write,
77      .release = led_drv_close,
```

```
78 };
```

第 49 行、第 60 行，会调用 led_operations 结构体中对应的函数。

6.4.2 测试程序

测试程序为 ledtest.c：

```
01
02 #include <sys/types.h>
03 #include <sys/stat.h>
04 #include <fcntl.h>
05 #include <unistd.h>
06 #include <stdio.h>
07 #include <string.h>
08
09 /*
10  * ./ledtest /dev/100ask_led0 on
11  * ./ledtest /dev/100ask_led0 off
12  */
13 int main(int argc, char **argv)
14 {
15     int fd;
16     char status;
17
18     /* 1. 判断参数 */
19     if (argc != 3)
20     {
21             printf("Usage: %s <dev> <on | off>\n", argv[0]);
22             return -1;
23     }
24
25     /* 2. 打开文件 */
26     fd = open(argv[1], O_RDWR);
27     if (fd == -1)
28     {
29             printf("can not open file %s\n", argv[1]);
30             return -1;
31     }
32
33     /* 3. 写文件 */
34     if (0 == strcmp(argv[2], "on"))
35     {
36             status = 1;
37             write(fd, &status, 1);
38     }
39     else
40     {
41             status = 0;
42             write(fd, &status, 1);
43     }
44
45     close(fd);
46
47     return 0;
48 }
49
```

第 26 行打开设备节点。

如果用户想点亮 LED，第 37 行会把值“1”通过 write 函数写入驱动程序。

如果用户想熄灭 LED，第 42 行会把值“0”通过 write 函数写入驱动程序。

6.4.3 上机测试

这只是一个示例程序，还没有真正操作硬件。测试程序操作驱动程序时，只会导致驱动程序中打印信息。

首先设置交叉工具链，修改驱动 Makefile 中内核的源码路径，编译驱动和测试程序。

启动开发板后，通过 NFS 访问编译好驱动程序、测试程序，就可以在开发板上如下操作了：

```
[root@100ask:~]# insmod 100ask_led.ko   // 装载驱动程序
[13449.134044] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_init li
ne 87
[root@100ask:~]# ls /dev/100ask_led* -l   // 可以得到 2 个设备节点
crw-------    1 root     root      235,   0 Jan 18 12:34 /dev/100ask_led0
crw-------    1 root     root      235,   1 Jan 18 12:34 /dev/100ask_led1
[root@100ask:~]#./ledtest /dev/100ask_led0 on    // 点亮 LED
[13463.176987] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_ope
n line 58
[13463.197877] /home/book/source/02_led_drv/01_led_drv_template/board_demo.c board_d
emo_led_init line 22, led 0
[13463.216232] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_wri
te line 45
[13463.232889] /home/book/source/02_led_drv/01_led_drv_template/board_demo.c board_d
emo_led_ctl line 28, led 0, on   // 可以看到这句“on”打印
[13463.247977] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_clo
se line 67
[root@100ask:~]#./ledtest /dev/100ask_led0 off     // 熄灭 LED
[13464.540637] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_ope
n line 58
[13464.554380] /home/book/source/02_led_drv/01_led_drv_template/board_demo.c board_d
emo_led_init line 22, led 0
[13464.569671] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_wri
te line 45
[13464.580615] /home/book/source/02_led_drv/01_led_drv_template/board_demo.c board_d
emo_led_ctl line 28, led 0, off    // 可以看到这句“off”打印
[13464.593397] /home/book/source/02_led_drv/01_led_drv_template/leddrv.c led_drv_clo
se line 67
```

6.5 课后作业

实现读 LED 状态的功能：涉及 APP 和驱动。

第7章 具体单板的 LED 驱动程序

我们选用的内核都是 4.x 版本，操作都是类似的：

- rk3399　linux 4.4.154
- rk3288　linux 4.4.154
- imx6ul　linux 4.9.88
- am3358　linux 4.9.168

录制视频时，我的 source insight 里总是使用某个版本的内核。这没有关系，驱动程序中调用的内核函数，在这些 4.x 版本的内核里都是一样的。

7.1 怎么写 LED 驱动程序？

详细步骤如下：

① 看原理图确定引脚，确定引脚输出什么电平才能点亮/熄灭 LED

② 看主芯片手册，确定寄存器操作方法：哪些寄存器？哪些位？地址是？

③ 编写驱动：先写框架，再写硬件操作的代码

注意：在芯片手册中确定的寄存器地址被称为物理地址，在 Linux 内核中无法直接使用。

需要使用内核提供的 ioremap 把物理地址映射为虚拟地址，使用虚拟地址。

7.1.1 ioremap 函数的使用：

① 函数原型：

```
void __iomem *ioremap(resource_size_t res_cookie, size_t size)
```

使用时，要包含头文件：

```
#include <asm/io.h>
```

② 作用：

把物理地址 phys_addr 开始的一段空间(大小为 size)，映射为虚拟地址；返回值是该段虚拟地址的首地址。

```
virt_addr  = ioremap(phys_addr, size);
```

实际上，它是按页(4096 字节)进行映射的，是整页整页地映射的。

假设 phys_addr = 0x10002，size=4，ioremap 的内部实现是：

a) phys_addr 按页取整，得到地址 0x10000

b) size 按页取整，得到 4096

c) 把起始地址 0x10000，大小为 4096 的这一块物理地址空间，映射到虚拟地址空间，

假设得到的虚拟空间起始地址为 0xf0010000

d) 那么 phys_addr = 0x10002 对应的 virt_addr = 0xf0010002

③ 不再使用该段虚拟地址时，要 iounmap(virt_addr)：

```
void iounmap(volatile void __iomem *cookie)
```

7.1.2 volatile 的使用：

① 编译器很聪明，会帮我们做些优化，比如：

```
int   a;
a = 0;   // 这句话可以优化掉，不影响 a 的结果
a = 1;
```

② 有时候编译器会自作聪明，比如：

```
int *p = ioremap(xxxx, 4);  // GPIO 寄存器的地址
*p = 0;   // 点灯，但是这句话被优化掉了
*p = 1;   // 灭灯
```

③ 对于上面的情况，为了避免编译器自动优化，需要加上 volatile，告诉它“这是容易出错的，别乱优化”：

```
volatile  int *p = ioremap(xxxx, 4);  // GPIO 寄存器的地址
*p = 0;   // 点灯，这句话不会被优化掉
*p = 1;   // 灭灯
```

7.2 百问网 IMX6ULL 的 LED 驱动程序

7.2.1 led 原理图

LED 原理图如图 7.1，它使用 GPIO5_IO03，引脚输出低电平时 LED 被点亮，输出高电平时 LED 被熄灭：

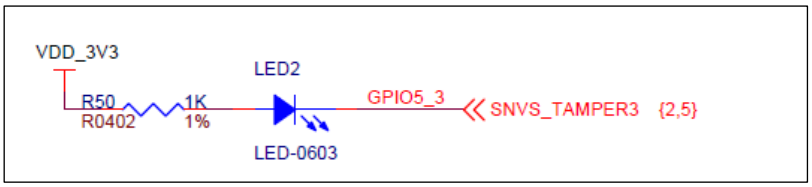


图 7.1 百问网 IMX6ULL 开发板 LED 原理图

7.2.2 所涉及的寄存器操作

GPIO 模块如图 7.2：

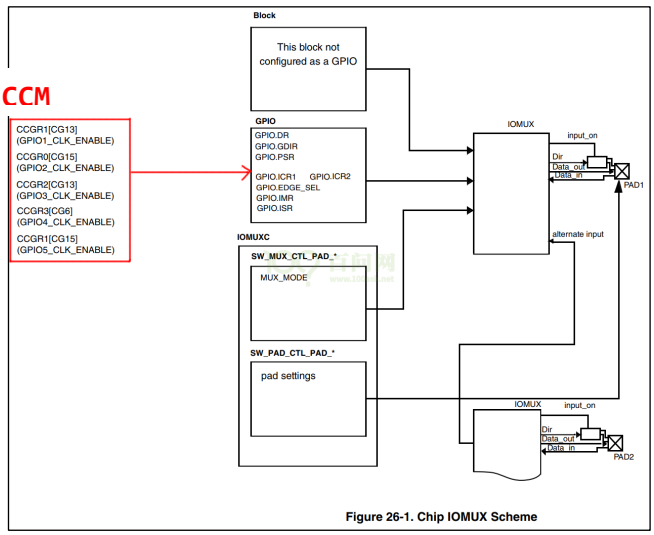


图 7.2 GPIO 模块内部示意图

代码中对硬件的操作截图如图 7.3，截图便于对比，后面有文字便于复制：

```
/* GPIO5_IO03 */
/* a. 使能GPIO5
 * set CCM to enable GPIO5
 * CCM_CCGR1[CG15] 0x20C406C
 * bit[31:30] = 0b11
 */

/* b. 设置GPIO5_IO03用于GPIO
 * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
 *      to configure GPIO5_IO03 as GPIO
 * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3  0x2290014
 * bit[3:0] = 0b0101 alt5
 */

/* b. 设置GPIO5_IO03作为output引脚
 * set GPIO5_GDIR to configure GPIO5_IO03 as output
 * GPIO5_GDIR  0x020AC000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置GPIO5_DR输出低电平
 * set GPIO5_DR to configure GPIO5_IO03 output 0
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b0
 */

/* e. 设置GPIO5_IO3输出高电平
 * set GPIO5_DR to configure GPIO5_IO03 output 1
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b1
 */
```

只有 3 个寄存器地址不同

这个寄存器涉及的位不同

① ② ③

图 7.3 代码操作硬件截图

第1步 使能 GPIO5

Address: 20C_4000h base + 6Ch offset = 20C_406Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR1 field descriptions

Field	Description
31–30 CG15	gpio5 clock (gpio5_clk_enable)
29–28 CG14	csu clock (csu_clk_enable)

图 7.4 寄存器中 GPIO5 的使能位

设置 b[31:30]就可以使能 GPIO5，设置为什么值呢？

注意：在 imx6ullrm.pdf 中，CCM_CCGR1 的 b[31:30]是保留位；我以前写程序时错用了 imx6ul(不是 imx6ull)的手册，导致程序中额外操作了这些保留位。不去设置 b[31:30]，GPIO5 也是默认使能的。

看下图，设置为 0b11：

CGR value	Clock Activity Description
00	Clock is off during all modes. Stop enter hardware handshake is disabled.
01	Clock is on in run mode, but off in WAIT and STOP modes
10	Not applicable (Reserved).
11	Clock is on during all modes, except STOP mode.

图 7.5 CCM_CCGR1 的值及其描述

① 00：该 GPIO 模块全程被关闭

② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭

③ 10：保留

④ 11：该 GPIO 模块全程使能

```
/* GPIO5_IO03 */
/* a. 使能 GPIO5
 * set CCM to enable GPIO5
 * CCM_CCGR1[CG15] 0x20C406C
 * bit[31:30] = 0b11
 */
```

第2步 设置 GPIO5_IO03 为 GPIO 模式

设置如下寄存器：

Address: 229_0000h base + 14h offset = 229_0014h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad SNVS_TAMPER3 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	Mux Mode Select Field **NOTE:** ALT5 mode is only valid when TAMPER PIN is used as GPIO. This depends on FUSE setting "TAMPER_PIN_DISABLE[1:0]". Following is the mux information when TAMPER PIN is used as GPIO: SNVS_TAMPER3 ==> GPIO5_03 101 **ALT5** — Select mux mode: ALT5 mux port, GPIO5_IO03 of instance - gpio Other Reserved

图 7.6 GPIO5_IO03 的 GPIO 模式

```
/* b. 设置 GPIO5_IO03 用于 GPIO
 * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
 *     to configure GPIO5_IO03 as GPIO
 * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3  0x2290014
 * bit[3:0] = 0b0101 alt5
 */
```

第3步 设置 GPIO5_IO03 为输出引脚，设置其输出电平

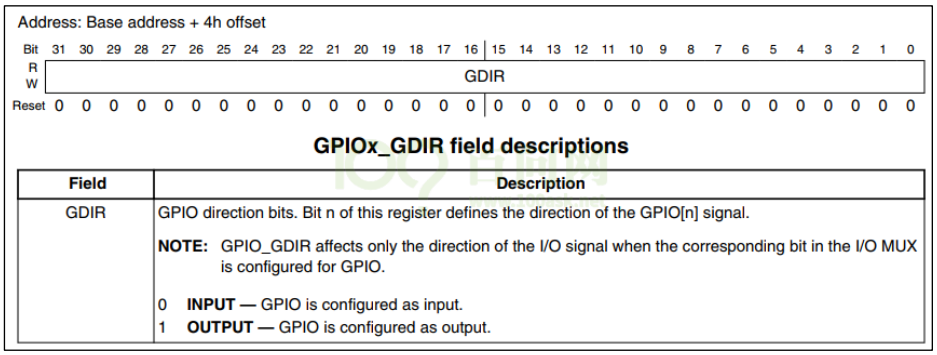
寄存器地址为：

GPIO memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
209_C000	GPIO data register (GPIO1_DR)	32	R/W	0000_0000h	28.5.1/1358
209_C004	GPIO direction register (GPIO1_GDIR)	32	R/W	0000_0000h	28.5.2/1359
20A_C000	GPIO data register (GPIO5_DR)	32	R/W	0000_0000h	28.5.1/1358
20A_C004	GPIO direction register (GPIO5_GDIR)	32	R/W	0000_0000h	28.5.2/1359

图 7.7 设置 GPIO5_IO03 为输出引脚

设置方向寄存器，把引脚设置为输出引脚：



Address: Base address + 4h offset

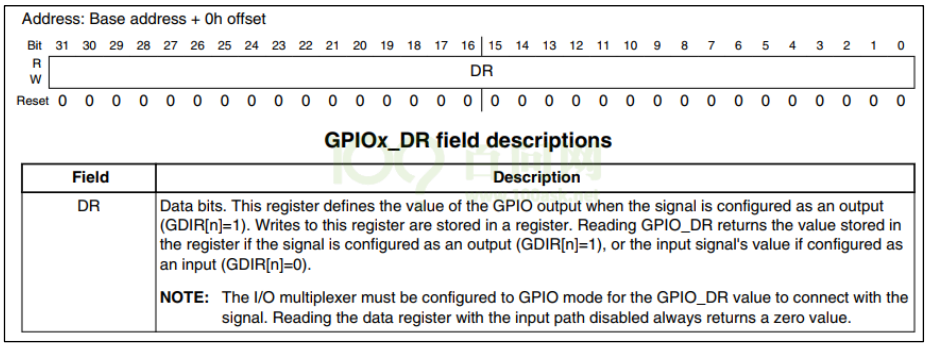
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	GDIR																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_GDIR field descriptions

Field	Description
GDIR	GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal. **NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO. 0 **INPUT** — GPIO is configured as input. 1 **OUTPUT** — GPIO is configured as output.

图 7.8 方向寄存器

设置数据寄存器，设置引脚的输出电平：



Address: Base address + 0h offset

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	DR																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_DR field descriptions

Field	Description
DR	Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0). **NOTE:** The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value.

图 7.9 数据寄存器

```
/* c. 设置 GPIO5_IO03 作为 output 引脚
 * set GPIO5_GDIR to configure GPIO5_IO03 as output
 * GPIO5_GDIR  0x020AC000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置 GPIO5_DR 输出低电平
 * set GPIO5_DR to configure GPIO5_IO03 output 0
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b0
 */

/* e. 设置 GPIO5_IO3 输出高电平
 * set GPIO5_DR to configure GPIO5_IO03 output 1
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b1
 */
```

7.2.3 写程序

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\02_led_drv\02_led_drv_for_boards\100ask_imx6ull_src_bin
```

硬件相关的文件是 board_fire_imx6ull-pro.c，其他文件跟 LED 框架驱动程序完全一样。

它首先构造了一个 led_operations 结构体，用来表示 LED 的硬件操作：

```
100 static struct led_operations board_demo_led_opr = {
101     .num  = 1,
102     .init = board_demo_led_init,
103     .ctl  = board_demo_led_ctl,
104 };
```

led_operations 结构体中有 init 函数指针，它指向 board_demo_led_init 函数，在里面将会初始化 LED 引脚：使能、设置为 GPIO 模式、设置为输出引脚。

值得关注的是第 35～38 行，对于寄存器要先使用 ioremap 得到它的虚拟地址，以后使用虚拟地址访问寄存器：

```
21 static volatile unsigned int *CCM_CCGR1                           ;
22 static volatile unsigned int *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3;
23 static volatile unsigned int *GPIO5_GDIR                          ;
24 static volatile unsigned int *GPIO5_DR                           ;
25
26 static int board_demo_led_init (int which) /* 初始化 LED, which-哪个 LED */
27 {
28     unsigned int val;
29
30     //printk("%s %s line %d, led %d\n", __FILE__, __FUNCTION__, __LINE__, which);
31     if (which == 0)
32     {
33         if (!CCM_CCGR1)
34         {
35             CCM_CCGR1  = ioremap(0x20C406C, 4);
36         IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 = ioremap(0x2290014, 4);
37             GPIO5_GDIR  = ioremap(0x020AC000 + 0x4, 4);
38             GPIO5_DR    = ioremap(0x020AC000 + 0, 4);
39         }
40
41         /* GPIO5_IO03 */
42         /* a. 使能 GPIO5
43          * set CCM to enable GPIO5
44          * CCM_CCGR1[CG15] 0x20C406C
45          * bit[31:30] = 0b11
46          */
47         *CCM_CCGR1 |= (3<<30);
48
49         /* b. 设置 GPIO5_IO03 用于 GPIO
50          * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
51          *      to configure GPIO5_IO03 as GPIO
52          * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3  0x2290014
53          * bit[3:0] = 0b0101 alt5
54          */
55         val = *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3;
56         val &= ~(0xf);
57         val |= (5);
58         *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 = val;
59
60
61         /* b. 设置 GPIO5_IO03 作为 output 引脚
62          * set GPIO5_GDIR to configure GPIO5_IO03 as output
```

```
63          * GPIO5_GDIR  0x020AC000 + 0x4
64          * bit[3] = 0b1
65          */
66         *GPIO5_GDIR |= (1<<3);
67     }
68
69     return 0;
70 }
```

led_operations 结构体中有 ctl 函数指针，它指向 board_demo_led_ctl 函数，在里面将会根据参数设置 LED 引脚的输出电平：

```
72 static int board_demo_led_ctl (int which, char status) /* 控制 LED, which-哪个 LED,
status:1-亮,0-灭 */
73 {
74     //printk("%s %s line %d, led %d, %s\n", __FILE__, __FUNCTION__, __LINE__, whic
h, status ? "on" : "off");
75     if (which == 0)
76     {
77         if (status) /* on: output 0*/
78         {
79             /* d. 设置 GPIO5_DR 输出低电平
80              * set GPIO5_DR to configure GPIO5_IO03 output 0
81              * GPIO5_DR 0x020AC000 + 0
82              * bit[3] = 0b0
83              */
84             *GPIO5_DR &= ~(1<<3);
85         }
86         else  /* off: output 1*/
87         {
88             /* e. 设置 GPIO5_IO3 输出高电平
89              * set GPIO5_DR to configure GPIO5_IO03 output 1
90              * GPIO5_DR 0x020AC000 + 0
91              * bit[3] = 0b1
92              */
93             *GPIO5_DR |= (1<<3);
94         }
95
96     }
97     return 0;
98 }
```

下面的 get_board_led_opr 函数供上层调用，给上层提供 led_operations 结构体：

```
106 struct led_operations *get_board_led_opr(void)
107 {
108     return &board_demo_led_opr;
109 }
```

7.2.4 上机实验

① 首先设置工具链，然后修改驱动程序 Makefile 指定内核源码路径，就可以编译驱动程序和测试程序了。

② 启动开发板，挂载 NFS 文件系统，这样就可以访问到 Ubuntu 中的文件。

③ 最后，就可以在开发板上进行下列测试。

注意：如果要使用板子自带的系统，关闭原有 LED 驱动的方法是类似的，也是进入开发板/sys/class/leds/目录，对于每一个 LED 在该目录下都有一个子目录，假设某个子目录名为 XXX，则执行如下命令：

```
[root@100ask:~]# echo none  >  /sys/class/leds/XXX/trigger
```

使用我们的系统时，要先禁止内核中原来的 LED 驱动，把“heatbeat”功能关闭，执行以下命令即可：

```
[root@100ask:~]#echo none > /sys/class/leds/cpu/trigger
```

这样就可以使用我们的驱动程序做实验了：

```
[root@100ask:~]#insmod  100ask_led.ko
[root@100ask:~]#/ledtest  /dev/100ask_led0  on
[root@100ask:~]#/ledtest  /dev/100ask_led0  off
```

如果想恢复原来的心跳功能，可以执行：

```
[root@100ask:~]#echo heartbeat > /sys/class/leds/cpu/trigger
```

7.2.5 课后作业

- 在驱动里有 ioremap，什么时候执行 iounmap？请完善程序
- 视频里我们只实现了点一个 LED，修改代码支持两个 LED。

第8章 驱动设计的思想

8.1 面向对象

- 字符设备驱动程序抽象出一个 file_operations 结构体；
- 我们写的程序针对硬件部分抽象出 led_operations 结构体。

8.2 分层

上下分层，比如我们前面写的 LED 驱动程序就分为 2 层：

① 上层实现硬件无关的操作，比如注册字符设备驱动：leddrv.c

② 下层实现硬件相关的操作，比如 board_A.c 实现单板 A 的 LED 操作

图 8.1 led 的分层

8.3 分离

还能不能改进？分离。

在 board_A.c 中，实现了一个 led_operations，为 LED 引脚实现了初始化函数、控制函数：

```
static struct led_operations board_demo_led_opr = {
    .num  = 1,
    .init = board_demo_led_init,
    .ctl  = board_demo_led_ctl,
};
```

如果硬件上更换一个引脚来控制 LED 怎么办？你要去修改上面结构体中的 init、ctl 函数。

实际情况是，每一款芯片它的 GPIO 操作都是类似的。比如：GPIO1_3、GPIO5_4 这 2 个引脚接到 LED：

① GPIO1_3 属于第 1 组，即 GPIO1。

a) 有方向寄存器 DIR、数据寄存器 DR 等，基础地址是 addr_base_addr_gpio1。

b) 设置为 output 引脚：修改 GPIO1 的 DIR 寄存器的 bit3。

c) 设置输出电平：修改 GPIO1 的 DR 寄存器的 bit3。

② GPIO5_4 属于第 5 组，即 GPIO5。

a) 有方向寄存器 DIR、数据寄存器 DR 等，基础地址是 addr_base_addr_gpio5。

b) 设置为 output 引脚：修改 GPIO5 的 DIR 寄存器的 bit4。

c) 设置输出电平：修改 GPIO5 的 DR 寄存器的 bit4。

既然引脚操作那么有规律，并且这是跟主芯片相关的，那可以针对该芯片写出比较通用的硬件操作代码。

比如 board_A.c 使用芯片 chipY，那就可以写出：chipY_gpio.c，它实现芯片 Y 的 GPIO 操作，适用于芯片 Y 的所有 GPIO 引脚。

使用时，我们只需要在 board_A_led.c 中指定使用哪一个引脚即可。程序结构如下：

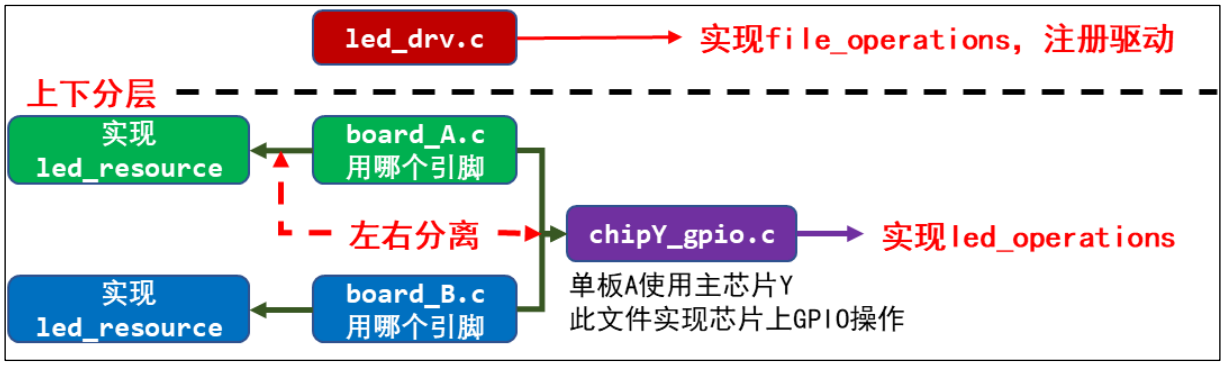


图 8.2 程序结构

以面向对象的思想，在 board_A_led.c 中实现 led_resouce 结构体，它定义“资源”——要用哪一个引脚。

在 chipY_gpio.c 中仍是实现 led_operations 结构体，它要写得更完善，支持所有 GPIO。

8.4 写示例代码

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\02_led_drv\03_led_drv_template_seperate
```

程序仍分为上下结构：上层 leddrv.c 向内核注册 file_operations 结构体；下层 chip_demo_gpio.c 提供 led_operations 结构体来操作硬件。

下层的代码分为 2 个：chip_demo_gpio.c 实现通用的 GPIO 操作，board_A_led.c 指定使用哪个 GPIO，即“资源”。

led_resource.h 中定义了 led_resource 结构体，用来描述 GPIO：

```
04 /* GPIO3_0 */
05 /* bit[31:16] = group */
06 /* bit[15:0]  = which pin */
07 #define GROUP(x) (x>>16)
08 #define PIN(x)   (x&0xFFFF)
09 #define GROUP_PIN(g,p) ((g<<16) | (p))
10
11 struct led_resource {
12     int pin;
13 };
15 struct led_resource *get_led_resouce(void);
```

board_A_led.c 指定使用哪个 GPIO，它实现一个 led_resource 结构体，并提供访问函数：

```
02 #include "led_resource.h"
03
04 static struct led_resource board_A_led = {
05      .pin = GROUP_PIN(3,1),
06 };
07
08 struct led_resource *get_led_resouce(void)
09 {
10      return &board_A_led;
11 }
```

chip_demo_gpio.c 中，首先获得 board_A_led.c 实现的 led_resource 结构体，然后再进行其他操作，请看下面第 26 行：

```
20 static struct led_resource *led_rsc;
21 static int board_demo_led_init (int which) /* 初始化 LED, which-哪个 LED */
22 {
23      //printk("%s %s line %d, led %d\n", __FILE__, __FUNCTION__, __LINE__, which);
24      if (!led_rsc)
25      {
26              led_rsc = get_led_resouce();
27      }
```

8.5 课后作业

使用“分离”的思想，去改造前面写的 LED 驱动程序：实现 led_resouce，在里面可以指定要使用哪一个 LED；改造 led_operations，让它能支持更多 GPIO。

注意：作为练习，led_operations 结构体不需要写得很完善，不需要支持所有 GPIO，你可以只支持若干个 GPIO 即可。

示例：

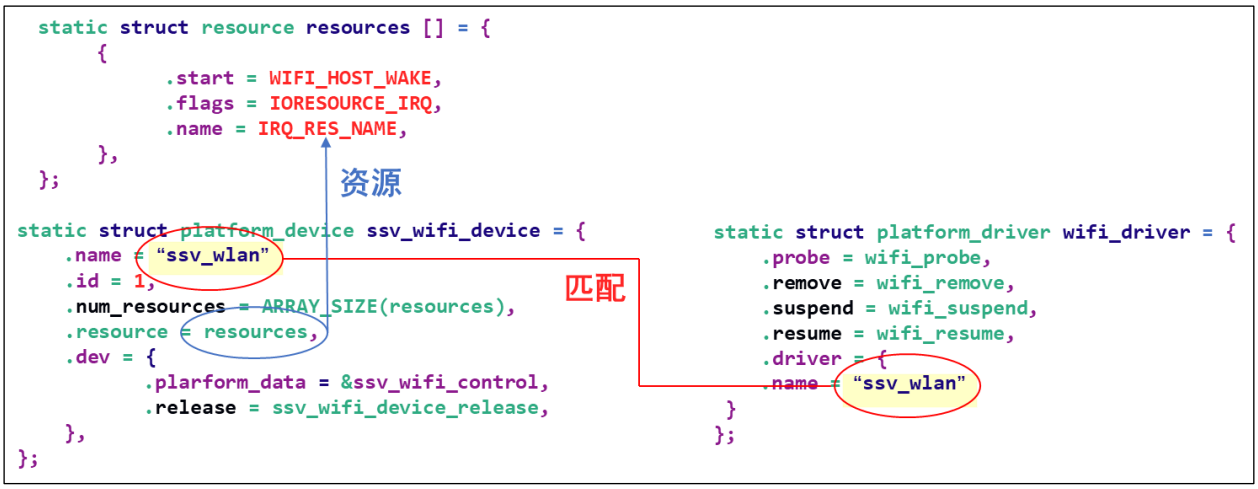


图 9.1 驱动模型示例

9.1 驱动编写的 3 种方法

以 LED 驱动为例。

9.1.1 传统写法

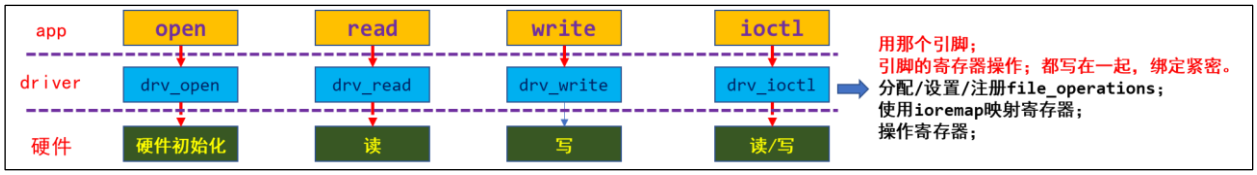


图 9.2 传统写法架构

使用哪个引脚，怎么操作引脚，都写死在代码中。
最简单，不考虑扩展性，可以快速实现功能。
修改引脚时，需要重新编译。

9.1.2 总线设备驱动模型

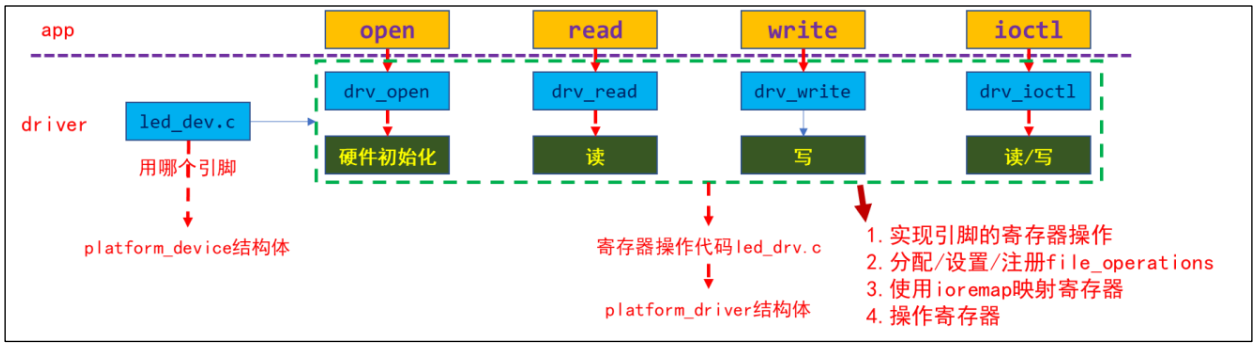


图 9.3 总线设备驱动模型

引入 platform_device/platform_driver，将“资源”与“驱动”分离开

来。

代码稍微复杂，但是易于扩展。

冗余代码太多，修改引脚时设备端的代码需要重新编译。

更换引脚时，图 9.3 中的 led_drv.c 基本不用改，但是需要修改 led_dev.c。

9.1.3 设备树

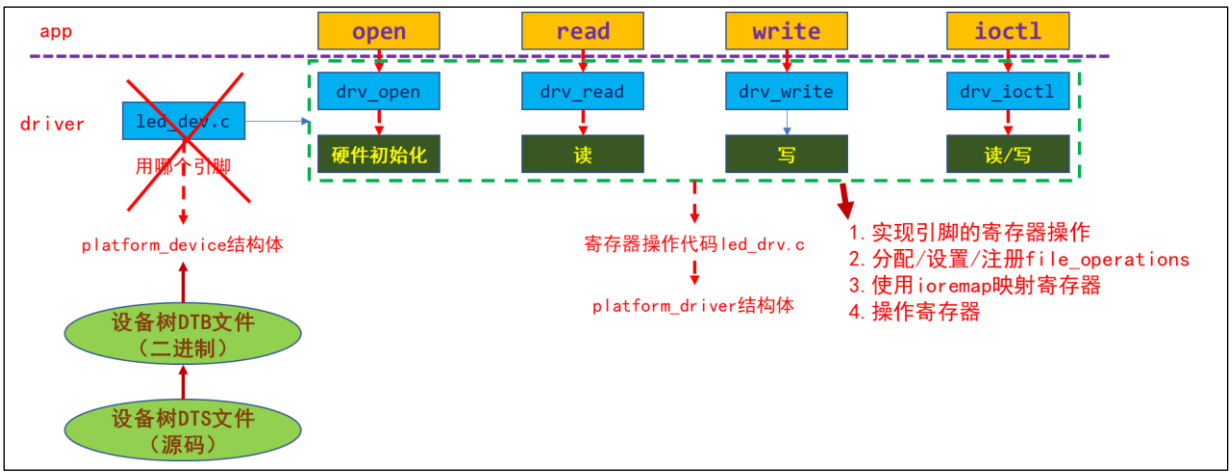


图 9.4 设备树驱动模型

通过配置文件——设备树来定义“资源”。

代码稍微复杂，但是易于扩展。

无冗余代码，修改引脚时只需要修改 dts 文件并编译得到 dtb 文件，把它传给内核。

无需重新编译内核/驱动。

9.2 在 Linux 中实现“分离”：Bus/Dev/Drv 模型

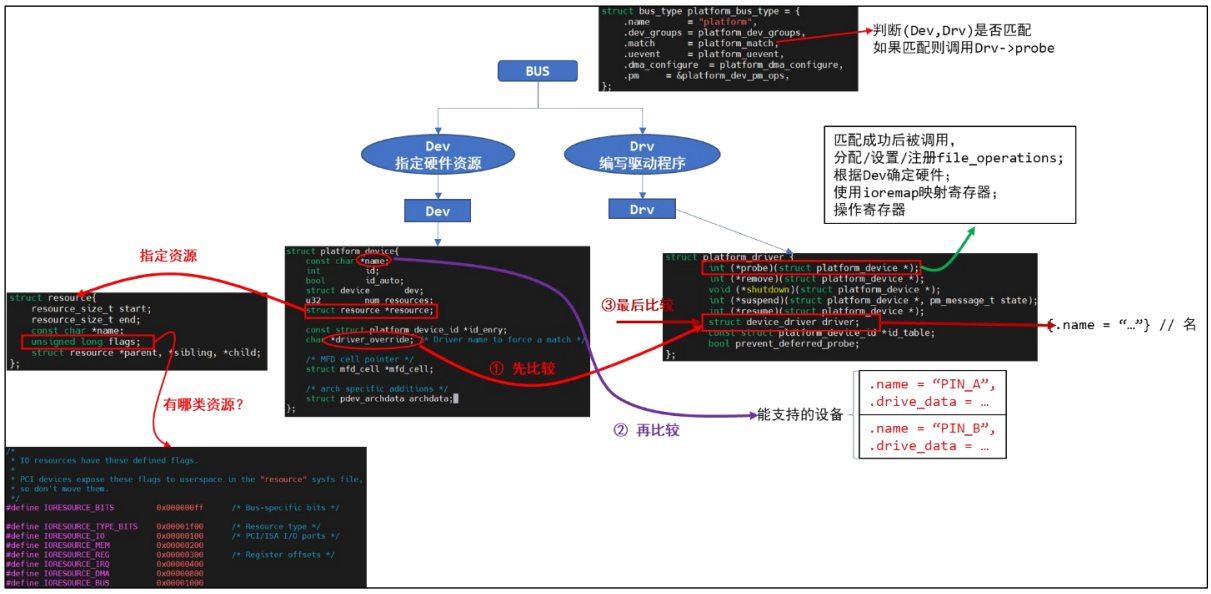


图 9.5 BUS/Dev/Drv 模型

9.3 匹配规则

9.3.1 最先比较

- platform_device.driver_override 和 platform_driver.driver.name
  可以设置 platform_device 的 driver_override，强制选择某个 platform_driver。

9.3.2 然后比较

- platform_device. name 和 platform_driver.id_table[i].name
  Platform_driver.id_table 是“platform_device_id”指针，表示该 drv 支持若干个 device，它里面列出了各个 device 的{.name, .driver_data}，其中的“name”表示该 drv 支持的设备的名字，driver_data 是些提供给该 device 的私有数据。

9.3.3 最后比较

- platform_device.name 和 platform_driver.driver.name
  platform_driver.id_table 可能为空，
  这时可以根据 platform_driver.driver.name 来寻找同名的 platform_device。

9.3.4 函数调用关系

```
platform_device_register
platform_device_add
    device_add
        bus_add_device // 放入链表
        bus_probe_device  // probe 枚举设备，即找到匹配的(dev, drv)
            device_initial_probe
                __device_attach
                    bus_for_each_drv(...,__device_attach_driver,...)
                        __device_attach_driver
                            driver_match_device(drv, dev) // 是否匹配
                            driver_probe_device         // 调用 drv 的 probe

platform_driver_register
__platform_driver_register
    driver_register
        bus_add_driver // 放入链表
            driver_attach(drv)
                    bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
                        __driver_attach
                            driver_match_device(drv, dev) // 是否匹配
                            driver_probe_device         // 调用 drv 的 probe
```

9.4 常用函数

这些函数可查看内核源码：drivers/base/platform.c，根据函数名即可知道其含义。

下面摘取常用的几个函数。

9.4.1 注册/反注册

```
platform_device_register/ platform_device_unregister
platform_driver_register/ platform_driver_unregister
platform_add_devices // 注册多个 device
```

9.4.2 获得资源

- 返回该 dev 中某类型(type)资源中的第几个(num)：

```
struct resource *platform_get_resource(struct platform_device *dev, unsigned int
                                       type,
                                       unsigned int num)
```

- 返回该 dev 所用的第几个(num)中断：

```
int platform_get_irq(struct platform_device *dev, unsigned int num)
```

- 通过名字(name)返回该 dev 的某类型(type)资源：

```
struct resource *platform_get_resource_byname(struct platform_device *dev,
                                              unsigned int type,
                                              const char *name)
```

- 通过名字(name)返回该 dev 的中断号：

```
int platform_get_irq_byname(struct platform_device *dev,  const char *name)
```

9.5 怎么写程序

9.5.1 分配/设置/注册 platform_device 结构体

在里面定义所用资源，指定设备名字。

9.5.2 分配/设置/注册 platform_driver 结构体

在其中的 probe 函数里，分配/设置/注册 file_operations 结构体，
并从 platform_device 中确实所用硬件资源。
指定 platform_driver 的名字。

9.6 课后作业

在内核源码中搜索 platform_device_register 可以得到很多驱动，选择一个作为例子：

① 确定它的名字
② 根据它的名字找到对应的 platform_driver
③ 进入 platform_device_register/platform_driver_register 内部，分析 dev 和 drv 的匹配过程

第10章 LED 模板驱动程序的改造：总线设备驱动模型

10.1 原来的框架

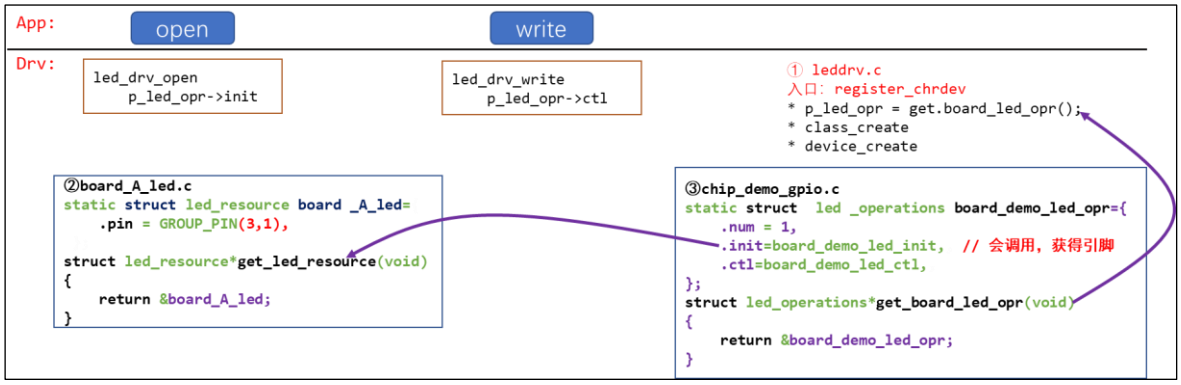


图 10.1 原本的驱动框架

10.2 要实现的框架

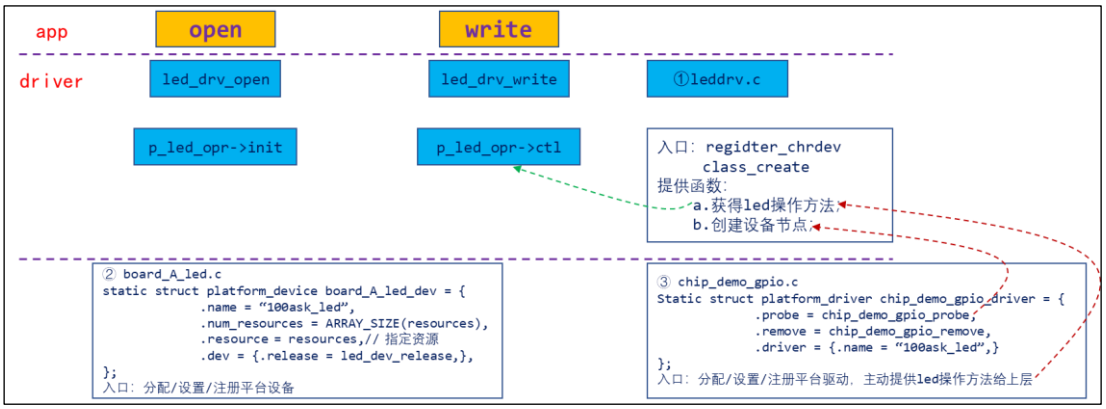


图 10.2 目标框架

10.3 写代码

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\02_led_drv\04_led_drv_template_bus_dev_drv
```

10.3.1 注意事项

① 如果 platform_device 中不提供 release 函数，如下图所示不提供红框部分的函数：

```
static struct platform_device board_A_led_dev = {
        .name = "100ask_led",
        .num_resources = ARRAY_SIZE(resources),
        .resource = resources,
        .dev = {
                .release = led_dev_release,
         },
};
```

图 10.3 release device

则在调用 platform_device_unregister 时会出现警告，如下图所示：

图 10.4 不释放的警告

你可以提供一个 release 函数，如果实在无事可做，把这函数写为空。

② EXPORT_SYMBOL

a.c 编译为 a.ko，里面定义了 func_a；如果它想让 b.ko 使用该函数，那么 a.c 里需要导出此函数(如果 a.c，b.c 都编进内核，则无需导出)：

```
EXPORT_SYMBOL(led_device_create);
```

并且，使用时要先加载 a.ko。如果先加载 b.ko，会有类似如下“Unknown symbol”的提示：

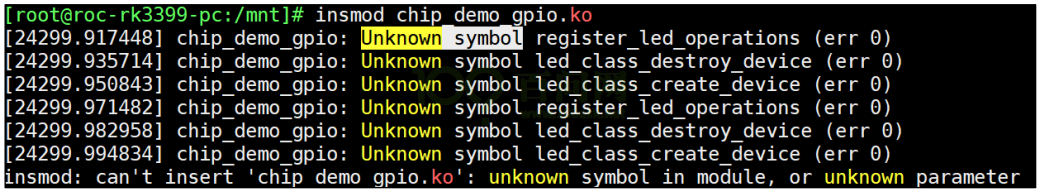


图 10.5 先加载 b.ko

10.3.2 实现 platform_device 结构体

board_A.c 作为一个可加载模块，里面也有入口函数、出口函数。在入口函数中注册 platform_device 结构体，在 platform_device 结构体中指定使用哪个 GPIO 引脚。

首先看入口函数，它调用 platform_device_register 函数，向内核注册 board_A_led_dev 结构体：

```
50 static int __init led_dev_init(void)
51 {
52     int err;
53
54     err = platform_device_register(&board_A_led_dev);
55
56     return 0;
57 }
```

board_A_led_dev 结构体定义如下。

```
23 static void led_dev_release(struct device *dev)
24 {
25 }
26
27 static struct resource resources[] = {
28     {
29         .start = GROUP_PIN(3,1),
30         .flags = IORESOURCE_IRQ,
31         .name = "100ask_led_pin",
32     },
33     {
34         .start = GROUP_PIN(5,8),
35         .flags = IORESOURCE_IRQ,
```

```
36        .name = "100ask_led_pin",
37    },
38 };
39
40
41 static struct platform_device board_A_led_dev = {
42         .name = "100ask_led",
43         .num_resources = ARRAY_SIZE(resources),
44         .resource = resources,
45         .dev = {
46                 .release = led_dev_release,
47          },
48 };
```

在 resouces 数组中指定了 2 个引脚(第 27～38 行)；

我们还提供了一个空函数 led_dev_release(第 23～25 行)，它被赋给 board_A_led_dev 结构体(第 46 行)，这个函数在卸载 platform_device 时会被调用，如果不提供的话内核会打印警告信息。

10.3.3 实现 platform_driver 结构体

chip_demo_gpio.c 中注册 platform_driver 结构体，它使用 Bus/Dev/Drv 模型，当有匹配的 platform_device 时，它的 probe 函数就会被调用。

在 probe 函数中所做的事情跟之前的代码没有差别。

先看入口函数：

```
138 static struct platform_driver chip_demo_gpio_driver = {
139     .probe      = chip_demo_gpio_probe,
140     .remove     = chip_demo_gpio_remove,
141     .driver     = {
142         .name   = "100ask_led",
143     },
144 };
145
146 static int __init chip_demo_gpio_drv_init(void)
147 {
148     int err;
149
150     err = platform_driver_register(&chip_demo_gpio_driver);
151     register_led_operations(&board_demo_led_opr);
152
153     return 0;
154 }
```

第 150 行向内核注册一个 platform_driver 结构体，这个结构体的核心在于第 140 行的 chip_demo_gpio_probe 函数。

chip_demo_gpio_probe 函数代码如下：。

```
100 static int chip_demo_gpio_probe(struct platform_device *pdev)
101 {
102     struct resource *res;
103     int i = 0;
104
```

```
105     while (1)
106     {
107         res = platform_get_resource(pdev, IORESOURCE_IRQ, i++);
108         if (!res)
109             break;
110
111         g_ledpins[g_ledcnt] = res->start;
112         led_class_create_device(g_ledcnt);
113         g_ledcnt++;
114     }
115     return 0;
116
117 }
```

- 第 107 行：从匹配的 platform_device 中获取资源，确定 GPIO 引脚。
- 第 111 行：把引脚记录下来，在操作硬件时要用。
- 第 112 行：新发现了一个 GPIO 引脚，就调用上层驱动的代码创建设备节点

操作硬件的代码如下，第 31、63 行的代码里用到了数组 g_ledpins，里面的值来自 platform_device，在 probe 函数中根据 platform_device 的资源确定了引脚：

```
23 static int g_ledpins[100];
24 static int g_ledcnt = 0;
25
26 static int board_demo_led_init (int which) /* 初始化 LED, which-哪个 LED */
27 {
28     //printk("%s %s line %d, led %d\n", __FILE__, __FUNCTION__, __LINE__, which);
29
30   printk("init gpio: group %d, pin %d\n", GROUP(g_ledpins[which]), PIN(g_ledpins[w
hich]));
31     switch(GROUP(g_ledpins[which]))
32     {
33         case 0:
34         {
35             printk("init pin of group 0 ...\n");
36             break;
37         }
38         case 1:
39         {
40             printk("init pin of group 1 ...\n");
41             break;
42         }
43         case 2:
44         {
45             printk("init pin of group 2 ...\n");
46             break;
47         }
48         case 3:
49         {
50             printk("init pin of group 3 ...\n");
51             break;
52         }
53     }
54
55     return 0;
```

```
56 }
57
58 static int board_demo_led_ctl (int which, char status) /* 控制 LED, which-哪个 LED,
status:1-亮,0-灭 */
59 {
60     //printk("%s %s line %d, led %d, %s\n", __FILE__, __FUNCTION__, __LINE__, whic
h, status ? "on" : "off");
61    printk("set led %s: group %d, pin %d\n", status ? "on" : "off", GROUP(g_ledpins
[which]), PIN(g_ledpins[which]));
62
63     switch(GROUP(g_ledpins[which]))
64     {
65         case 0:
66         {
67             printk("set pin of group 0 ...\n");
68             break;
69         }
70         case 1:
71         {
72             printk("set pin of group 1 ...\n");
73             break;
74         }
75         case 2:
76         {
77             printk("set pin of group 2 ...\n");
78             break;
79         }
80         case 3:
81         {
82             printk("set pin of group 3 ...\n");
83             break;
84         }
85     }
86
87     return 0;
88 }
89
90 static struct led_operations board_demo_led_opr = {
91     .init = board_demo_led_init,
92     .ctl  = board_demo_led_ctl,
93 };
94
95 struct led_operations *get_board_led_opr(void)
96 {
97     return &board_demo_led_opr;
98 }
```

10.4 课后作业

完善半成品程序：04_led_drv_template_bus_dev_drv_unfinished。

请仿照本节提供的程序(位于 04_led_drv_template_bus_dev_drv 目录)，改造你所用的单板的 LED 驱动程序。

第11章 驱动进化之路：设备树的引入及简明教程

- 官方文档(可以下载到 devicetree-specification-v0.2.pdf):
  https://www.devicetree.org/specifications/
- 内核文档:
  Documentation/devicetree/booting-without-of.txt

我录制的设备树视频，它是基于 s3c2440 的，用的是 linux 4.19；需要深入研究的可以看该视频(收费)。

注意，如果只是想入门，看本文档及视频即可。

11.1 设备树的引入与作用

以 LED 驱动为例，如果你要更换 LED 所用的 GPIO 引脚，需要修改驱动程序源码、重新编译驱动、重新加载驱动。

在内核中，使用同一个芯片的板子，它们所用的外设资源不一样，比如 A 板用 GPIO A，B 板用 GPIO B。而 GPIO 的驱动程序既支持 GPIO A 也支持 GPIO B，你需要指定使用哪一个引脚，怎么指定？在 c 代码中指定。

随着 ARM 芯片的流行，内核中针对这些 ARM 板保存有大量的、没有技术含量的文件。

Linus 大发雷霆："this whole ARM thing is a f*cking pain in the ass"。

于是，Linux 内核开始引入设备树。

设备树并不是重新发明出来的，在 Linux 内核中其他平台如 PowerPC，早就使用设备树来描述硬件了。

Linus 发火之后，内核开始全面使用设备树来改造，神人就神人。

有一种错误的观点，说“新驱动都是用设备树来写了”。设备树不可能用来写驱动。

请想想，要操作硬件就需要去操作复杂的寄存器，如果设备树可以操作寄存器，那么它就是“驱动”，它就一样很复杂。

设备树只是用来给内核里的驱动程序，指定硬件的信息。比如 LED 驱动，在内核的驱动程序里去操作寄存器，但是操作哪一个引脚？这由设备树指定。

你可以事先体验一下设备树，板子启动后执行下面的命令：

```
# ls /sys/firmware/
devicetree  fdt
```

/sys/firmware/devicetree 目录下是以目录结构程现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用 cat 命令把它打印出来；对于数值，可以用 hexdump 把它打印出来。

一个单板启动时，u-boot 先运行，它的作用是启动内核。U-boot 会把内核和设备树文件都读入内存，然后启动内核。在启动内核时会把设备树在内存中的

地址告诉内核。

11.2 设备树的语法

为什么叫“树”？

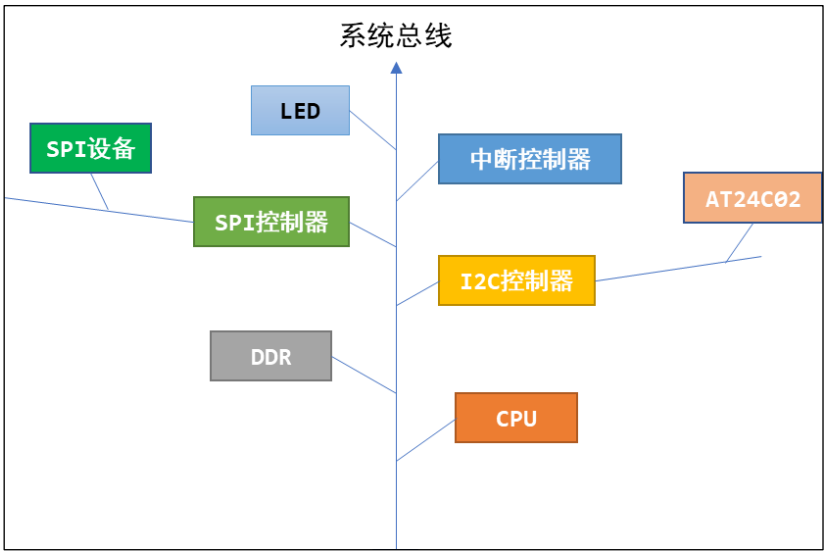


图 11.1 总线“树”

怎么描述这棵树？

我们需要编写设备树文件(dts: device tree source)，它需要编译为dtb(device tree blob)文件，内核使用的是 dtb 文件。

dts 文件是根本，它的语法很简单。

下面是一个设备树示例：

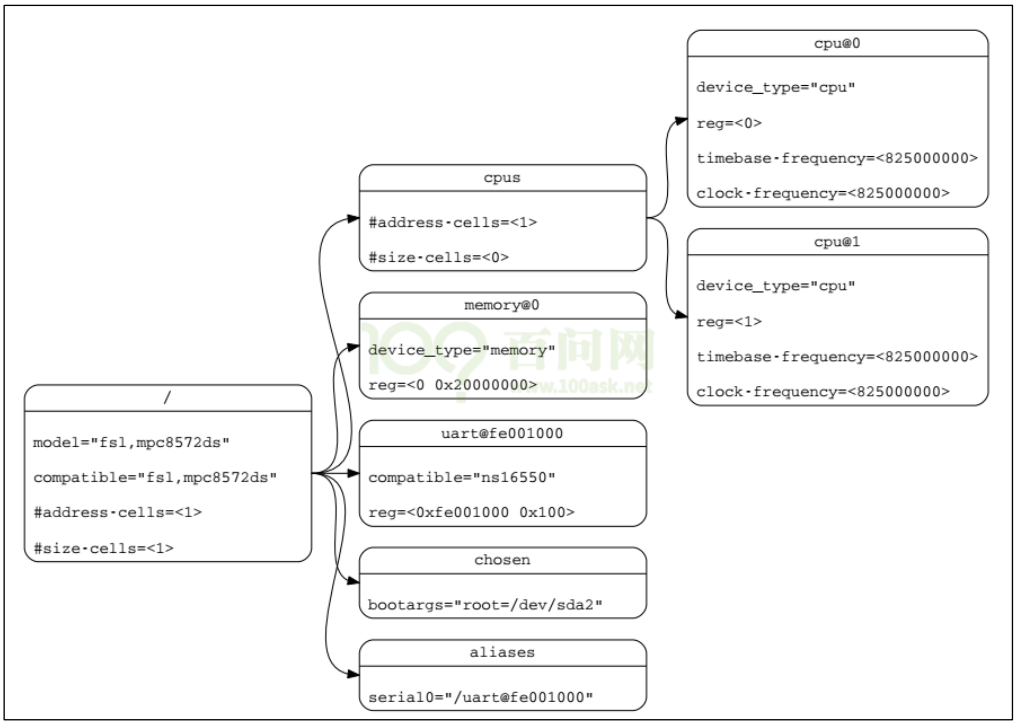


图 11.2 设备树示例

它对应的 dts 文件如下：

```
/dts-v1/;

/ {
    model="fsl,mpc8572ds"
    compatible="fsl,mpc8572ds"
    #address-cells=<1>
    #size-cells=<1>

    cpus {
        #address-cells=<1>
        #size-cells=<0>
        cpu@0 {
            device_type="cpu"
            reg=<0>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };

        cpu@1 {
            device_type="cpu"
            reg=<1>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };

    };

    memory@0 {
        device_type="memory"
        reg=<0 0x20000000>
    };

    uart@fe001000 {
        compatible="ns16550"
        reg=<0xfe001000 0x100>
    };

    chosen {
        bootargs="root=/dev/sda2";
    };

    aliases {
        serial0="/uart@fe001000"
    };
};
```

图 11.3 设备树 dts 文件

11.2.1 Devicetree 格式

1 DTS 文件的格式

DTS 文件布局(layout):

```
/dts-v1/;                // 表示版本
[memory reservations]    // 格式为: /memreserve/ <address> <length>;
/ {
    [property definitions]
    [child nodes]
};
```

2 node 的格式

设备树中的基本单元，被称为“node”，其格式为:

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
};
```

label 是标号，可以省略。label 的作用是为了方便地引用 node，比如:

```
/dts-v1/;
/ {
    uart0: uart@fe001000 {
        compatible="ns16550";
        reg=<0xfe001000 0x100>;
    };
};
```

可以使用下面 2 种方法来修改 uart@fe001000 这个 node:

```
// 在根节点之外使用 label 引用 node:
&uart0 {
    status = “disabled”;
};
或在根节点之外使用全路径:
&{/uart@fe001000}  {
    status = “disabled”;
};
```

3 properties 的格式

简单地说，properties 就是“name=value”，value 有多种取值方式。

- Property 格式 1:

```
[label:] property-name = value;
```

- Property 格式 2(没有值):

```
[label:] property-name;
```

- Property 取值只有 3 种:

```
arrays of cells(1 个或多个 32 位数据，64 位数据使用 2 个 32 位数据表示),
string(字符串),
bytestring(1 个或多个字节)
```

示例:

a) Arrays of cells : cell 就是一个 32 位的数据，用尖括号包围起来

```
interrupts = <17 0xc>;
```

- 64bit 数据使用 2 个 cell 来表示，用尖括号包围起来:

```
clock-frequency = <0x00000001 0x00000000>;
```

b) A null-terminated string (有结束符的字符串)，用双引号包围起来:

```
compatible = "simple-bus";
```

c) A bytestring(字节序列) ，用中括号包围起来:

```
local-mac-address = [00 00 12 34 56 78];  // 每个 byte 使用 2 个 16 进制数来表示
local-mac-address = [000012345678];       // 每个 byte 使用 2 个 16 进制数来表示
```

- 可以是各种值的组合，用逗号隔开:

```
compatible = "ns16550", "ns8250";
example = <0xf00f0000 19>, "a strange property format";
```

11.2.2 dts 文件包含 dtsi 文件

设备树文件不需要我们从零写出来，内核支持了某款芯片比如 imx6ull，在内核的 arch/arm/boot/dts 目录下就有了能用的设备树模板，一般命名为 xxxx.dtsi。“i”表示“include”，被别的文件引用的。

我们使用某款芯片制作出了自己的单板，所用资源跟 xxxx.dtsi 是大部分相同，小部分不同，所以需要引脚 xxxx.dtsi 并修改。

dtsi 文件跟 dts 文件的语法是完全一样的。

dts 中可以包含.h 头文件，也可以包含 dtsi 文件，在.h 头文件中可以定义一些宏。

示例：

```
/dts-v1/;
#include <dt-bindings/input/input.h>
#include "imx6ull.dtsi"

/ {
……
};
```

11.2.3 常用的属性

1 #address-cells、#size-cells

- cell 指一个 32 位的数值，
- address-cells：address 要用多少个 32 位数来表示；
- size-cells：size 要用多少个 32 位数来表示。

比如一段内存，怎么描述它的起始地址和大小？

下例中，address-cells 为 1，所以 reg 中用 1 个数来表示地址，即用 0x80000000 来表示地址；size-cells 为 1，所以 reg 中用 1 个数来表示大小，即用 0x20000000 表示大小：

```
/ {
#address-cells = <1>;
#size-cells = <1>;
memory {
reg = <0x80000000 0x20000000>;
    };
};
```

2 compatible

“compatible”表示“兼容”，对于某个 LED，内核中可能有 A、B、C 三个驱动都支持它，那可以这样写：

```
led {
compatible = “A”, “B”, “C”;
};
```

内核启动时，就会为这个 LED 按这样的优先顺序为它找到驱动程序：A、B、C。

根节点下也有 compatible 属性，用来选择哪一个“machine desc”：一个内核可以支持 machine A，也支持 machine B，内核启动后会根据根节点的 compatible 属性找到对应的 machine desc 结构体，执行其中的初始化函数。

compatible 的值，建议取这样的形式："manufacturer,model"，即“厂家名，模块名”。

注意：machine desc 的意思就是“机器描述”，学到内核启动流程时才涉及。

3 model

model 属性与 compatible 属性有些类似，但是有差别。

compatible 属性是一个字符串列表，表示可以你的硬件兼容 A、B、C 等驱动；

model 用来准确地定义这个硬件是什么。

比如根节点中可以这样写：

```
{
    compatible = "samsung,smdk2440", "samsung,mini2440";
    model = "jz2440_v3";
};
```

它表示这个单板，可以兼容内核中的“smdk2440”，也兼容“mini2440”。

从 compatible 属性中可以知道它兼容哪些板，但是它到底是什么板？用 model 属性来明确。

4 status

dtsi 文件中定义了很多设备，但是在你的板子上某些设备是没有的。这时你可以给这个设备节点添加一个 status 属性，设置为“disabled”：

```
&uart1 {
     status = "disabled";
};
```

Table 2.4: Values for status property

Value	Description
"okay"	Indicates the device is operational. 设备正常运行
"disabled"	Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). 设备不可操作，但是后面可以恢复工作 Refer to the device binding for details on what disabled means for a given device.
"fail"	Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair. 发生了严重错误，需修复
"fail-sss"	Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The *sss* portion of the value is specific to the device and indicates the error condition detected. 发生了严重错误，需修复；sss表示错误信息

图 11.4 设备属性

5 reg

reg 的本意是 register，用来表示寄存器地址。

但是在设备树里，它可以用来描述一段空间。反正对于 ARM 系统，寄存器和内存是统一编址的，即访问寄存器时用某块地址，访问内存时用某块地址，在访问方法上没有区别。

reg 属性的值，是一系列的“address size”，用多少个 32 位的数来表示 address 和 size，由其父节点的#address-cells、#size-cells 决定。

示例：

```
/dts-v1/;
/ {
#address-cells = <1>;
#size-cells = <1>;
memory {
reg = <0x80000000 0x20000000>;
};
};
```

6 name(过时了，建议不用)

它的值是字符串，用来表示节点的名字。在跟 platform_driver 匹配时，优先级最低。

compatible 属性在匹配过程中，优先级最高。

11.2.3.7 1device_type(过时了，建议不用)

它的值是字符串，用来表示节点的类型。在跟 platform_driver 匹配时，优先级为中。

compatible 属性在匹配过程中，优先级最高。

11.2.4 常用的节点(node)

1 根节点

dts 文件中必须有一个根节点：

```
/dts-v1/;
/ {
model = "SMDK24440";
compatible = "samsung,smdk2440";

#address-cells = <1>;
#size-cells = <1>;
};
```

根节点中必须有这些属性：

```
#address-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述地址(address)
#size-cells   // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述大小(size)
compatible   // 定义一系列的字符串，用来指定内核中哪个 machine_desc 可以支持本设备
             // 即这个板子兼容哪些平台
             // uImage : smdk2410 smdk2440 mini2440     ==> machine_desc

model       // 咱这个板子是什么
```

```
            // 比如有 2 款板子配置基本一致，它们的 compatible 是一样的
            // 那么就通过 model 来分辨这 2 款板子
```

2 CPU 节点

一般不需要我们设置，在 dtsi 文件中都定义好了：

```
cpus {
       #address-cells = <1>;
       #size-cells = <0>;

       cpu0: cpu@0 {
           .......
       }
};
```

3 memory 节点

芯片厂家不可能事先确定你的板子使用多大的内存，所以 memory 节点需要板厂设置，比如：

```
memory {
reg = <0x80000000 0x20000000>;
};
```

4 chosen 节点

我们可以通过设备树文件给内核传入一些参数，这要在 chosen 节点中设置 bootargs 属性：

```
chosen {
bootargs = "noinitrd root=/dev/mtdblock4 rw init=/linuxrc console=ttySAC0,115200";
};
```

11.3 编译、更换设备树

我们一般不会从零写 dts 文件，而是修改。程序员水平有高有低，改得对不对？需要编译一下。并且内核直接使用 dts 文件的话，就太低效了，它也需要使用二进制格式的 dtb 文件。

11.3.1 在内核中直接 make

设置 ARCH、CROSS_COMPILE、PATH 这三个环境变量后，进入 ubuntu 上板子内核源码的目录，执行如下命令即可编译 dtb 文件：

```
make  dtbs  V=1
```

11.3.2 手工编译

除非你对设备树比较了解，否则不建议手工使用 dtc 工具直接编译。

内核目录下 scripts/dtc/dtc 是设备树的编译工具，直接使用它的话，包含其他文件时不能使用“#include”，而必须使用“/incldue”。

编译、反编译的示例命令如下，“-I”指定输入格式，“-O”指定输出格式，“-o”指定输出文件：

```
./scripts/dtc/dtc -I dts -O dtb -o tmp.dtb arch/arm/boot/dts/xxx.dts  // 编译 dts 为 dtb
./scripts/dtc/dtc -I dtb -O dts -o tmp.dts arch/arm/boot/dts/xxx.dtb  // 反编译 dtb 为 dts
```

11.3.3 给开发板更换设备树文件

怎么给各个单板编译出设备树文件，它们的设备树文件是哪一个？

基本方法都是：设置 ARCH、CROSS_COMPILE、PATH 这三个环境变量后，在内核源码目录中执行：

```
make  dtbs
```

11.3.4 板子启动后查看设备树

板子启动后执行下面的命令：

```
# ls /sys/firmware/
devicetree  fdt
```

/sys/firmware/devicetree 目录下是以目录结构程现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用 cat 命令把它打印出来；对于数值，可以用 hexdump 把它打印出来。

还可以看到/sys/firmware/fdt 文件，它就是 dtb 格式的设备树文件，可以把它复制出来放到 ubuntu 上，执行下面的命令反编译出来(-I dtb：输入格式是 dtb，-O dts：输出格式是 dts)：

```
cd  板子所用的内核源码目录
./scripts/dtc/dtc  -I  dtb  -O  dts   /从板子上/复制出来的/fdt  -o   tmp.dts
```

11.4 内核对设备树的处理

从源代码文件 dts 文件开始，设备树的处理过程为：

图 11.5 设备树的处理过程

① dts 在 PC 机上被编译为 dtb 文件；
② u-boot 把 dtb 文件传给内核；
③ 内核解析 dtb 文件，把每一个节点都转换为 device_node 结构体；
④ 对于某些 device_node 结构体，会被转换为 platform_device 结构体。

11.4.1 dtb 中每一个节点都被转换为 device_node 结构体

```
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;
    struct  property *deadprops;    /* removed properties */
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    struct  kobject kobj;
    unsigned long _flags;
    void    *data;
#if defined(CONFIG_SPARC)
    const char *path_component_name;
    unsigned int unique_id;
    struct of_irq_controller *irq_trans;
#endif
} « end device_node » ;
```

```
struct property {
    char    *name;
    int length;
    void    *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};
```

图 11.6 dtb 节点

根节点被保存在全局变量 of_root 中，从 of_root 开始可以访问到任意节点。

11.4.2 哪些设备树节点会被转换为 platform_device

① 根节点下含有 compatile 属性的子节点

② 含有特定 compatile 属性的节点的子节点

如果一个节点的 compatile 属性，它的值是这 4 者之一："simple-bus","simple-mfd","isa","arm,amba-bus"，那么它的子结点(需含 compatile 属性)也可以转换为 platform_device。

③ 总线 I2C、SPI 节点下的子节点：不转换为 platform_device

某个总线下到子节点，应该交给对应的总线驱动程序来处理，它们不应该被转换为 platform_device。

比如以下的节点中：

- /mytest 会被转换为 platform_device，因为它兼容"simple-bus"；
  它的子节点/mytest/mytest@0 也会被转换为 platform_device
- /i2c 节点一般表示 i2c 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver；
- /i2c/at24c02 节点不会被转换为 platform_device，它被如何处理完全由父节点的 platform_driver 决定，一般是被创建为一个 i2c_client。
- 类似的也有/spi 节点，它一般也是用来表示 SPI 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver；
- /spi/flash@0 节点不会被转换为 platform_device，它被如何处理完全由父节点的 platform_driver 决定，一般是被创建为一个 spi_device。

```
{
    mytest {
        compatile = "mytest", "simple-bus";
```

```
        mytest@0 {
              compatile = "mytest_0";
        };
    };

    i2c {
        compatile = "samsung,i2c";
        at24c02 {
              compatile = "at24c02";
        };
    };

    spi {
        compatile = "samsung,spi";
        flash@0 {
              compatible = "winbond,w25q32dw";
              spi-max-frequency = <25000000>;
              reg = <0>;
            };
    };
};
```

11.4.3 怎么转换为 platform_device

内核处理设备树的函数调用过程，这里不去分析；我们只需要得到如下结论：

- platform_device 中含有 resource 数组， 它来自 device_node 的 reg，interrupts 属性；
- platform_device.dev.of_node 指向 device_node，可以通过它获得其他属性

11.5 platform_device 如何与 platform_driver 配对

从设备树转换得来的 platform_device 会被注册进内核里，以后当我们每注册一个 platform_driver 时，它们就会两两确定能否配对，如果能配对成功就调用 platform_driver 的 probe 函数。

套路是一样的。我们需要将前面讲过的“匹配规则”再完善一下，先贴源码：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev = to_platform_device(dev);
        struct platform_driver *pdrv = to_platform_driver(drv);

        /* When driver_override is set, only bind to the matching driver */
①       if (pdev->driver_override)
                return !strcmp(pdev->driver_override, drv->name);

        /* Attempt an OF style match first */
②       if (of_driver_match_device(dev, drv))
                return 1;

        /* Then try ACPI style match */
        if (acpi_driver_match_device(dev, drv))
                return 1;

        /* Then try to match against the id table */
③       if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;

        /* fall-back to driver name match */
④       return (strcmp(pdev->name, drv->name) == 0);
}
```

图 11.7 platform_match 源码

11.5.1 最先比较：是否强制选择某个 driver

- 比较：

platform_device.driver_override 和 platform_driver.driver.name

可以设置 platform_device 的 driver_override，强制选择某个 platform_driver。

11.5.2 然后比较：设备树信息

- 比较：

platform_device.dev.of_node 和 platform_driver.driver.of_match_table。

由设备树节点转换得来的 platform_device 中，含有一个结构体：of_node。它的类型如下：

```
struct device_node {
    const char *name; 来自节点的name属性
    const char *type; 来自节点的device_type属性
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;含有compatible属性
```

如果一个 platform_driver 支持设备树，它的 platform_driver.driver.of_match_table 是一个数组，类型如下：

```
struct of_device_id {
    char    name[32];
    char    type[32];
    char    compatible[128];
    const void *data;
};
```

使用设备树信息来判断 dev 和 drv 是否配对时：

① 首先，如果 of_match_table 中含有 compatible 值，就跟 dev 的 compatile 属性比较，若一致则成功，否则返回失败；

② 其次，如果 of_match_table 中含有 type 值，就跟 dev 的 device_type 属性比较，若一致则成功，否则返回失败；

③ 最后，如果 of_match_table 中含有 name 值，就跟 dev 的 name 属性比较，若一致则成功，否则返回失败。

而设备树中建议不再使用 devcie_type 和 name 属性，所以基本上只使用设备节点的 compatible 属性来寻找匹配的 platform_driver。

11.5.3 接下来比较：platform_device_id

比较 platform_device. name 和 platform_driver.id_table[i].name，id_table 中可能有多项。

platform_driver.id_table 是“platform_device_id”指针，表示该 drv 支持若干个 device，它里面列出了各个 device 的{.name, .driver_data}，其中的“name”表示该 drv 支持的设备的名字，driver_data 是些提供给该 device 的私有数据。

11.5.4 最后比较

- platform_device.name 和 platform_driver.driver.name

  platform_driver.id_table 可能为空，

  这时可以根据 platform_driver.driver.name 来寻找同名的 platform_device。

11.5.5 一个图概括所有的配对过程

概括出了这个图：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev = to_platform_device(dev);
        struct platform_driver *pdrv = to_platform_driver(drv);

        /* When driver_override is set, only bind to the matching driver */
①       if (pdev->driver_override)
                return !strcmp(pdev->driver_override, drv->name);

        /* Attempt an OF style match first */
②       if (of_driver_match_device(dev, drv))
                return 1;

        /* Then try ACPI style match */
        if (acpi_driver_match_device(dev, drv))
                return 1;

        /* Then try to match against the id table */
③       if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;

        /* fall-back to driver name match */
④       return (strcmp(pdev->name, drv->name) == 0);
}
```

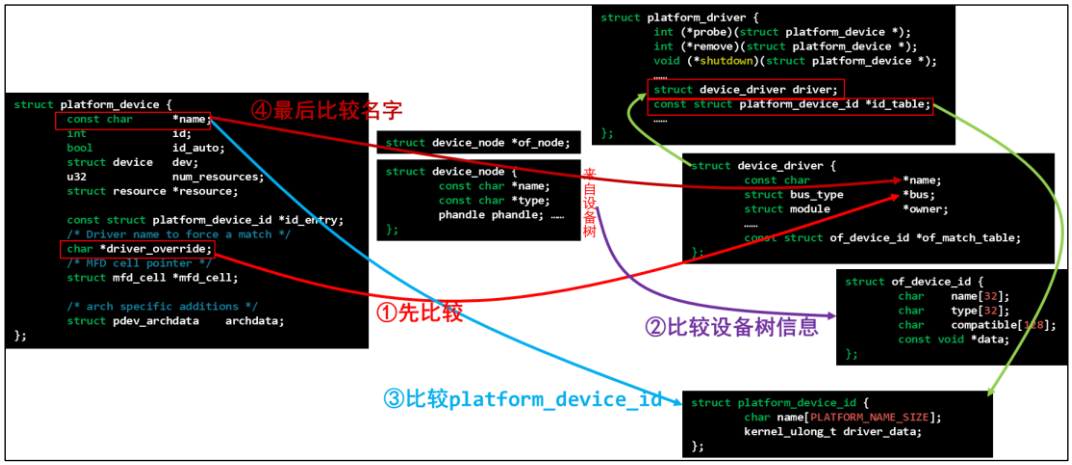


图 11.8 配对过程

11.6 没有转换为 platform_device 的节点，如何使用

任意驱动程序里，都可以直接访问设备树。

你可以使用<>>>>>>>>第五篇 11.7 内核里操作设备树的常用函数>节中介绍的函数找到节点，读出里面的值。

11.7 内核里操作设备树的常用函数

内核源码中 include/linux/目录下有很多 of 开头的头文件，of 表示“open firmware”即开放固件。

11.7.1 内核中设备树相关的头文件介绍

设备树的处理过程是：dtb -> device_node -> platform_device。

1 处理 DTB

```
of_fdt.h            // dtb 文件的相关操作函数，我们一般用不到，
// 因为 dtb 文件在内核中已经被转换为 device_node 树(它更易于使用)
```

2 处理 device_node

```
of.h                // 提供设备树的一般处理函数，
// 比如 of_property_read_u32(读取某个属性的 u32 值)，
// of_get_child_count(获取某个 device_node 的子节点数)
of_address.h        // 地址相关的函数，
// 比如 of_get_address(获得 reg 属性中的 addr，size 值)
// of_match_device (从 matches 数组中取出与当前设备最匹配的一项)
of_dma.h            // 设备树中 DMA 相关属性的函数
of_gpio.h           // GPIO 相关的函数
of_graph.h          // GPU 相关驱动中用到的函数，从设备树中获得 GPU 信息
of_iommu.h          // 很少用到
of_irq.h            // 中断相关的函数
of_mdio.h           // MDIO (Ethernet PHY) API
of_net.h            // OF helpers for network devices.
of_pci.h            // PCI 相关函数
of_pdt.h            // 很少用到
of_reserved_mem.h   // reserved_mem 的相关函数
```

3 处理 platform_device

```
of_platform.h       // 把 device_node 转换为 platform_device 时用到的函数，
                    // 比如 of_device_alloc(根据 device_node 分配设置 platform_device)，
                    // of_find_device_by_node (根据 device_node 查找到 platform_device)，
                    //   of_platform_bus_probe (处理 device_node 及它的子节点)
of_device.h         // 设备相关的函数，比如 of_match_device
```

11.7.2 platform_device 相关的函数

of_platform.h 中声明了很多函数，但是作为驱动开发者，我们只使用其中的 1、2 个。其他的都是给内核自己使用的，内核使用它们来处理设备树，转换得到 platform_device。

1 of_find_device_by_node

函数原型为：

```
extern struct platform_device *of_find_device_by_node(struct device_node *np);
```

设备树中的每一个节点，在内核里都有一个 device_node；你可以使用 device_node 去找到对应的 platform_device。

2 platform_get_resource

这个函数跟设备树没什么关系，但是设备树中的节点被转换为 platform_device 后，设备树中的 reg 属性、interrupts 属性也会被转换为

“resource”。

这时，你可以使用这个函数取出这些资源。

函数原型为：

```
/**
 * platform_get_resource - get a resource for a device
 * @dev: platform device
 * @type: resource type   // 取哪类资源？IORESOURCE_MEM、IORESOURCE_REG
*                        // IORESOURCE_IRQ 等
 * @num: resource index  // 这类资源中的哪一个？
 */
struct resource *platform_get_resource(struct platform_device *dev,
                       unsigned int type, unsigned int num);
```

对于设备树节点中的 reg 属性，它对应 IORESOURCE_MEM 类型的资源；

对于设备树节点中的 interrupts 属性，它对应 IORESOURCE_IRQ 类型的资源。

11.7.3 有些节点不会生成 platform_device，怎么访问它们

内核会把 dtb 文件解析出一系列的 device_node 结构体，我们可以直接访问这些 device_node。

内核源码 incldue/linux/of.h 中声明了 device_node 和属性 property 的操作函数，device_node 和 property 的结构体定义如下：

```
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct  property *properties;
    struct  property *deadprops;    /* removed properties */
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    struct  kobject kobj;
    unsigned long _flags;
    void    *data;
#if defined(CONFIG_SPARC)
    const char *path_component_name;
    unsigned int unique_id;
    struct of_irq_controller *irq_trans;
#endif
} « end device_node » ;
```

```
struct property {
    char    *name;
    int length;
    void    *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};
```

图 11.9 device_node 和 property

1 找到节点

① of_find_node_by_path

根据路径找到节点，比如“/”就对应根节点，“/memory”对应 memory 节点。

函数原型：

```
static inline struct device_node *of_find_node_by_path(const char *path);
```

② of_find_node_by_name

根据名字找到节点，节点如果定义了 name 属性，那我们可以根据名字找到它。

函数原型：

```
extern struct device_node *of_find_node_by_name(struct device_node *from,const char *name);
```

参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“name”属性，所以这函数也不建议使用。

③ of_find_node_by_type

根据类型找到节点，节点如果定义了 device_type 属性，那我们可以根据类型找到它。

函数原型：

```
extern struct device_node *of_find_node_by_type(struct device_node *from, const char *type);
```

参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“device_type”属性，所以这函数也不建议使用。

④ of_find_compatible_node

根据 compatible 找到节点，节点如果定义了 compatible 属性，那我们可以根据 compatible 属性找到它。

函数原型：

```
extern struct device_node *of_find_compatible_node(struct device_node *from, cons
                                                   t char *type,
                                                 const char *compat);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。
- 参数 compat 是一个字符串，用来指定 compatible 属性的值；
- 参数 type 是一个字符串，用来指定 device_type 属性的值，可以传入 NULL。

⑤ of_find_node_by_phandle

根据 phandle 找到节点。dts 文件被编译为 dtb 文件时，每一个节点都有一个数字 ID，这些数字 ID 彼此不同。可以使用数字 ID 来找到 device_node。这些数字 ID 就是 phandle。

函数原型：

```
extern struct device_node *of_find_node_by_phandle(phandle handle);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑥ of_get_parent

找到 device_node 的父节点。

函数原型：

```
extern struct device_node *of_get_parent(const struct device_node *node);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑦ of_get_next_parent

这个函数名比较奇怪，怎么可能有“next parent”？

它实际上也是找到 device_node 的父节点，跟 of_get_parent 的返回结果是一样的。

差别在于它多调用下列函数，把 node 节点的引用计数减少了 1。这意味着

调用 of_get_next_parent 之后，你不再需要调用 of_node_put 释放 node 节点。

```
of_node_put(node);
```

函数原型：

```
extern struct device_node *of_get_next_parent(struct device_node *node);
```

- 参数 from 表示从哪一个节点开始寻找，传入 NULL 表示从根节点开始寻找。

⑧ of_get_next_child

取出下一个子节点。

函数原型：

```
extern struct device_node *of_get_next_child(const struct device_node *node,
                                             struct device_node *prev);
```

- 参数 node 表示父节点；
- prev 表示上一个子节点，设为 NULL 时表示想找到第 1 个子节点。

不断调用 of_get_next_child 时，不断更新 pre 参数，就可以得到所有的子节点。

⑨ of_get_next_available_child

取出下一个“可用”的子节点，有些节点的 status 是“disabled”，那就会跳过这些节点。

函数原型：

```
struct device_node *of_get_next_available_child( const struct device_node *node,
                                                 struct device_node *prev);
```

- 参数 node 表示父节点；
- prev 表示上一个子节点，设为 NULL 时表示想找到第 1 个子节点。

⑩ of_get_child_by_name

根据名字取出子节点。

函数原型：

```
extern struct device_node *of_get_child_by_name(const struct device_node *node,
                                                const char *name);
```

- 参数 node 表示父节点；
- name 表示子节点的名字。

2 找到属性——of_find_property

内核源码 incldue/linux/of.h 中声明了 device_node 的操作函数，当然也包括属性的操作函数：of_find_property

找到节点中的属性。

函数原型：

```
extern struct property *of_find_property(const struct device_node *np,
                                         const char *name,
                                         int *lenp);
```

- 参数 np 表示节点，我们要在这个节点中找到名为 name 的属性。
- lenp 用来保存这个属性的长度，即它的值的长度。

在设备树中，节点大概是这样：

```
xxx_node {
    xxx_pp_name = “hello”;
};
```
上述节点中，“xxx_pp_name”就是属性的名字，值的长度是 6。

3 获取属性的值

① of_get_property

根据名字找到节点的属性，并且返回它的值。

函数原型：

```
/*
 * Find a property with a given name for a given node
 * and return the value.
 */
const void *of_get_property(const struct device_node *np,
                            const char *name,
                            int *lenp)
```

- 参数 np 表示节点，我们要在这个节点中找到名为 name 的属性，然后返回它的值。
- lenp 用来保存这个属性的长度，即它的值的长度。

② of_property_count_elems_of_size

根据名字找到节点的属性，确定它的值有多少个元素(elem)。

函数原型：

```
* of_property_count_elems_of_size - Count the number of elements in a property
 *
 * @np:    device node from which the property value is to be read.
 * @propname:  name of the property to be searched.
 * @elem_size: size of the individual element
 *
 * Search for a property in a device node and count the number of elements of
 * size elem_size in it. Returns number of elements on sucess, -EINVAL if the
 * property does not exist or its length does not match a multiple of elem_size
 * and -ENODATA if the property does not have a value.
 */
int of_property_count_elems_of_size(const struct device_node *np,
                                    const char *propname,
                                    int elem_size)
```

③ 参数 np 表示节点，我们要在这个节点中找到名为 propname 的属性，然后返回下列结果：

```
return prop->length / elem_size;
```

在设备树中，节点大概是这样：

```
xxx_node {
    xxx_pp_name = <0x50000000 1024>  <0x60000000  2048>;
};
```

- 调用 `of_property_count_elems_of_size(np, “xxx_pp_name”, 8)`时，返回值是 2；
- 调用 `of_property_count_elems_of_size(np, “xxx_pp_name”, 4)`时，返回值是 4。

④ 读整数 u32/u64

函数原型为：

```
static inline int of_property_read_u32(const struct device_node *np,
```

```
                      const char *propname,
                      u32 *out_value);

extern int of_property_read_u64(const struct device_node *np,
                                const char *propname,
                                u64 *out_value);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name1 = <0x50000000>;
    name2 = <0x50000000  0x60000000>;
};
```

- 调用 `of_property_read_u32 (np, “name1”, &val)`时，val 将得到值 0x50000000；
- 调用 `of_property_read_u64 (np, “name2”, &val)`时，val 将得到值 0x6000000050000000。

⑤ 读某个整数 u32/u64

函数原型为：

```
extern int of_property_read_u32_index(const struct device_node *np,
                                      const char *propname,
                                      u32 index, u32 *out_value);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name2 = <0x50000000  0x60000000>;
};
```

- 调用 `of_property_read_u32 (np, “name2”, 1, &val)`时，val 将得到值 0x60000000。

⑥ 读数组

函数原型为：

```
int of_property_read_variable_u8_array(const struct device_node *np,
                                       const char *propname,
                                       u8 *out_values,
                                       size_t sz_min, size_t sz_max);

int of_property_read_variable_u16_array(const struct device_node *np,
                                       const char *propname,
                                       u16 *out_values,
                                       size_t sz_min, size_t sz_max);

int of_property_read_variable_u32_array(const struct device_node *np,
                                       const char *propname,
                                       u32 *out_values,
                                       size_t sz_min, size_t sz_max);

int of_property_read_variable_u64_array(const struct device_node *np,
                                       const char *propname,
                                       u64 *out_values,
                                       size_t sz_min, size_t sz_max);
```

在设备树中，节点大概是这样：

```
xxx_node {
    name2 = <0x50000012  0x60000034>;
};
```

上述例子中属性 name2 的值，长度为 8。

- 调用 `of_property_read_variable_u8_array (np, “name2”, out_values, 1, 10)`时，`out_values` 中将会保存这 8 个字节： `0x12,0x00,0x00,0x50,0x34,0x00,0x00,0x60`。

- 调用 `of_property_read_variable_u16_array (np, “name2”, out_values, 1, 10)`时，out_values 中将会保存这 4 个 16 位数值： `0x0012, 0x5000,0x0034,0x6000。`

总之，这些函数要么能取到全部的数值，要么一个数值都取不到；

- 如果值的长度在 sz_min 和 sz_max 之间，就返回全部的数值；
- 否则一个数值都不返回。

⑦ 读字符串

函数原型为：

```
int of_property_read_string(const struct device_node *np,
                            const char *propname,
                            const char **out_string);
```

- 返回节点 np 的属性(名为 propname)的值；
- (*out_string)指向这个值，把它当作字符串。

11.8 怎么修改设备树文件

一个写得好的驱动程序，它会尽量确定所用资源。只把不能确定的资源留给设备树，让设备树来指定。根据原理图确定"驱动程序无法确定的硬件资源"，再在设备树文件中填写对应内容。

那么，所填写内容的格式是什么？

11.8.1 使用芯片厂家提供的工具

有些芯片，厂家提供了对应的设备树生成工具，可以选择某个引脚用于某些功能，就可以自动生成设备树节点。

你再把这些节点复制到内核的设备树文件里即可。

11.8.2 看绑定文档

内核文档 Documentation/devicetree/bindings/

做得好的厂家也会提供设备树的说明文档

11.8.3 参考同类型单板的设备树文件

11.8.4 网上搜索

11.8.5 实在没办法时，只能去研究驱动源码

第12章 LED 模板驱动程序的改造：设备树

12.1 总结 3 种写驱动程序的方法

12.1.1 资源和驱动在同一个文件里

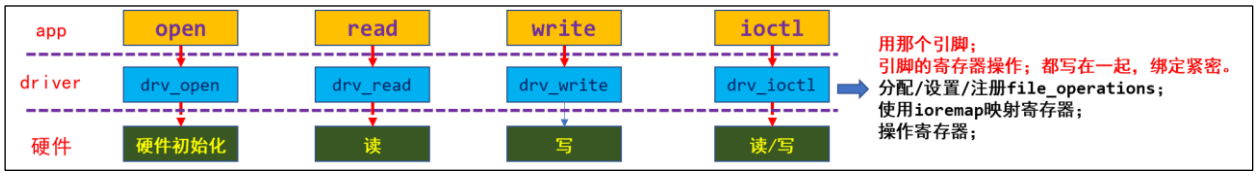


图 12.1 方法 1 传统写法

12.1.2 资源用 platform_device 指定、驱动在 platform_driver 实现

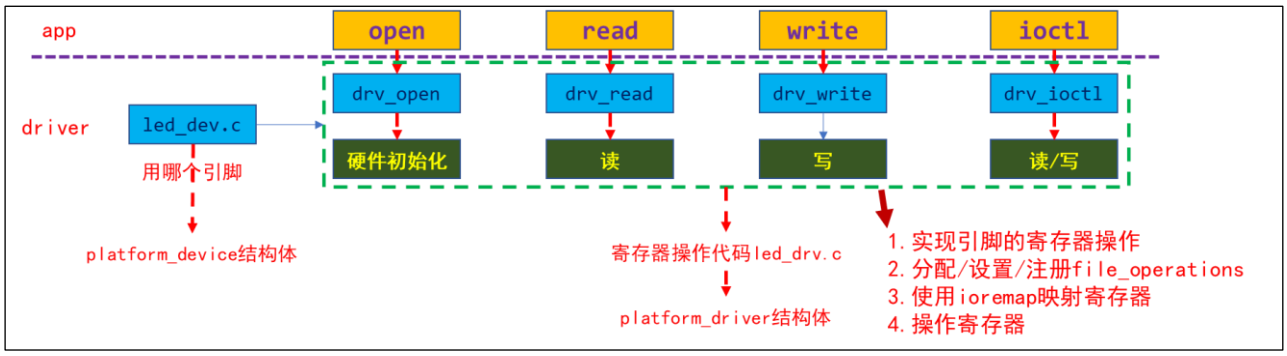


图 12.2 方法 2 分层分离

12.1.3 资源用设备树指定驱动在 platform_driver 实现

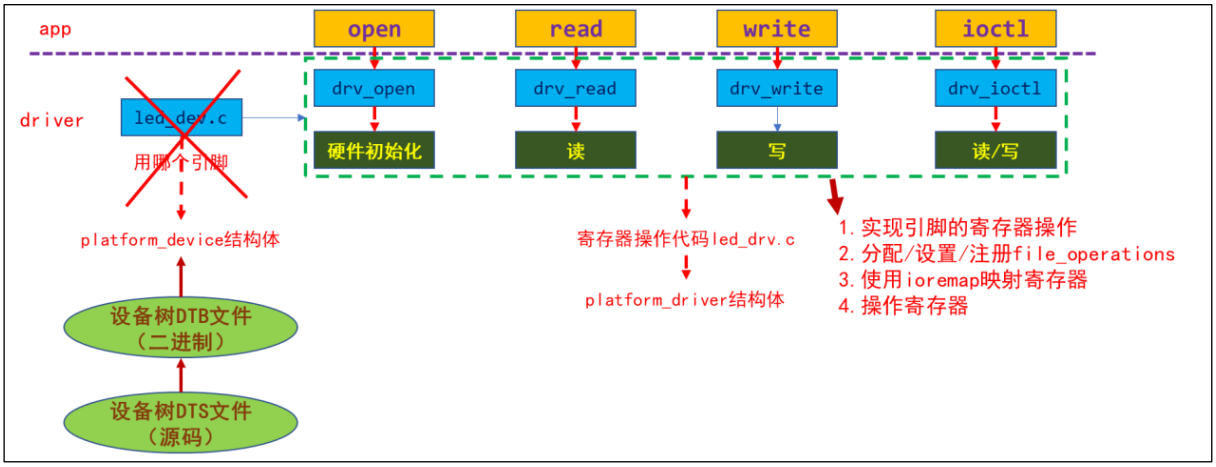


图 12.3 方法 3 使用设备树

核心永远是 file_operations 结构体。

上述三种方法，只是指定“硬件资源”的方式不一样。

从图 12.3 可以知道，platform_device/platform_driver 只是编程的技巧，不涉及驱动的核心。

12.2 怎么使用设备树写驱动程序

12.2.1 设备树节点要与 platform_driver 能匹配

在我们的工作中，驱动要求设备树节点提供什么，我们就得按这要求去编写设备树。

但是，匹配过程所要求的东西是固定的：

① 设备树要有 compatible 属性，它的值是一个字符串

② platform_driver 中要有 of_match_table，其中一项的.compatible 成员设置为一个字符串

③ 上述 2 个字符串要一致。

示例如下：

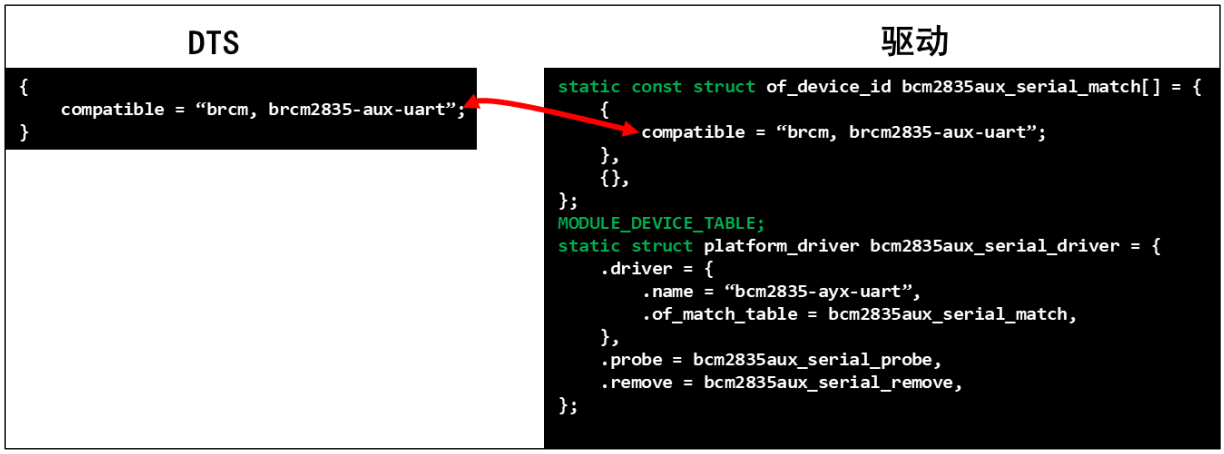


图 12.4 设备树驱动示例

12.2.2 设备树节点指定资源，platform_driver 获得资源

如果在设备树节点里使用 reg 属性，那么内核生成对应的 platform_device 时会用 reg 属性来设置 IORESOURCE_MEM 类型的资源。

如果在设备树节点里使用 interrupts 属性，那么内核生成对应的 platform_device 时会用 reg 属性来设置 IORESOURCE_IRQ 类型的资源。对于 interrupts 属性，内核会检查它的有效性，所以不建议在设备树里使用该属性来表示其他资源。

在我们的工作中，驱动要求设备树节点提供什么，我们就得按这要求去编写设备树。驱动程序中根据 pin 属性来确定引脚，那么我们就在设备树节点中添加 pin 属性。

设备树节点中：

```
#define GROUP_PIN(g,p) ((g<<16) | (p))
100ask_led0 {
   compatible = “100ask,led”;
   pin = <GROUP_PIN(5, 3)>;
};
```

驱动程序中，可以从 platform_device 中得到 device_node，再用

of_property_read_u32 得到属性的值：

```
struct  device_node* np = pdev->dev. of_node;
int led_pin;
int err = of_property_read_u32(np, “pin”, &led_pin);
```

12.3 开始编程

12.3.1 修改设备树添加 led 设备节点

在本实验中，需要添加的设备节点代码是一样的，你需要找到你的单板所用的设备树文件，在它的根节点下添加如下内容：

```
#define GROUP_PIN(g,p) ((g<<16) | (p))
100ask_led@0 {
    compatible = "100ask,leddrv";
    pin = <GROUP_PIN(3, 1)>;
};

100ask_led@1 {
    compatible = "100as,leddrv";
    pin = <GROUP_PIN(5, 8)>;
};
```

- 对百问网 imx6ull Pro 板

  设备树文件是：内核源码目录中 `arch/arm/boot/dts/100ask_imx6ull-14x14.dts`

  修改、编译后得到 `arch/arm/boot/dts/100ask_imx6ull-14x14.dtb` 文件。

  对于这款板子，本教程中我们使用 SD 卡上的系统。

  要更换板上的设备树文件，你可以使用 SD 卡启动开发板后，更换这个文件：`/boot/100ask_imx6ull-14x14.dtb`

12.3.2 修改 platform_driver 的源码

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\02_led_drv\05_led_drv_template_device_tree
```

关键代码在 chip_demo_gpio.c，主要看里面的 platform_driver，代码如下。

第 166 行指定了 of_match_table，它是用来跟设备树节点匹配的，如果设备树节点中有 compatile 属性，并且其值等于第 157 行的“100as,leddrv”，就会导致第 162 行的 probe 函数被调用。

```
156 static const struct of_device_id ask100_leds[] = {
157     { .compatible = "100as,leddrv" },
158     { },
159 };
160
161 static struct platform_driver chip_demo_gpio_driver = {
162     .probe      = chip_demo_gpio_probe,
163     .remove     = chip_demo_gpio_remove,
164     .driver     = {
```

```
165         .name   = "100ask_led",
166         .of_match_table = ask100_leds,
167     },
168 };
169
170 static int __init chip_demo_gpio_drv_init(void)
171 {
172     int err;
173
174     err = platform_driver_register(&chip_demo_gpio_driver);
175     register_led_operations(&board_demo_led_opr);
176
177     return 0;
178 }
```

12.4 上机实验

① 使用新的设备树 dtb 文件启动单板，查看/sys/firmware/devicetree/base 下有无节点

② 查看/sys/devices/platform 目录下有无对应的 platform_device

③ 加载驱动：

```
[root@100ask:~]# insmod  leddrv.ko
[root@100ask:~]# insmod  chip_demo_gpio.ko
```

④ 测试驱动：

```
[root@100ask:~]# ./ledtest   /dev/100ask_led0  on
[root@100ask:~]# ./ledtest   /dev/100ask_led0  off
```

12.5 调试技巧

/sys 目录下有很多内核、驱动的信息。

12.5.1 设备树的信息

以下目录对应设备树的根节点，可以从此进去找到自己定义的节点。

```
cd /sys/firmware/devicetree/base/
```

节点是目录，属性是文件。

属性值是字符串时，用 cat 命令可以打印出来；属性值是数值时，用 hexdump 命令可以打印出来。

12.5.2 platform_device 的信息

以下目录含有注册进内核的所有 platform_device：

```
/sys/devices/platform
```

一个设备对应一个目录，进入某个目录后，如果它有“driver”子目录，就表示这个 platform_device 跟某个 platform_driver 配对了。

比如下面的结果中，平台设备“ff8a0000.i2s”已经跟平台驱动“rockchip-i2s”配对了：

```
/sys/devices/platform/ff8a0000.i2s]# ls driver -ld
```

```
lrwxrwxrwx    1 root     root            0 Jan 18 16:28 driver -> ../../../bus/platfo
rm/drivers/rockchip-i2s
```

12.5.3 platform_driver 的信息

以下目录含有注册进内核的所有 platform_driver：

```
/sys/bus/platform/drivers
```

一个 driver 对应一个目录，进入某个目录后，如果它有配对的设备，可以直接看到。

比如下面的结果中，平台驱动“rockchip-i2s”跟 2 个平台设备“平台设备“ff890000.i2s”、“ff8a0000.i2s”配对了：

```
[root@roc-rk3399-pc:/sys/bus/platform/drivers/rockchip-i2s]# ls
bind      ff890000.i2s  ff8a0000.i2s  uevent    unbind
```

注意：一个平台设备只能配对一个平台驱动，一个平台驱动可以配对多个平台设备。

12.6 课后作业

请仿照本节提供的程序(位于 05_led_drv_template_device_tree 目录)，改造你所用的单板的 LED 驱动程序。

第13章 APP 怎么读取按键值

在做单片机开发时，要读取 GPIO 按键，我们通常是执行一个循环，不断地检测 GPIO 引脚电平有没有发生变化。但是在 Linux 系统中，读取 GPIO 按键要考虑到效率，引入了很多种方法：查询方式(非阻塞)、休眠-唤醒(阻塞方式)、poll 方式、异步通知方式。这 4 种方法并不仅仅用于 GPIO 按键，在所有的 APP 调用驱动程序过程中，都是使用这些方法。通过这 4 种方式的学习，我们可以掌握如下知识：

① 驱动的基本技能：中断、休眠、唤醒、poll 等机制。

这些基本技能是驱动开发的基础，其他大型驱动复杂的地方是它的框架及设计思想，但是基本技术就这些。

② APP 开发的基本技能：阻塞 、非阻塞、休眠、poll、异步通知。

13.1 妈妈怎么知道孩子醒了

图 13.1 示例

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式

- 简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒

- 不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式

- 要浪费点时间，但是可以继续干活。
- 妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

- 妈妈、小孩互不耽误。

这 4 种方法没有优劣之分，在不同的场合使用不同的方法。

13.2 APP 读取按键的 4 种方法

APP 去读取按键和举例的场景很相似，也有 4 种方法：

① 查询方式
② 休眠-唤醒方式
③ poll 方式
④ 异步通知方式

第 2、3、4 种方法，都涉及中断服务程序。中断，就像小孩醒了会哭闹一样，中断不经意间到来，它会做某些事情：唤醒 APP、向 APP 发信号。

所以，在按键驱动程序中，中断是核心。

实际上，中断无论是在单片机还是在 Linux 中都很重要。在 Linux 中，中断的知识还涉及进程、线程等。

13.2.1 查询方式

这种方法最简单：

图 13.2 查询方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚。APP 调用 read 时，导致驱动中对应的 read 函数被调用，它读取寄存器，把引脚状态直接返回给 APP。

13.2.2 休眠-唤醒方式

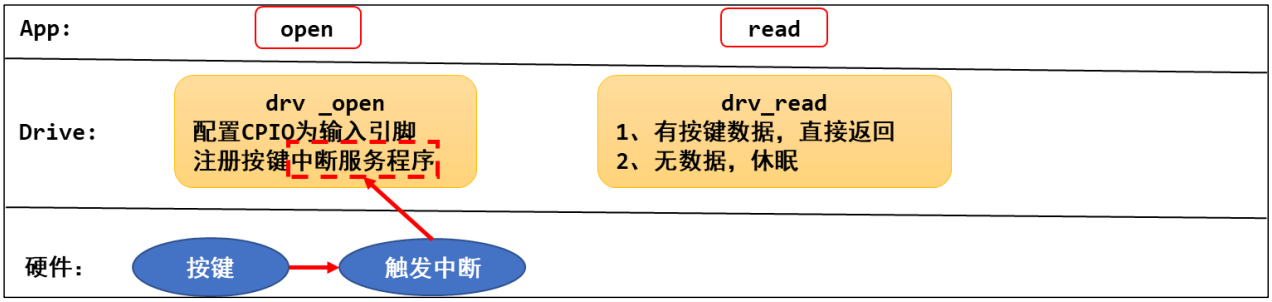


图 13.3 休眠唤醒方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 调用 read 时，导致驱动中对应的 read 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

APP 被唤醒后继续在内核态运行，即继续执行驱动代码，把按键数据返回给 APP(的用户空间)。

13.2.3 poll 方式

上面的休眠-唤醒方式有个缺点：如果用户一直没操作按键，那么 APP 就会永远休眠。

我们可以给 APP 定个闹钟，这就是 poll 方式。

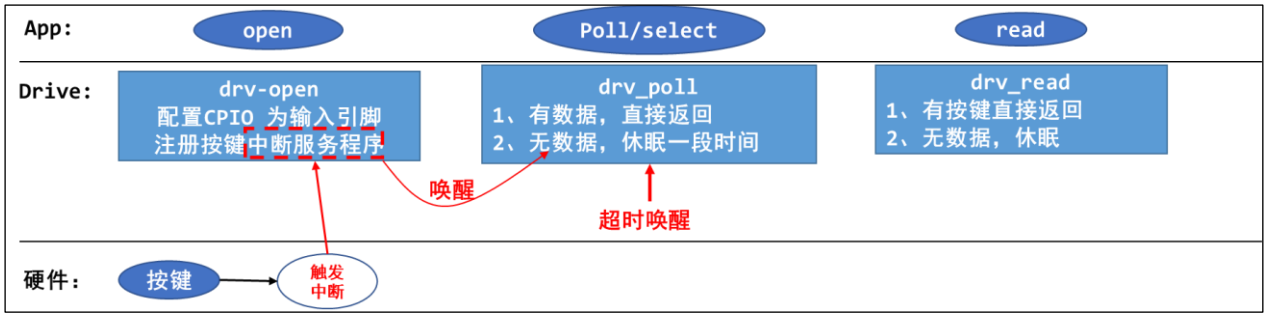


图 13.4 poll 方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,poll 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 调用 poll 或 select 函数，意图是“查询”是否有数据，这 2 个函数都可以指定一个超时时间，即在这段时间内没有数据的话就返回错误。这会导致驱动中对应的 poll 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠一段时间。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

如果用户没按下按键，但是超时时间到了，内核也会唤醒 APP。

所以 APP 被唤醒有 2 种原因：用户操作了按键，超时。被唤醒的 APP 在内核态继续运行，即继续执行驱动代码，把“状态”返回给 APP(的用户空间)。

APP 得到 poll/select 函数的返回结果后，如果确认是有数据的，则再调用 read 函数，这会导致驱动中的 read 函数被调用，这时驱动程序中含有数据，会直接返回数据。

13.2.4 异步通知方式

1 异步通知的原理：发信号

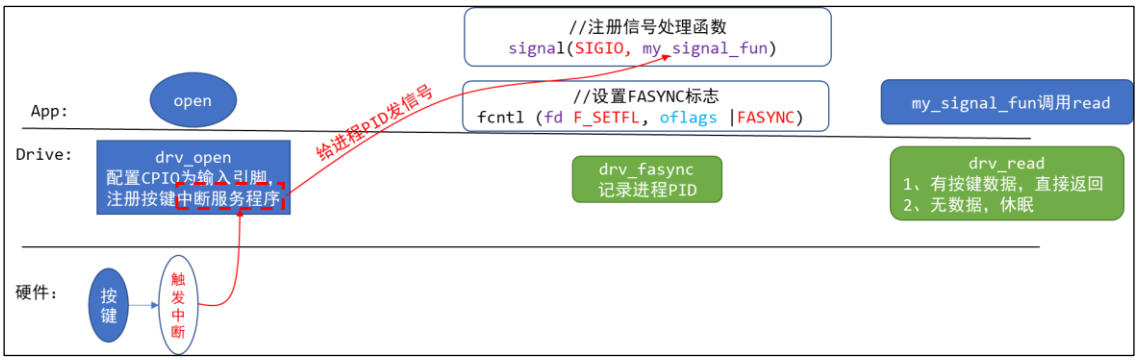


图 13.5 异步通知的原理

异步通知的实现原理是：内核给 APP 发信号。信号有很多种，这里发的是 SIGIO。

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,fasync 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 给信号 SIGIO 注册自己的处理函数：my_signal_fun。
- APP 调用 fcntl 函数，把驱动程序的 flag 改为 FASYNC，这会导致驱动程序的 fasync 函数被调用，它只是简单记录进程 PID。
- 当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，然后给进程 PID 发送 SIGIO 信号。
- APP 收到信号后会被打断，先执行信号处理函数：在信号处理函数中可以去调用 read 函数读取按键值。
- 信号处理函数返回后，APP 会继续执行原先被打断的代码。

2 应用程序之间发信号示例代码

使用 GIT 下载所有源码后，本节源码位于如下目录：

01_all_series_quickstart
05_嵌入式 Linux 驱动开发基础知识\source\03_signal_example

代码并不复杂，如下。

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <signal.h>
04 void my_sig_func(int signo)
05 {
06     printf("get a signal : %d\n", signo);
07 }
08
09 int main(int argc, char **argv)
10 {
```

```
11      int i = 0;
12
13      signal(SIGIO, my_sig_func);
14
15      while (1)
16      {
17          printf("Hello, world %d!\n", i++);
18          sleep(2);
19      }
20
21      return 0;
22  }
```

第 13 行注册信号处理函数，第 15 行就是一个无限循环。在它运行期间，你可以用另一个 APP 发信号给它。

在 Ubuntu 上的测试方法：

```
$ gcc   -o   signal  signal.c   // 编译程序
$ ./signal &                   // 后台运行
$ ps  -A | grep signal          // 查看进程 ID，假设是 9527
$ kill  -SIGIO  9527           // 给这个进程发信号
```

13.2.5 驱动程序提供能力，不提供策略

我们的驱动程序可以实现上述 4 种提供按键的方法，但是驱动程序不应该限制 APP 使用哪种方法。

这就是驱动设计的一个原理：提供能力，不提供策略。就是说，你想用哪种方法都行，驱动程序都可以提供；但是驱动程序不能限制你使用哪种方法。

第14章 查询方式的按键驱动程序_编写框架

14.1 LED 驱动回顾

对于 LED，APP 调用 open 函数导致驱动程序的 led_open 函数被调用。在里面，把 GPIO 配置为输出引脚。安装驱动程序后并不意味着会使用对应的硬件，而 APP 要使用对应的硬件，必须先调用 open 函数。所以建议在驱动程序的 open 函数中去设置引脚。

APP 继续调用 write 函数传入数值，在驱动程序的 led_write 函数根据该数值去设置 GPIO 的数据寄存器，从而控制 GPIO 的输出电平。

怎么操作寄存器？从芯片手册得到对应寄存器的物理地址，在驱动程序中使用 ioremap 函数映射得到虚拟地址。驱动程序中使用虚拟地址去访问寄存器。

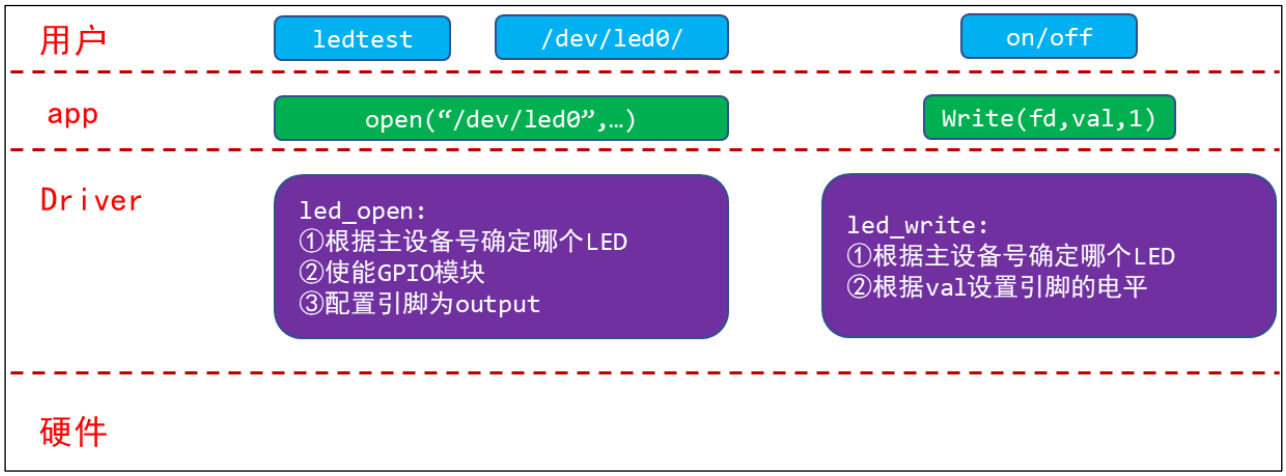


图 14.1 回顾 LED 驱动

14.2 按键驱动编写思路

GPIO 按键的原理图一般有如下 2 种：

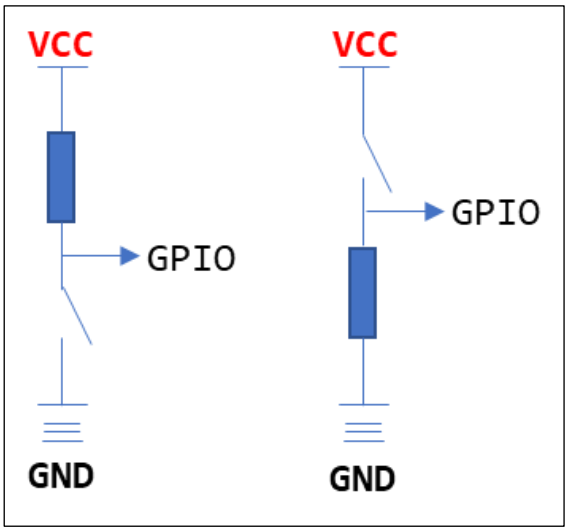


图 14.2 按键原理图示意

- 按键没被按下时，上图中左边的 GPIO 电平为高，右边的 GPIO 电平为低。
- 按键被按下后，上图中左边的 GPIO 电平为低，右边的 GPIO 电平为高。

编写按键驱动程序最简单的方法如图 14.3 所示：

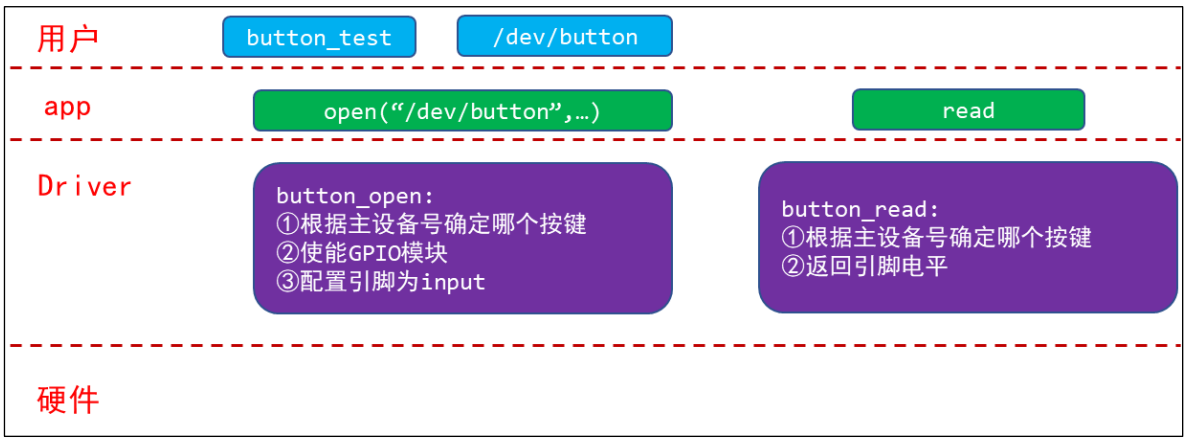


图 14.3 按键驱动编写

回顾一下编写驱动程序的套路：

图 14.4 驱动程序编写套路

对于使用查询方式的按键驱动程序，我们只需要实现 button_open、button_read。

14.3 编程：先写框架

我们的目的写出一个容易扩展到各种芯片、各种板子的按键驱动程序，所以驱动程序分为上下两层：

① button_drv.c 分配/设置/注册 file_operations 结构体

起承上启下的作用，向上提供 button_open,button_read 供 APP 调用。

而这 2 个函数又会调用底层硬件提供的 p_button_opr 中的 init、read 函数操作硬件。

② board_xxx.c 分配/设置/注册 button_operations 结构体

这个结构体是我们自己抽象出来的，里面定义单板 xxx 的按键操作函数。

这样的结构易于扩展，对于不同的单板，只需要替换 board_xxx.c 提供自己的 button_operations 结构体即可。

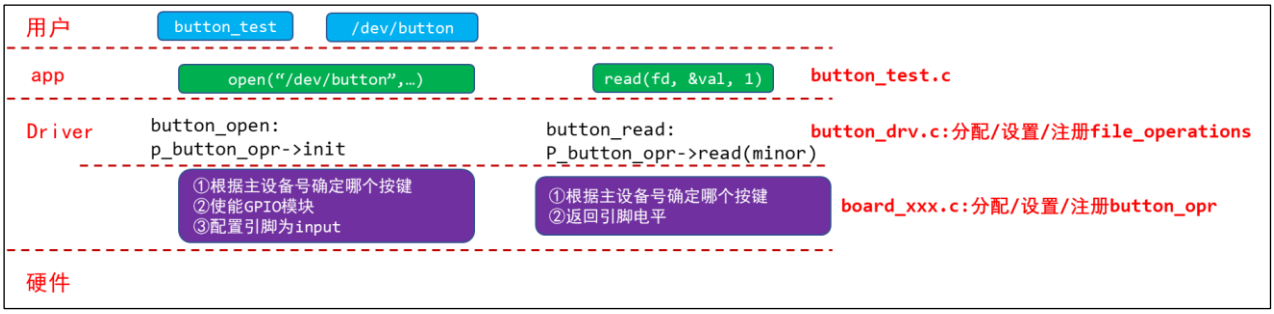


图 14.5 按键驱动框架

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\04_button_drv\01_button_drv_template
```

14.3.1 把按键的操作抽象出一个 button_operations 结构体

首先看看 button_drv.h，它定义了一个 button_operations 结构体，把按键的操作抽象为这个结构体：

```
04 struct button_operations {
05     int count;
06     void (*init) (int which);
07     int (*read) (int which);
08 };
09
10 void register_button_operations(struct button_operations *opr);
11 void unregister_button_operations(void);2
```

再看看 board_xxx.c，它实现了一个 button_operations 结构体，代码如下。

第 45 行调用 register_button_operations 函数，把这个结构体注册到上层驱动中。

```
37 static struct button_operations my_buttons_ops ={
38     .count = 2,
39     .init  = board_xxx_button_init_gpio,
40     .read  = board_xxx_button_read_gpio,
41 };
42
43 int board_xxx_button_init(void)
44 {
45     register_button_operations(&my_buttons_ops);
46     return 0;
47 }
```

14.3.2 驱动程序的上层：file_operations 结构体

上层是 button_drv.c，它的核心是 file_operations 结构体，首先看看入口函数，代码如下。

第 83 行向内核注册一个 file_operations 结构体。

第 85 行创建一个 class，但是该 class 下还没有 device，在后面获得底层硬件的信息时再在 class 下创建 device：这只是用来创建设备节点，它不是驱动程序的核心。

```
81 int button_init(void)
82 {
83     major = register_chrdev(0, "100ask_button", &button_fops);
84
85     button_class = class_create(THIS_MODULE, "100ask_button");
86     if (IS_ERR(button_class))
87         return -1;
88
89     return 0;
90 }
```

再来看看 button_drv.c 中 file_operations 结构体的成员函数，代码如下。

第 34、44 行都用到一个 button_operations 指针，它是从何而来？

```
28 static struct button_operations *p_button_opr;
29 static struct class *button_class;
30
31 static int button_open (struct inode *inode, struct file *file)
32 {
33     int minor = iminor(inode);
34     p_button_opr->init(minor);
35     return 0;
36 }
37
38 static ssize_t button_read (struct file *file, char __user *buf, size_t size, loff
_t *off)
39 {
40     unsigned int minor = iminor(file_inode(file));
41     char level;
42     int err;
43
44     level = p_button_opr->read(minor);
45     err = copy_to_user(buf, &level, 1);
46     return 1;
47 }
48
49
50 static struct file_operations button_fops = {
51     .open = button_open,
52     .read = button_read,
53 };
```

上面第 34、44 行都用到一个 button_operations 指针，来自于底层硬件相关的代码。

底层代码调用 register_button_operations 函数，向上提供这个结构体指针。

register_button_operations 函数代码如下，它还根据底层提供的 button_operations 调用 device_create，这是创建设备节点(第 62 行)。

```
55 void register_button_operations(struct button_operations *opr)
56 {
57     int i;
58
59     p_button_opr = opr;
60     for (i = 0; i < opr->count; i++)
61     {
62         device_create(button_class, NULL, MKDEV(major, i), NULL, "100ask_button%d", i);
63     }
64 }
```

14.4 测试

这只是一个示例程序，还没有真正操作硬件。测试程序操作驱动程序时，只会导致驱动程序中打印信息。

首先设置交叉工具链，修改驱动 Makefile 中内核的源码路径，编译驱动和测试程序。

启动开发板后，通过 NFS 访问编译好驱动程序、测试程序，就可以在开发板上如下操作了：

```
[root@100ask:~]# insmod button_drv.ko   // 装载驱动程序
[  435.276713] button_drv: loading out-of-tree module taints kernel.
[root@100ask:~]# insmod board_xxx.ko
[root@100ask:~]# ls /dev/100ask_button* -l     // 查看设备节点
crw-------    1 root     root      236,   0 Jan 18 08:57 /dev/100ask_button0
crw-------    1 root     root      236,   1 Jan 18 08:57 /dev/100ask_button1
[root@100ask:~]#./button_test /dev/100ask_button0    // 读按键
[  450.886180] /home/book/source/04_button_drv/01_button_drv_template/board_xxx.c bo
ard_xxx_button_init_gpio 28, init gpio for button 0
[  450.910915] /home/book/source/04_button_drv/01_button_drv_template/board_xxx.c bo
ard_xxx_button_read_gpio 33, read gpio for button 0
get button : 1    // 得到数据
```

14.5 课后怎业

合并 LED、BUTTON 框架驱动程序：01_led_drv_template、01_button_drv_template，合并为：gpio_drv_template

第15章 具体单板的按键驱动程序(查询方式)

15.1 GPIO 操作回顾

参考章节《>>>>>>>>第五篇第 3 章普适的 GPIO 引脚操作方法》、《>>>>>>>>第五篇第 4 章具体单板的 GPIO 操作方法》。

① 使能电源/时钟控制器；
② 配置引脚模式；
③ 配置引脚方向——输入/输出；
④ 输出电平/读取电平；

15.2 百问网 IMX6ULL 的按键驱动程序(查询方式)

15.2.1 先看原理图确定引脚及操作方法

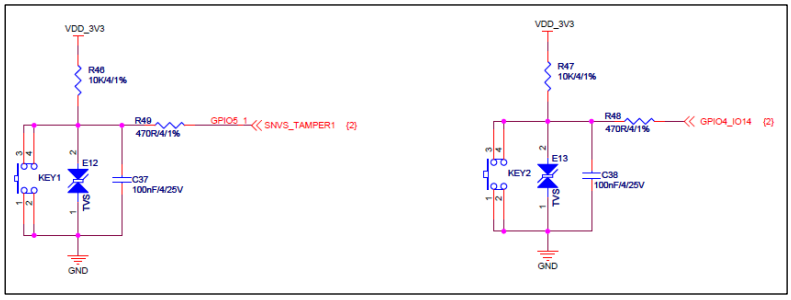


图 15.1 按键原理图

- 平时按键电平为高，按下按键后电平为低。
- 按键引脚为 GPIO5_IO01、GPIO4_IO14。

注意：视频里使用 QEMU 来做实验，QEMU 里的按键平时为低电平，按下后为高电平，跟实际开发板刚好相反。

15.2.2 再看芯片手册确定寄存器及操作方法

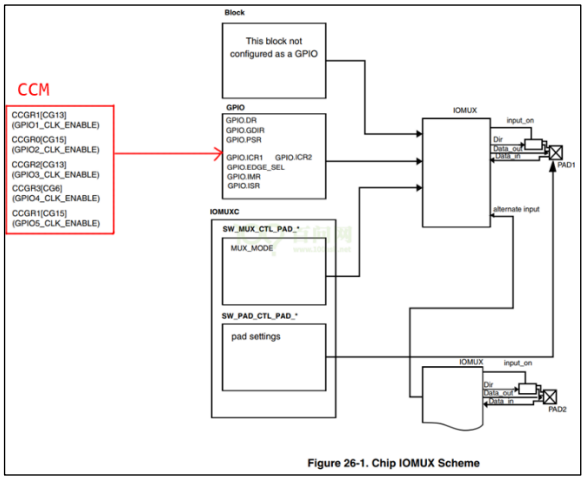


图 15.2 GPIO 寄存器控制示意图

第1步 使能 GPIO5

18.6.24 CCM Clock Gating Register 1 (CCM_CCGR1)

The figure below represents the CCM Clock Gating Register 1(CCM_CCGR1). The clock gating registers define the clock gating for power reduction of each clock (CG(i) bits). There are 8 CGR registers. The number of registers required is determined by the number of peripherals in the system.

Address: 20C_4000h base + 6Ch offset = 20C_406Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR1 field descriptions

Field	Description
31–30 CG15	Reserved 地址保留，默认使能GPIO5

图 15.3 使能 GPIO5

第2步 使能 GPIO4

18.6.26 CCM Clock Gating Register 3 (CCM_CCGR3)

The figure below represents the CCM Clock Gating Register 3 (CCM_CCGR3). The clock gating Registers define the clock gating for power reduction of each clock (CG(i) bits). There are 8 CGR registers. The number of registers required is determined by the number of peripherals in the system.

Address: 20C_4000h base + 74h offset = 20C_4074h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR3 field descriptions (continued)

Field	Description
17–16 CG8	wdog1 clock (wdog1_clk_enable)
15–14 CG7	qspi clock (qspi_clk_enable)
13–12 CG6	gpio4 clock (gpio4_clk_enable)
11–10 CG5	lcdif1 pix clock (lcdif1_pix_clk_enable)
9–8 CG4	CA7 CCM DAP clock (ccm_dap_clk_enable)
7–6 CG3	uart6 clock (uart6_clk_enable)
5–4 CG2	epdc clock (epdc_clk_enable)
3–2 CG1	uart5 clock (uart5_clk_enable)
CG0	Reserved

图 15.4 使能 GPIO4

设置 CCM_CCGR1 b[31:30]、CCM_CCGR3 b[13:12]就可以使能 GPIO5、GPIO4，设置为什么值呢？

注意：在 imx6ullrm.pdf 中，CCM_CCGR1 的 b[31:30]是保留位；我以前写程

序时错用了 imx6ul(不是 imx6ull)的手册，导致程序中额外操作了这些保留位。不去设置 b[31:30]，GPIO5 也是默认使能的。

看下图，设置为 0b11：

CGR value	Clock Activity Description
00	Clock is off during all modes. Stop enter hardware handshake is disabled.
01	Clock is on in run mode, but off in WAIT and STOP modes
10	Not applicable (Reserved).
11	Clock is on during all modes, except STOP mode.

① 00：该 GPIO 模块全程被关闭

② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭

③ 10：保留

④ 11：该 GPIO 模块全程使能

第3步 设置 GPIO5_IO01、GPIO4_IO18 为 GPIO 模式

① 对于 GPIO5_IO01，设置如下寄存器：

Address: 229_0000h base + Ch offset = 229_000Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER1 field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad SNVS_TAMPER1 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	**NOTE:** ALT5 mode is only valid when TAMPER PIN is used as GPIO. This depends on FUSE setting "TAMPER_PIN_DISABLE[1:0]". Following is the mux information when TAMPER PIN is used as GPIO: SNVS_TAMPER1 ==> GPIO5_01 101 **ALT5** — Select mux mode: ALT5 mux port, GPIO5_IO01 of instance - gpio5 Other Reserved

图 15.5 设置 GPIO5_IO1

② 对于 GPIO4_IO14，设置如下寄存器：

32.6.92 SW_MUX_CTL_PAD_NAND_CE1_B SW MUX Control Register (IOMUXC_SW_MUX_CTL_PAD_NAND_CE1_B)

SW_MUX_CTL Register

Address: 20E_0000h base + 1B0h offset = 20E_01B0h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

IOMUXC_SW_MUX_CTL_PAD_NAND_CE1_B field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad NAND_CE1_B 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	MUX Mode Select Field. Select 1 of 9 iomux modes to be used for pad: NAND_CE1_B. 0000 **ALT0** — Select mux mode: ALT0 mux port: RAWNAND_CE1_B of instance: rawnand 0001 **ALT1** — Select mux mode: ALT1 mux port: USDHC1_DATA6 of instance: usdhc1 0010 **ALT2** — Select mux mode: ALT2 mux port: QSPI_A_DATA02 of instance: qspi 0011 **ALT3** — Select mux mode: ALT3 mux port: ECSPI3_MOSI of instance: ecspi3 0100 **ALT4** — Select mux mode: ALT4 mux port: EIM_ADDR18 of instance: eim 0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO4_IO14 of instance: gpio4 1000 **ALT8** — Select mux mode: ALT8 mux port: UART3_CTS_B of instance: uart3

图 15.6 设置 GPIO4_IO14

第4步 设置 GPIO5_IO01、GPIO4_IO14 为输入引脚，读取引脚电平

寄存器地址为：

20A_8000	GPIO data register (GPIO4_DR)	32	R/W	0000_0000h	28.5.1/1358
20A_8004	GPIO direction register (GPIO4_GDIR)	32	R/W	0000_0000h	28.5.2/1359
20A_C000	GPIO data register (GPIO5_DR)	32	R/W	0000_0000h	28.5.1/1358
20A_C004	GPIO direction register (GPIO5_GDIR)	32	R/W	0000_0000h	28.5.2/1359

图 15.7 GPIO_DR 寄存器

设置方向寄存器，把引脚设置为输出引脚：

Address: Base address + 4h offset

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	GDIR																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_GDIR field descriptions

Field	Description
GDIR	GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal. **NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO. 0 **INPUT** — GPIO is configured as input. 1 **OUTPUT** — GPIO is configured as output.

图 15.8 设置方向寄存器

读取引脚状态寄存器，得到引脚电平：

Address: Base address + 8h offset

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	PSR																															
W																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_PSR field descriptions

Field	Description
PSR	GPIO pad status bits (status bits). Reading GPIO_PSR returns the state of the corresponding input signal. Settings: **NOTE:** The IOMUXC must be configured to GPIO mode for GPIO_PSR to reflect the state of the corresponding signal.

图 15.9 GPIO 状态寄存器

15.2.3 编程

1 程序框架

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\04_button_drv\02_button_drv_for_boards\05_button_drv_for_100ask_imx6ull
```

2 硬件相关的代码

主要看 board_100ask_imx6ull.c。涉及的寄存器挺多，一个一个去执行 ioremap 效率太低。先定义结构体，然后对结构体指针进行 ioremap。

对于 IOMUXC，可以如下定义：

```
struct iomux {
    volatile unsigned int unnames[23];
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO01;
```

```
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO02;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO04;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO05;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO06;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO07;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO08;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO09;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_CTS_B;
};
```

对于 GPIO，可以如下定义：

```
struct imx6ull_gpio {
    volatile unsigned int dr;
    volatile unsigned int gdir;
    volatile unsigned int psr;
    volatile unsigned int icr1;
    volatile unsigned int icr2;
    volatile unsigned int imr;
    volatile unsigned int isr;
    volatile unsigned int edge_sel;
};
struct imx6ull_gpio gpio4 = ioremap(0x020A8000, sizeof(struct imx6ull_gpio));
struct imx6ull_gpio gpio5 = ioremap(0x20AC000, sizeof(struct imx6ull_gpio));
```

看一个驱动程序，先看它的入口函数，代码如下。

```
119 static struct button_operations my_buttons_ops = {
120     .count = 2,
121     .init = board_imx6ull_button_init,
122     .read = board_imx6ull_button_read,
123 };
124
125 int board_imx6ull_button_drv_init(void)
126 {
127     register_button_operations(&my_buttons_ops);
128     return 0;
129 }
```

第 127 行向上层驱动注册一个 button_operations 结构体，该结构体在第 119～123 行定义。

button_operations 结构体中有 init 函数指针，它指向 board_imx6ull_button_init 函数，在里面将会初始化 LED 引脚：使能、设置为 GPIO 模式、设置为输出引脚。代码如下。

值得关注的是第 71～78 行，对于寄存器要先使用 ioremap 得到它的虚拟地址，以后使用虚拟地址访问寄存器。

```
50 /* enable GPIO4 */
 51 static volatile unsigned int *CCM_CCGR3;
 52
 53 /* enable GPIO5 */
 54 static volatile unsigned int *CCM_CCGR1;
 55
 56 /* set GPIO5_IO03 as GPIO */
 57 static volatile unsigned int *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER1;
```

```
 58
 59 /* set GPIO4_IO14 as GPIO */
 60 static volatile unsigned int *IOMUXC_SW_MUX_CTL_PAD_NAND_CE1_B;
 61
 62 static struct iomux *iomux;
 63
 64 static struct imx6ull_gpio *gpio4;
 65 static struct imx6ull_gpio *gpio5;
 66
 67 static void board_imx6ull_button_init (int which) /* 初始化 button, which-哪个 butt
on */
 68 {
 69     if (!CCM_CCGR1)
 70     {
 71         CCM_CCGR1 = ioremap(0x20C406C, 4);
 72         CCM_CCGR3 = ioremap(0x20C4074, 4);
 73         IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER1 = ioremap(0x229000C, 4);
 74                 IOMUXC_SW_MUX_CTL_PAD_NAND_CE1_B        = ioremap(0x20E01B0, 4);
 75
 76         iomux = ioremap(0x20e0000, sizeof(struct iomux));
 77         gpio4 = ioremap(0x020A8000, sizeof(struct imx6ull_gpio));
 78         gpio5 = ioremap(0x20AC000, sizeof(struct imx6ull_gpio));
 79     }
 80
 81     if (which == 0)
 82     {
 83         /* 1. enable GPIO5
 84          * CG15, b[31:30] = 0b11
 85          */
 86         *CCM_CCGR1 |= (3<<30);
 87
 88         /* 2. set GPIO5_IO01 as GPIO
 89          * MUX_MODE, b[3:0] = 0b101
 90          */
 91         *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER1 = 5;
 92
 93         /* 3. set GPIO5_IO01 as input
 94          * GPIO5 GDIR, b[1] = 0b0
 95          */
 96         gpio5->gdir &= ~(1<<1);
 97     }
 98     else if(which == 1)
 99     {
100         /* 1. enable GPIO4
101          * CG6, b[13:12] = 0b11
102          */
103         *CCM_CCGR3 |= (3<<12);
104
105         /* 2. set GPIO4_IO14 as GPIO
106          * MUX_MODE, b[3:0] = 0b101
107          */
108         IOMUXC_SW_MUX_CTL_PAD_NAND_CE1_B = 5;
109
110         /* 3. set GPIO4_IO14 as input
111          * GPIO4 GDIR, b[14] = 0b0
112          */
```

```
113         gpio4->gdir &= ~(1<<14);
114     }
115
116 }
```

button_operations 结构体中还有有 read 函数指针，它指向 board_imx6ull_button_read 函数，在里面将会读取并返回按键引脚的电平。代码如下。

```
118 static int board_imx6ull_button_read (int which) /* 读 button, which-哪个 */
119 {
120     //printk("%s %s line %d, button %d, 0x%x\n", __FILE__, __FUNCTION__, __LINE_
_, which, *GPIO1_DATAIN);
121     if (which == 0)
122         return (gpio5->psr & (1<<1)) ? 1 : 0;
123     else
124         return (gpio4->psr & (1<<14)) ? 1 : 0;
125 }
```

15.2.4 测试

先启动 IMX6ULL 挂载 NFS 文件系统。

安装驱动程序之后执行测试程序，观察它的返回值(执行测试程序的同时操作按键)：

```
[root@100ask:~]# insmod button_drv.ko
[root@100ask:~]# insmod board_drv.ko
[root@100ask:~]# insmod board_100ask_imx6ull.ko
[root@100ask:~]#./button_test  /dev/100ask_button0
[root@100ask:~]#./button_test  /dev/100ask_button1
```

15.2.5 课后作业

修改 button_test.c，使用按键来点灯

第16章 GPIO 和 Pinctrl 子系统的使用

参考文档：

- 内核 Documentation\devicetree\bindings\Pinctrl\ 目录下：

```
Pinctrl-bindings.txt
```

- 内核 Documentation\gpio 目录下：

```
Pinctrl-bindings.txt
```

- 内核 Documentation\devicetree\bindings\gpio 目录下：

```
gpio.txt
```

注意：本章的重点在于“使用”，深入讲解放在“驱动大全”的视频里。

前面的视频，我们使用直接操作寄存器的方法编写驱动。这只是为了让大家掌握驱动程序的本质，在实际开发过程中我们可不这样做，太低效了！如果驱动开发都是这样去查找寄存器，那我们就变成“寄存器工程师”了，即使是做单片机的都不执着于裸写寄存器了。

Linux 下针对引脚有 2 个重要的子系统：GPIO、Pinctrl。

16.1 Pinctrl 子系统重要概念

16.1.1 引入

无论是哪种芯片，都有类似图 16.1 的结构：

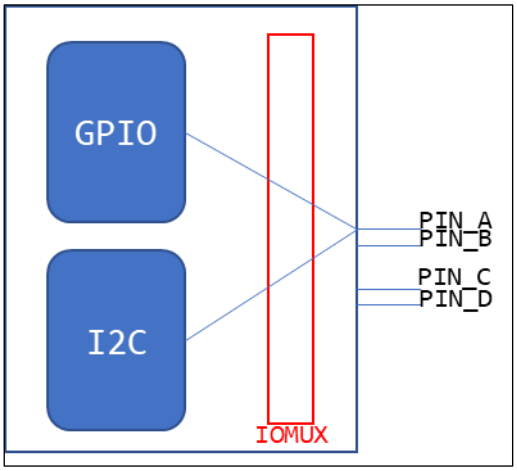


图 16.1 GPIO 与外设连接示意图

- 要想让 pinA、B 用于 GPIO，需要设置 IOMUX 让它们连接到 GPIO 模块；
- 要想让 pinA、B 用于 I2C，需要设置 IOMUX 让它们连接到 I2C 模块。

所以 GPIO、I2C 应该是并列的关系，它们能够使用之前，需要设置 IOMUX。有时候并不仅仅是设置 IOMUX，还要配置引脚，比如上拉、下拉、开漏等等。

现在的芯片动辄几百个引脚，在使用到 GPIO 功能时，让你一个引脚一个引脚去找对应的寄存器，这要疯掉。术业有专攻，这些累活就让芯片厂家做吧——他们是 BSP 工程师。我们在他们的基础上开发，我们是驱动工程师。开玩笑的，BSP 工程师是更懂他自家的芯片，但是如果驱动工程师看不懂他们的代码，那你

的进步也有限啊。

所以，要把引脚的复用、配置抽出来，做成 Pinctrl 子系统，给 GPIO、I2C 等模块使用。

BSP 工程师要做什么？看图 16.2：

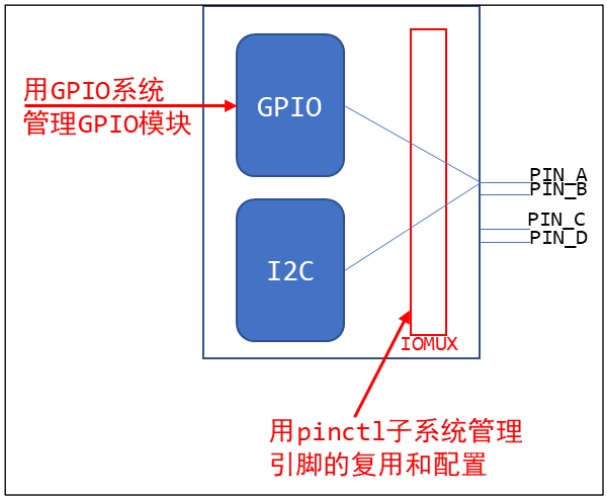


图 16.2 工程师的任务

等 BSP 工程师在 GPIO 子系统、Pinctrl 子系统中把自家芯片的支持加进去后，我们就可以非常方便地使用这些引脚了：点灯简直太简单了。

等等，GPIO 模块在图中跟 I2C 不是并列的吗？干嘛在讲 Pinctrl 时还把 GPIO 子系统拉进来？

大多数的芯片，没有单独的 IOMUX 模块，引脚的复用、配置等等，就是在 GPIO 模块内部实现的。

在硬件上 GPIO 和 Pinctrl 是如此密切相关，在软件上它们的关系也非常密切。

所以这 2 个子系统我们一起讲解。

16.1.2 重要概念

从设备树开始学习 Pintrl 会比较容易。

主要参考文档是：内核 Documentation\devicetree\bindings\pinctrl\pinctrl-bindings.txt

这会涉及 2 个对象：pin controller、client device。

前者提供服务：可以用它来复用引脚、配置引脚。

后者使用服务：声明自己要使用哪些引脚的哪些功能，怎么配置它们。

① pin controller：

在芯片手册里你找不到 pin controller，它是一个软件上的概念，你可以认为它对应 IOMUX——用来复用引脚，还可以配置引脚(比如上下拉电阻等)。

注意，pin controller 和 GPIO Controller 不是一回事，前者控制的引脚可用于 GPIO 功能、I2C 功能；后者只是把引脚配置为输入、输出等简单的功能。即先用 pin controller 把引脚配置为 GPIO，再用 GPIO Controler 把引

脚配置为输入或输出。

② client device

“客户设备”，谁的客户？Pinctrl 系统的客户，那就是使用 Pinctrl 系统的设备，使用引脚的设备。它在设备树里会被定义为一个节点，在节点里声明要用哪些引脚。

图 16.3 就可以把几个重要概念理清楚：

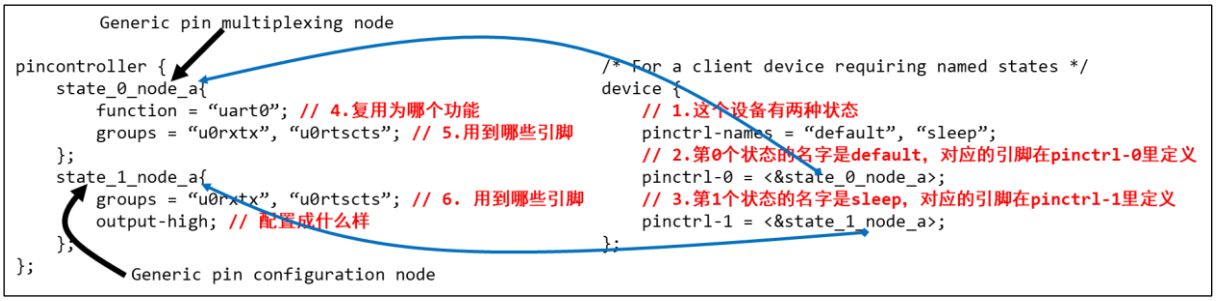


图 16.3 复用节点与配置节点

上图中，左边是 pin controller 节点，右边是 client device 节点：

a) pin state：

对于一个“client device”来说，比如对于一个 UART 设备，它有多个“状态”：default、sleep 等，那对应的引脚也有这些状态。

怎么理解？

比如默认状态下，UART 设备是工作的，那么所用的引脚就要复用为 UART 功能。

在休眠状态下，为了省电，可以把这些引脚复用为 GPIO 功能；或者直接把它们配置输出高电平。

上图中，pinctrl-names 里定义了 2 种状态：default、sleep。

- 第 0 种状态用到的引脚在 pinctrl-0 中定义，它是 state_0_node_a，位于 pincontroller 节点中。
- 第 1 种状态用到的引脚在 pinctrl-1 中定义，它是 state_1_node_a，位于 pincontroller 节点中。

当这个设备处于 default 状态时，pinctrl 子系统会自动根据上述信息把所用引脚复用为 uart0 功能。

当这这个设备处于 sleep 状态时，pinctrl 子系统会自动根据上述信息把所用引脚配置为高电平。

b) groups 和 function：

一个设备会用到一个或多个引脚，这些引脚就可以归为一组(group)；

这些引脚可以复用为某个功能：function。

当然：一个设备可以用到多组引脚，比如 A1、A2 两组引脚，A1 组复用为 F1 功能，A2 组复用为 F2 功能。

c) Generic pin multiplexing node 和 Generic pin configuration node

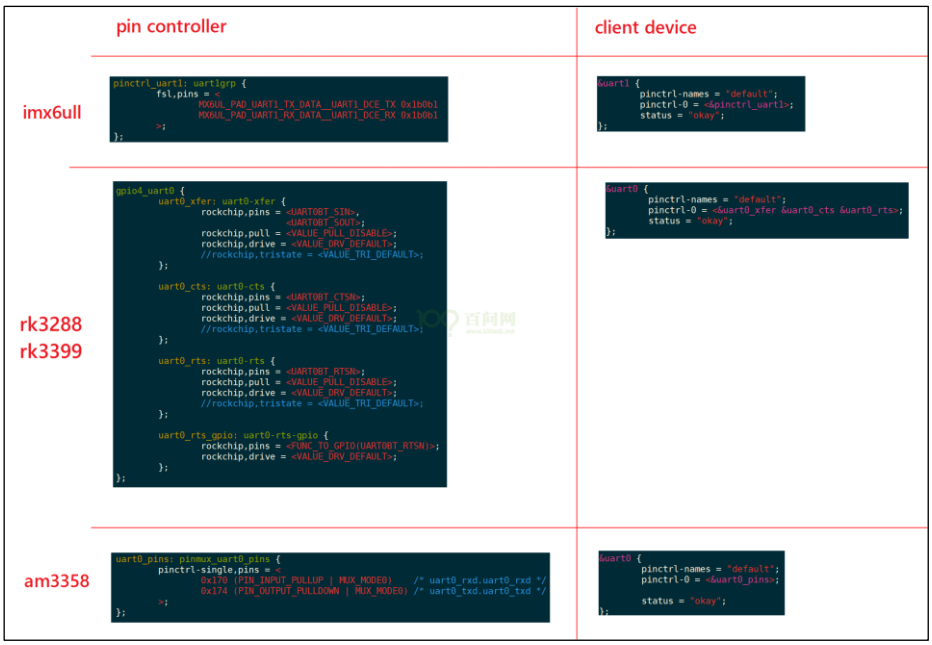
在上图左边的 pin controller 节点中，有子节点或孙节点，它们是给

client device 使用的。

- 可以用来描述复用信息：哪组(group)引脚复用为哪个功能(function)；
- 可以用来描述配置信息：哪组(group)引脚配置为哪个设置功能(setting)，比如上拉、下拉等。

注意：pin controller 节点的格式，没有统一的标准!!!! 每家芯片都不一样。甚至上面的 group、function 关键字也不一定有，但是概念是有的。

16.1.3 示例



16.1.4 代码中怎么引用 pinctrl

这是透明的，我们的驱动基本不用管。当设备切换状态时，对应的 pinctrl 就会被调用。

比如在 platform_device 和 platform_driver 的枚举过程中，流程如下：

```
really_probe:
    /* If using pinctrl, bind pins now before probing */
    ret = pinctrl_bind_pins(dev);                    // 1. 引脚被设置为某个状态：根本不用我们自己去调用代码
            dev->pins->default_state = pinctrl_lookup_state(dev->pins->p,
                                PINCTRL_STATE_DEFAULT);  /* 获得"default"状态的pinctrl */
            dev->pins->init_state = pinctrl_lookup_state(dev->pins->p,
                                PINCTRL_STATE_INIT);     /* 获得"init"状态的pinctrl */

            ret = pinctrl_select_state(dev->pins->p, dev->pins->init_state);    /* 优先设置"init"状态的引脚 */
            ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state); /* 如果没有init状态, 则设置"default"状态的引脚 */

    ......
    ret = drv->probe(dev); // 2. 调用到我们的代码
```

当系统休眠时，也会去设置该设备 sleep 状态对应的引脚，不需要我们自己去调用代码。非要自己调用，也有函数：

```
devm_pinctrl_get_select_default(struct device *dev);      // 使用"default"状态的引脚
pinctrl_get_select(struct device *dev, const char *name); // 根据 name 选择某种状态的引脚
pinctrl_put(struct pinctrl *p);   // 不再使用，退出时调用
```

16.2 GPIO 子系统重要概念

16.2.1 引入

要操作 GPIO 引脚，先把所用引脚配置为 GPIO 功能，这通过 Pinctrl 子系统来实现。

然后就可以根据设置引脚方向(输入还是输出)、读值——获得电平状态，写值——输出高低电平。

以前我们通过寄存器来操作 GPIO 引脚，即使 LED 驱动程序，对于不同的板子它的代码也完全不同。

当 BSP 工程师实现了 GPIO 子系统后，我们就可以：

- 在设备树里指定 GPIO 引脚
- 在驱动代码中：使用 GPIO 子系统的标准函数获得 GPIO、设置 GPIO 方向、读取/设置 GPIO 值。

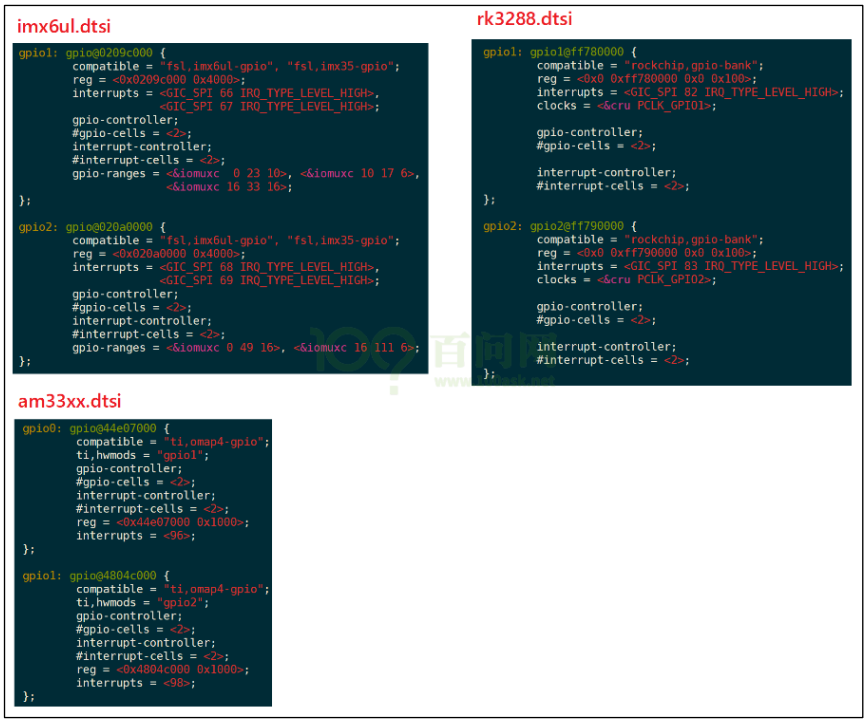
这样的驱动代码，将是单板无关的。

16.2.2 在设备树中指定引脚

在几乎所有 ARM 芯片中，GPIO 都分为几组，每组中有若干个引脚。所以在使用 GPIO 子系统之前，就要先确定：它是哪组的？组里的哪一个？

在设备树中，“GPIO 组”就是一个 GPIO Controller，这通常都由芯片厂家设置好。我们要做的是找到它名字，比如“gpio1”，然后指定要用它里面的哪个引脚，比如<&gpio1 0>。

有代码更直观，下图是一些芯片的 GPIO 控制器节点，它们一般都是厂家定义好，在 xxx.dtsi 文件中：



我们暂时只需要关心里面的这 2 个属性：

```
gpio-controller;
#gpio-cells = <2>;
```

- “gpio-controller”表示这个节点是一个 GPIO Controller，它下面有很多引脚。
- “#gpio-cells = <2>”表示这个控制器下每一个引脚要用 2 个 32 位的数(cell)来描述。

为什么要用 2 个数？其实使用多个 cell 来描述一个引脚，这是 GPIO Controller 自己决定的。比如可以用其中一个 cell 来表示那是哪一个引脚，用另一个 cell 来表示它是高电平有效还是低电平有效，甚至还可以用更多的 cell 来示其他特性。

普遍的用法是，用第 1 个 cell 来表示哪一个引脚，用第 2 个 cell 来表示有效电平：

```
GPIO_ACTIVE_HIGH :  高电平有效
GPIO_ACTIVE_LOW  :  低电平有效
```

定义 GPIO Controller 是芯片厂家的事，我们怎么引用某个引脚呢？在自己的设备节点中使用属性"[<name>-]gpios"，示例如下：

100ask_imx6ull-14x14.dts

```
led0: cpu {
        label = "cpu";
        gpios = <&gpio5 3 GPIO_ACTIVE_LOW>;
        default-state = "on";
        linux,default-trigger = "heartbeat";
};
gt9xx@5d {
        compatible = "goodix,gt9xx";
        reg = <0x5d>;
        status = "okay";
        interrupt-parent = <&gpio1>;
        interrupts = <5 IRQ_TYPE_EDGE_FALLING>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc_reset &pinctrl_touchscreen_int>;

        reset-gpios = <&gpio5 2 GPIO_ACTIVE_LOW>;
        irq-gpios   = <&gpio1 5 IRQ_TYPE_EDGE_FALLING>;
```

上图中，可以使用 gpios 属性，也可以使用 name-gpios 属性。

16.2.3 在驱动代码中调用 GPIO 子系统

在设备树中指定了 GPIO 引脚，在驱动代码中如何使用？

也就是 GPIO 子系统的接口函数是什么？

GPIO 子系统有两套接口：基于描述符的(descriptor-based)、老的(legacy)。前者的函数都有前缀“gpiod_”，它使用 gpio_desc 结构体来表示一个引脚；后者的函数都有前缀“gpio_”，它使用一个整数来表示一个引脚。

要操作一个引脚，首先要 get 引脚，然后设置方向，读值、写值。

驱动程序中要包含头文件，

```
#include <linux/gpio/consumer.h>   // descriptor-based
```

或

```
#include <linux/gpio.h>            // legacy
```

下表列出常用的函数：

表 16-1 常用函数

descriptor-based	legacy
获得 GPIO	
gpiod_get	gpio_request
gpiod_get_index	
gpiod_get_array	gpio_request_array
devm_gpiod_get	
devm_gpiod_get_index	
devm_gpiod_get_array	
设置方向	
gpiod_direction_input	gpio_direction_input
gpiod_direction_output	gpio_direction_output
读值、写值	
gpiod_get_value	gpio_get_value
gpiod_set_value	gpio_set_value
释放 GPIO	
gpio_free	gpio_free
gpiod_put	gpio_free_array
gpiod_put_array	
devm_gpiod_put	
devm_gpiod_put_array	

有前缀“devm_”的含义是“设备资源管理”(Managed Device Resource)，这是一种自动释放资源的机制。它的思想是“资源是属于设备的，设备不存在时资源就可以自动释放”。

比如在 Linux 开发过程中，先申请了 GPIO，再申请内存；如果内存申请失败，那么在返回之前就需要先释放 GPIO 资源。如果使用 devm 的相关函数，在内存申请失败时可以直接返回：设备的销毁函数会自动地释放已经申请了的 GPIO 资源。

建议使用“devm_”版本的相关函数。

假设备在设备树中有如下节点：

```
foo_device {
    compatible = "acme,foo";
    ...
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
            <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
            <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */
```

```
        power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
    };
```

那么可以使用下面的函数获得引脚：

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);
power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

注意：gpiod_set_value 设置的值是“逻辑值”，不一定等于物理值。

什么意思？

Function (example)	active-low property	physical line
gpiod_set_raw_value(desc, 0);	don't care	low
gpiod_set_raw_value(desc, 1);	don't care	high
gpiod_set_value(desc, 0);	default (active-high)	low
gpiod_set_value(desc, 1);	default (active-high)	high
gpiod_set_value(desc, 0);	active-low	high
gpiod_set_value(desc, 1);	active-low	low

如果设备树里引脚指定为GPIO_ACTIVE_LOW
那么gpiod_set_value的逻辑值跟引脚的物理值相反

旧的“gpio_”函数没办法根据设备树信息获得引脚，它需要先知道引脚号。

引脚号怎么确定？

在 GPIO 子系统中，每注册一个 GPIO Controller 时会确定它的“base number”，那么这个控制器里的第 n 号引脚的号码就是：base number + n。

但是如果硬件有变化、设备树有变化，这个 base number 并不能保证是固定的，应该查看 sysfs 来确定 base number。

16.2.4 sysfs 中的访问方法

在 sysfs 中访问 GPIO，实际上用的就是引脚号，老的方法。

- 先确定某个 GPIO Controller 的基准引脚号(base number)，再计算出某个引脚的号码。

方法如下：

① 先在开发板的/sys/class/gpio 目录下，找到各个 gpiochipXXX 目录：

```
[root@imx6ull:/sys/class/gpio]# ls
export       gpio30       gpiochip128  gpiochip504  gpiochip96
gpio133      gpiochip0    gpiochip32   gpiochip64   unexport
[root@imx6ull:/sys/class/gpio]#
```

② 然后进入某个 gpiochip 目录，查看文件 label 的内容

③ 根据 label 的内容对比设备树

label 内容来自设备树，比如它的寄存器基地址。用来跟设备树(dtsi 文件)比较，就可以知道这对应哪一个 GPIO Controller。

下图是在 100asK_imx6ull 上运行的结果，通过对比设备树可知 gpiochip96 对应 gpio4：

```
[root@imx6ull:/sys/class/gpio]# cd gpiochip96/
[root@imx6ull:/sys/class/gpio/gpiochip96]# ls
base       device     label      ngpio      power      subsystem  uevent
[root@imx6ull:/sys/class/gpio/gpiochip96]# cat label
20a8000.gpio
```

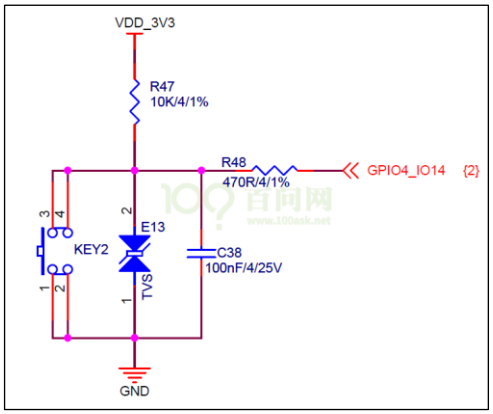
imx6ull.dtsi

```
gpio4: gpio@020a8000 {
        compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
        reg = <0x020a8000 0x4000>;
```

所以 gpio4 这组引脚的基准引脚号就是 96，这也可以“cat base”来再次确认。

- 基于 sysfs 操作引脚：

以 100ask_imx6ull 为例，它有一个按键，原理图如下：



那么 GPIO4_14 的号码是 96+14=110，可以如下操作读取按键值：

```
[root@100ask:~]# echo  110 > /sys/class/gpio/export
[root@100ask:~]# echo in > /sys/class/gpio/gpio110/direction
[root@100ask:~]# cat /sys/class/gpio/gpio110/value
[root@100ask:~]# echo  110 > /sys/class/gpio/unexport
```

注意：如果驱动程序已经使用了该引脚，那么将会 export 失败，会提示下面的错误：

```
-sh: echo: write error: Device or resource busy
```

对于输出引脚，假设引脚号为 N，可以用下面的方法设置它的值为 1：

```
[root@100ask:~]# echo  N > /sys/class/gpio/export
[root@100ask:~]# echo out > /sys/class/gpio/gpioN/direction
[root@100ask:~]# echo 1 > /sys/class/gpio/gpioN/value
[root@100ask:~]# echo  N > /sys/class/gpio/unexport
```

16.3 基于 GPIO 子系统的 LED 驱动程序

16.3.1 编写思路

GPIO 的地位跟其他模块，比如 I2C、UART 的地方是一样的，要使用某个引脚，需要先把引脚配置为 GPIO 功能，这要使用 Pinctrl 子系统，只需要在设备树里指定就可以。在驱动代码上不需要我们做任何事情。

GPIO 本身需要确定引脚，这也需要在设备树里指定。

设备树节点会被内核转换为 platform_device。

对应的，驱动代码中要注册一个 platform_driver，在 probe 函数中：获得引脚、注册 file_operations。

在 file_operations 中：设置方向、读值/写值。图 16.4 就是一个设备树的例子：

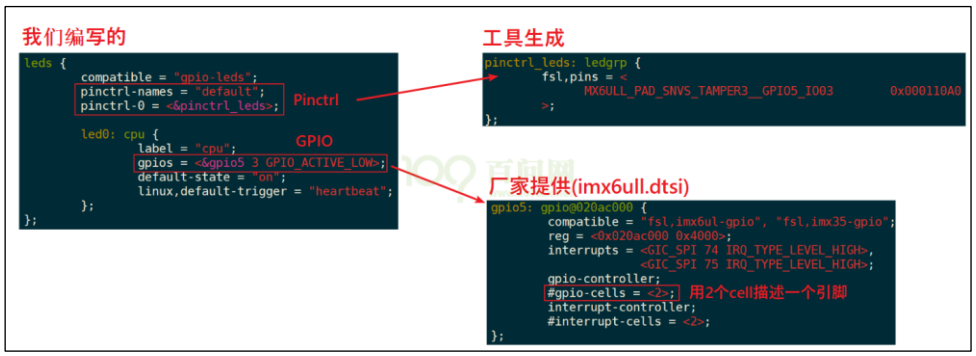


图 16.4 设备树示意

16.3.2 在设备树中添加 Pinctrl 信息

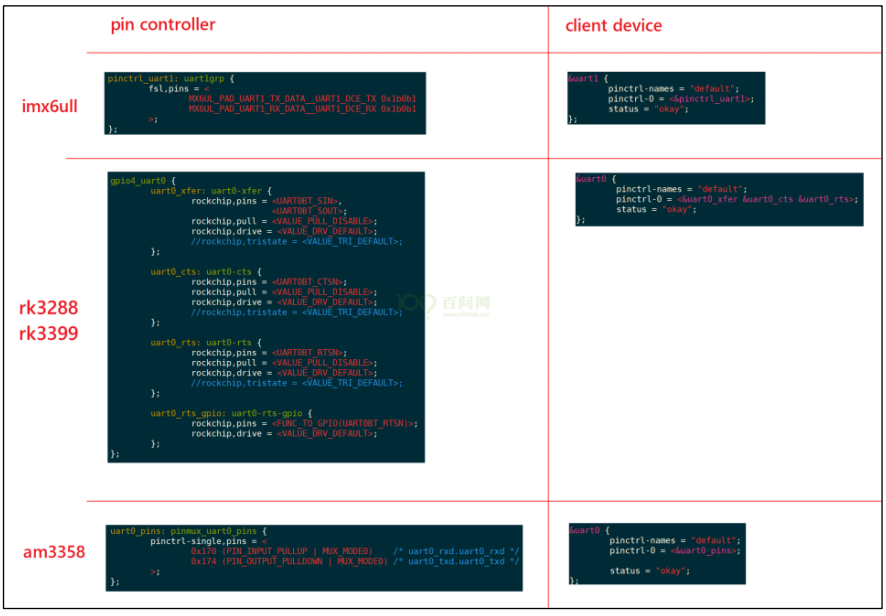
有些芯片提供了设备树生成工具，在 GUI 界面中选择引脚功能和配置信息，就可以自动生成 Pinctrl 子结点。把它复制到你的设备树文件中，再在 client device 结点中引用就可以。

有些芯片只提供文档，那就去阅读文档，一般在内核源码目录 Documentation\devicetree\bindings\pinctrl 下面，保存有该厂家的文档。

如果连文档都没有，那只能参考内核源码中的设备树文件，在内核源码目录 arch/arm/boot/dts 目录下。

最后一步，网络搜索。

Pinctrl 子节点的样式如下：



16.3.3 在设备树中添加 GPIO 信息

先查看电路原理图确定所用引脚，再在设备树中指定：添加”[name]-gpios”属性，指定使用的是哪一个 GPIO Controller 里的哪一个引脚，还有其他 Flag 信息，比如 GPIO_ACTIVE_LOW 等。具体需要多少个 cell 来描述一个引脚，需要查看设备树中这个 GPIO Controller 节点里的“#gpio-cells”属性值，也可以查看内核文档。

示例如下：

100ask_imx6ull-14x14.dts

```
led0: cpu {
        label = "cpu";
        gpios = <&gpio5 3 GPIO_ACTIVE_LOW>;
        default-state = "on";
        linux,default-trigger = "heartbeat";
};
gt9xx@5d {
        compatible = "goodix,gt9xx";
        reg = <0x5d>;
        status = "okay";
        interrupt-parent = <&gpio1>;
        interrupts = <5 IRQ_TYPE_EDGE_FALLING>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc_reset &pinctrl_touchscreen_int>;

        reset-gpios = <&gpio5 2 GPIO_ACTIVE_LOW>;
        irq-gpios   = <&gpio1 5 IRQ_TYPE_EDGE_FALLING>;
```

16.3.4 编程示例

在实际操作过程中也许会碰到意外的问题，现场演示如何解决。

第1步 定义、注册一个 platform_driver

第2步 在它的 probe 函数里：

a) 根据 platform_device 的设备树信息确定 GPIO：gpiod_get

b) 定义、注册一个 file_operations 结构体

c) 在 file_operarions 中使用 GPIO 子系统的函数操作 GPIO：

```
gpiod_direction_output、gpiod_set_value
```

好处：这些代码对所有的板子都是完全一样的！

使用 GIT 命令载后，源码 leddrv.c 位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\05_gpio_and_pinctrl\01_led
```

摘录重点内容：

- 注册 platform_driver

注意下面第 122 行的"100ask,leddrv"，它会跟设备树中节点的 compatible 对应：

```
121 static const struct of_device_id ask100_leds[] = {
122     { .compatible = "100ask,leddrv" },
123     { },
124 };
```

```
125
126 /* 1. 定义 platform_driver */
127 static struct platform_driver chip_demo_gpio_driver = {
128     .probe      = chip_demo_gpio_probe,
129     .remove     = chip_demo_gpio_remove,
130     .driver     = {
131         .name   = "100ask_led",
132         .of_match_table = ask100_leds,
133     },
134 };
135
136 /* 2. 在入口函数注册 platform_driver */
137 static int __init led_init(void)
138 {
139     int err;
140
141     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
142
143     err = platform_driver_register(&chip_demo_gpio_driver);
144
145     return err;
146 }
```

- 在 probe 函数中获得 GPIO

核心代码是第 87 行,它从该设备(对应设备树中的设备节点)获取名为“led”的引脚。在设备树中，必定有一属性名为“led-gpios”或“led-gpio”。

```
77 /* 4. 从 platform_device 获得 GPIO
78  *    把 file_operations 结构体告诉内核：注册驱动程序
79  */
80 static int chip_demo_gpio_probe(struct platform_device *pdev)
81 {
82     //int err;
83
84     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
85
86     /* 4.1 设备树中定义有: led-gpios=<...>; */
87    led_gpio = gpiod_get(&pdev->dev, "led", 0);
88     if (IS_ERR(led_gpio)) {
89             dev_err(&pdev->dev, "Failed to get GPIO for led\n");
90             return PTR_ERR(led_gpio);
91     }
```

- 注册 file_operations 结构体：这是老套路了：

```
93     /* 4.2 注册 file_operations      */
94     major = register_chrdev(0, "100ask_led", &led_drv);  /* /dev/led */
95
96     led_class = class_create(THIS_MODULE, "100ask_led_class");
97     if (IS_ERR(led_class)) {
98             printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
99             unregister_chrdev(major, "led");
100            gpiod_put(led_gpio);
101            return PTR_ERR(led_class);
102    }
103
104    device_create(led_class, NULL, MKDEV(major, 0), NULL, "100ask_led%d", 0); /*
/dev/100ask_led0 */
```

- 在 open 函数中调用 GPIO 函数设置引脚方向：

```
51 static int led_drv_open (struct inode *node, struct file *file)
52 {
53      //int minor = iminor(node);
54
55      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
56      /* 根据次设备号初始化 LED */
57      gpiod_direction_output(led_gpio, 0);
58
59      return 0;
60 }
```

- 在 write 函数中调用 GPIO 函数设置引脚值：

```
34 /* write(fd, &val, 1); */
35 static ssize_t led_drv_write (struct file *file, const char __user *buf, size_t si
ze, loff_t *offset)
36 {
37      int err;
38      char status;
39      //struct inode *inode = file_inode(file);
40      //int minor = iminor(inode);
41
42      printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
43      err = copy_from_user(&status, buf, 1);
44
45      /* 根据次设备号和 status 控制 LED */
46      gpiod_set_value(led_gpio, status);
47
48      return 1;
49 }
```

- 释放 GPIO：

```
gpiod_put(led_gpio);
```

16.4 在 100ASK_IMX6ULL 上机实验

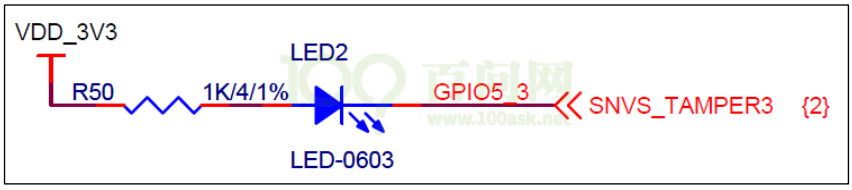
16.4.1 确定引脚并生成设备树节点

NXP 公司对于 IMX6ULL 芯片，有设备树生成工具。我们也把它上传到 GIT 去了，使用 GIT 命令载后，在这个目录下：

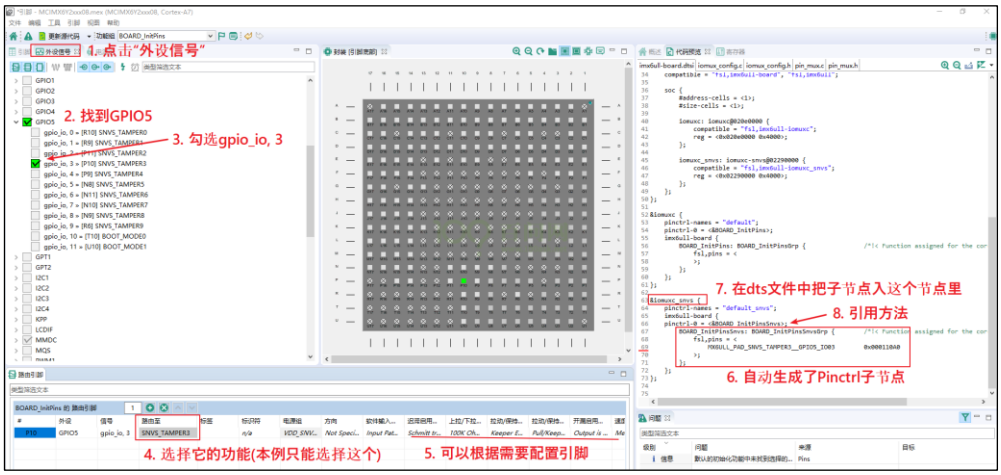
```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\05_gpio_and_pinctrl\tools\imx\
```

安装“Pins_Tool_for_i.MX_Processors_v6_x64.exe”后运行，打开 IMX6ULL 的配置文件“MCIMX6Y2xxx08.mex”，就可以在 GUI 界面中选择引脚，配置它的功能，这就可以自动生成 Pinctrl 的子节点信息。

100ASK_IMX6ULL 使用的 LED 原理图如下，可知引脚是 GPIO5_3：



在设备树工具中，如下图操作：



把自动生成的设备树信息，放到内核源码 `arch/arm/boot/dts/100ask_imx6ull-14x14.dts` 中，代码如下：

1 Pinctrl 信息：

```
&iomuxc_snvs {
……
        myled_for_gpio_subsys: myled_for_gpio_subsys{
            fsl,pins = <
                MX6ULL_PAD_SNVS_TAMPER3__GPIO5_IO03        0x000110A0
            >;
        };
```

2 设备节点信息(放在根节点下)：

```
        myled {
            compatible = "100ask,leddrv";
            pinctrl-names = "default";
            pinctrl-0 = <&myled_for_gpio_subsys>;
            led-gpios = <&gpio5 3 GPIO_ACTIVE_LOW>;
        };
```

16.4.2 编译程序

编译设备树后，要更新设备树。

编译驱动程序时，“leddrv_未测试的原始版本.c”是有错误信息的，“leddrv.c”是修改过的。

测试方法，在板子上执行命令：

```
[root@100ask:~]# insmod  leddrv.ko
[root@100ask:~]# ls /dev/100ask_led0
[root@100ask:~]# ./ledtest /dev/100ask_led0 on
[root@100ask:~]# ./ledtest /dev/100ask_led0 off
```

第17章 异常与中断的概念及处理流程

17.1 中断的引入

17.1.1 妈妈怎么知道孩子醒了

图 17.1 妈妈和孩子

妈妈怎么知道卧室里小孩醒了？

① 查询方式：时不时进房间看一下
简单，但是累

② 休眠-唤醒：进去房间陪小孩一起睡觉，小孩醒了会吵醒她
不累，但是妈妈干不了活了

③ poll 方式：妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟
要浪费点时间，但是可以继续干活。
妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 异步通知：妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈
妈妈、小孩互不耽误。

后面的 3 种方式，都需要“小孩来中断妈妈”：中断她的睡眠、中断她的工作。实际上，能“中断”妈妈的事情可多了：

① 发生了各种声音
② 可忽略的远处猫叫
③ 快递员按门铃
④ 卧室中小孩哭了

妈妈当前正在看书，被这些事件“中断”后她会怎么做？流程如下：

第1步 先在书中放入书签，合上书

第2步 去处理，对于不同的情况，处理方法不同：

- ◆ 对于门铃：开门取快递
- ◆ 对于哭声：照顾小孩

第3步 回来继续看书

17.1.2 嵌入系统中也有类似的情况

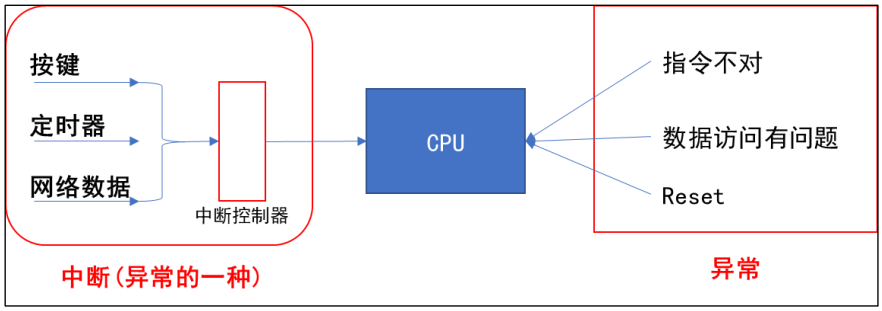


图 17.2 中断与异常

CPU 在运行的过程中，也会被各种“异常”打断。这些“异常”有：

① 指令未定义
② 指令、数据访问有问题
③ SWI(软中断)
④ 快中断
⑤ 中断

中断也属于一种“异常”，导致中断发生的情况有很多，比如：

- 按键
- 定时器
- ADC 转换完成
- UART 发送完数据、收到数据
- 等等

这些众多的“中断源”，汇集到“中断控制器”，由“中断控制器”选择优先级最高的中断并通知 CPU。

17.2 中断的处理流程

arm 对异常(中断)处理过程：

① 初始化：
  a) 设置中断源，让它可以产生中断
  b) 设置中断控制器(可以屏蔽某个中断，优先级)
  c) 设置 CPU 总开关(使能中断)
② 执行其他程序：正常程序
③ 产生中断：比如按下按键--->中断控制器--->CPU
④ CPU 每执行完一条指令都会检查有无中断/异常产生
⑤ CPU 发现有中断/异常产生，开始处理。

对于不同的异常，跳去不同的地址执行程序。

这地址上，只是一条跳转指令，跳去执行某个函数(地址)，这个就是异常向量。③④⑤都是硬件做的。

⑥ 这些函数做什么事情？

软件做的：

a) 保存现场(各种寄存器)
b) 处理异常(中断):分辨中断源，再调用不同的处理函数
c) 恢复现场

17.3 异常向量表

u-boot 或是 Linux 内核，都有类似如下的代码：

```
_start: b  reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq //发生中断时，CPU 跳到这个地址执行该指令 **假设地址为 0x18**
    ldr pc, _fiq
```

这就是异常向量表，每一条指令对应一种异常。

发生复位时，CPU 就去 执行第 1 条指令：b reset。

发生中断时，CPU 就去执行“ldr pc, _irq”这条指令。

这些指令存放的位置是固定的，比如对于 ARM9 芯片中断向量的地址是 0x18。

当发生中断时，CPU 就强制跳去执行 0x18 处的代码。

在向量表里，一般都是放置一条跳转指令，发生该异常时，CPU 就会执行向量表中的跳转指令，去调用更复杂的函数。

当然，向量表的位置并不总是从 0 地址开始，很多芯片可以设置某个 vector base 寄存器，指定向量表在其他位置，比如设置 vector base 为 0x80000000，指定为 DDR 的某个地址。但是表中的各个异常向量的偏移地址，是固定的：复位向量偏移地址是 0，中断是 0x18。

17.4 参考资料

对于 ARM 的中断控制器，述语上称之为 GIC (Generic Interrupt Controller)，到目前已经更新到 v4 版本了。

各个版本的差别可以看这里：

https://developer.arm.com/ip-products/system-ip/system-controllers/interrupt-controllers

简单地说，GIC v3/v4 用于 ARMv8 架构，即 64 位 ARM 芯片。

而 GIC v2 用于 ARMv7 和其他更低的架构。

以后在驱动大全里讲解中断时，我们再深入分析，到时会涉及单核、多核等知识。

第18章 Linux 系统对中断的处理

18.1 进程、线程、中断的核心：栈

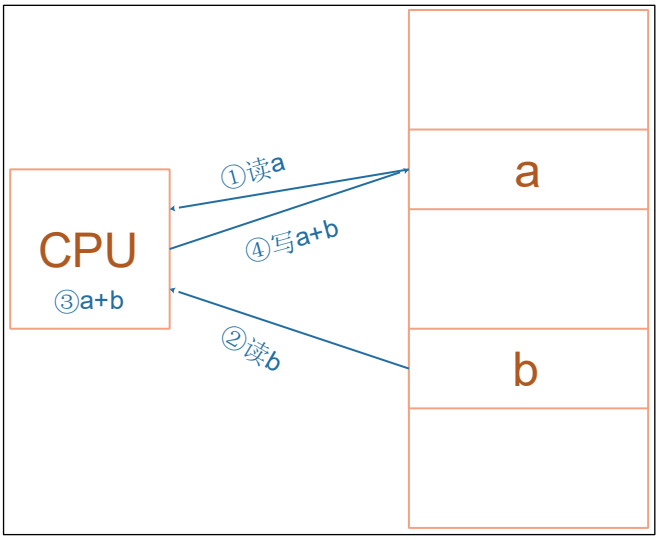
中断中断，中断谁？
中断当前正在运行的进程、线程。
进程、线程是什么？内核如何切换进程、线程、中断？
要理解这些概念，必须理解栈的作用。

18.1.1 18.1.1ARM 处理器程序运行的过程

ARM 芯片属于精简指令集计算机(RISC：Reduced Instruction Set Computing)，它所用的指令比较简单，有如下特点：

① 对内存只有读、写指令
② 对于数据的运算是在 CPU 内部实现
③ 使用 RISC 指令的 CPU 复杂度小一点，易于设计

比如对于 a=a+b 这样的算式，需要经过下面 4 个步骤才可以实现：



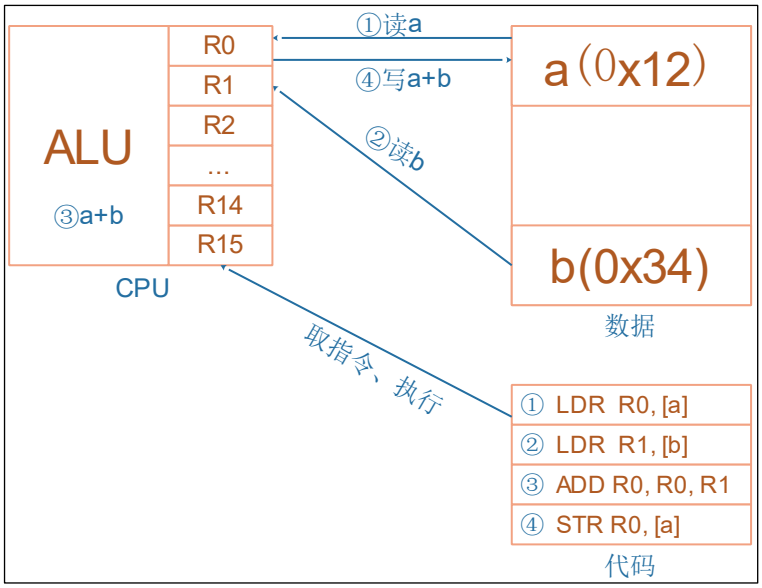
细看这几个步骤，有些疑问：

① 读 a，那么 a 的值读出来后保存在 CPU 里面哪里？
② 读 b，那么 b 的值读出来后保存在 CPU 里面哪里？
③ a+b 的结果又保存在哪里？

我们需要深入 ARM 处理器的内部。简单概括如下，我们先忽略各种 CPU 模式(系统模式、用户模式等等)。

注意：如果想入理解 ARM 处理器架构，应该从裸机开始学习。我们即将写好近 30 个裸机程序的文档，估计还 3 月底发布。

注意：为了加快学习速度，建议先不看裸机。



CPU 运行时，先去取得指令，再执行指令：

① 把内存 a 的值读入 CPU 寄存器 R0

② 把内存 b 的值读入 CPU 寄存器 R1

③ 把 R0、R1 累加，存入 R0

④ 把 R0 的值写入内存 a

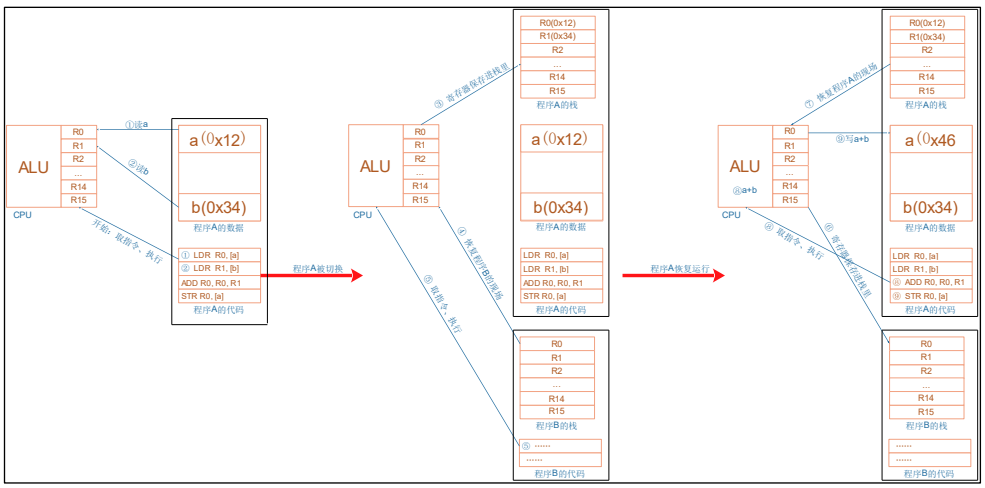
18.1.2 程序被中断时，怎么保存现场

从上图可知，CPU 内部的寄存器很重要，如果要暂停一个程序，中断一个程序，就需要把这些寄存器的值保存下来：这就称为保存现场。

保存在哪里？内存，这块内存就称之为栈。

程序要继续执行，就先从栈中恢复那些 CPU 内部寄存器的值。

这个场景并不局限于中断，下图可以概括程序 A、B 的切换过程，其他情况是类似的：



① 函数调用：

a) 在函数 A 里调用函数 B，实际就是中断函数 A 的执行。

b) 那么需要把函数 A 调用 B 之前瞬间的 CPU 寄存器的值，保存到栈里；

c) 再去执行函数 B；
d) 函数 B 返回之后，就从栈中恢复函数 A 对应的 CPU 寄存器值，继续执行。

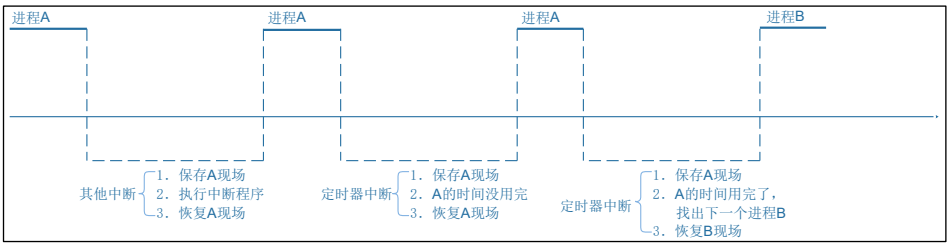
② 中断处理

a) 进程 A 正在执行，这时候发生了中断。
b) CPU 强制跳到中断异常向量地址去执行，
c) 这时就需要保存进程 A 被中断瞬间的 CPU 寄存器值，
d) 可以保存在进程 A 的内核态栈，也可以保存在进程 A 的内核结构体中。
e) 中断处理完毕，要继续运行进程 A 之前，恢复这些值。

③ 进程切换

a) 在所谓的多任务操作系统中，我们以为多个程序是同时运行的。
b) 如果我们能感知微秒、纳秒级的事件，可以发现操作系统时让这些程序依次执行一小段时间，进程 A 的时间用完了，就切换到进程 B。
c) 怎么切换？
d) 切换过程是发生在内核态里的，跟中断的处理类似。
e) 进程 A 的被切换瞬间的 CPU 寄存器值保存在某个地方；
f) 恢复进程 B 之前保存的 CPU 寄存器值，这样就可以运行进程 B 了。

所以，在中断处理的过程中，伴存着进程的保存现场、恢复现场。进程的调度也是使用栈来保存、恢复现场：



18.1.3 进程、线程的概念

假设我们写一个音乐播放器，在播放音乐的同时会根据按键选择下一首歌。把事情简化为 2 件事：发送音频数据、读取按键。那可以这样写程序：

```
int main(int argc, char **argv)
{
    int key;

    while (1)
    {
        key = read_key();

        if (key != -1)
        {
            switch (key)
            {
                case NEXT:
```

```
                    select_next_music(); // 在GUI选中下一首歌
                    break;
            }
        }
        else
        {
            send_music();
        }
    }

    return 0;
}
```

这个程序只有一条主线，读按键、播放音乐都是顺序执行。

无论按键是否被按下，read_key 函数必须马上返回，否则会使得后续的 send_music 受到阻滞导致音乐播放不流畅。

读取按键、播放音乐能否分为两个程序进行？可以，但是开销太大：读按键的程序，要把按键通知播放音乐的程序，进程间通信的效率没那么高。

这时可以用多线程之编程，读取按键是一个线程，播放音乐是另一个线程，它们之间可以通过全局变量传递数据，示意代码如下：

```
int g_key;
void key_thread_fn()
{
    while (1)
    {
        g_key = read_key();
        if (g_key != -1)
        {
            switch (g_key)
            {
                case NEXT:
                    select_next_music(); // 在GUI选中下一首歌
                    break;
            }
        }
    }
}
void music_fn()
{
    while (1)
    {
        if (g_key == STOP)
            stop_music();
        else
        {
           send_music();
        }
    }

}

int main(int argc, char **argv)
{
    int key;
```

```
    create_thread(key_thread_fn);
    create_thread(music_fn);
    while (1)
    {
        sleep(10);
    }
    return 0;
}
```

这样，按键的读取及 GUI 显示、音乐的播放，可以分开来，不必混杂在一起。

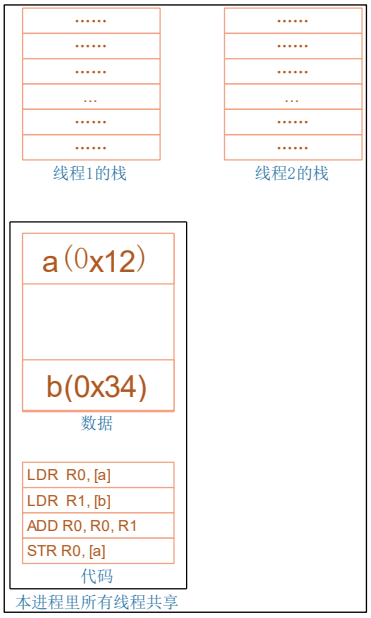
按键线程可以使用阻塞方式读取按键，无按键时是休眠的，这可以节省 CPU 资源。

音乐线程专注于音乐的播放和控制，不用理会按键的具体读取工作。并且这 2 个线程通过全局变量 g_key 传递数据，高效而简单。

在 Linux 中：资源分配的单位是进程，调度的单位是线程。

也就是说，在一个进程里，可能有多个线程，这些线程共用打开的文件句柄、全局变量等等。

而这些线程，之间是互相独立的，“同时运行”，也就是说：每一个线程，都有自己的栈。如下图示：



18.2 18.2 Linux 系统对中断处理的演进

从 2005 年我接触 Linux 到现在 15 年了，Linux 中断系统的变化并不大。比较重要的就是引入了 threaded irq：使用内核线程来处理中断。

Linux 系统中有硬件中断，也有软件中断。对硬件中断的处理有 2 个原则：不能嵌套，越快越好。

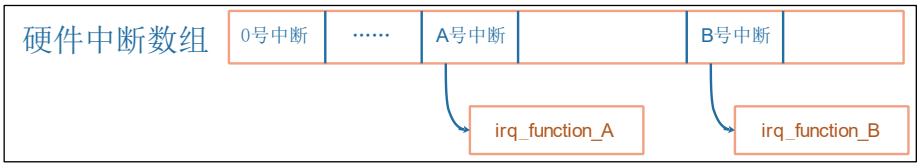
参考资料：

https://blog.csdn.net/myarrow/article/details/9287169

18.2.1 Linux 对中断的扩展：硬件中断、软件中断

Linux 系统把中断的意义扩展了，对于按键中断等硬件产生的中断，称之为“硬件中断”(hard irq)。每个硬件中断都有对应的处理函数，比如按键中断、网卡中断的处理函数肯定不一样。

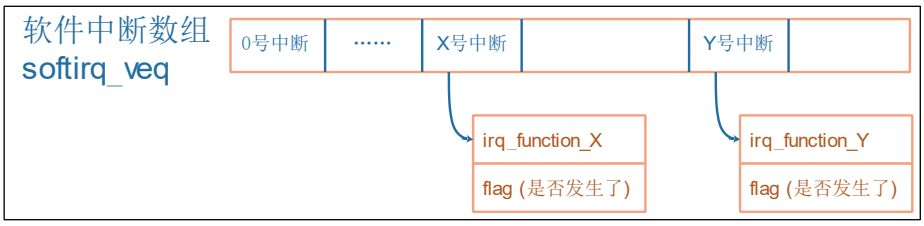
为方便理解，你可以先认为对硬件中断的处理是用数组来实现的，数组里存放的是函数指针：



注意：上图是简化的，Linux 中这个数组复杂多了。

当发生 A 中断时，对应的 irq_function_A 函数被调用。硬件导致该函数被调用。

相对的，还可以人为地制造中断：软件中断(soft irq)，如下图所示：



注意：上图是简化的，Linux 中这个数组复杂多了。

问题来了：

① 软件中断何时生产？

由软件决定，对于 X 号软件中断，只需要把它的 flag 设置为 1 就表示发生了该中断。

② 软件中断何时处理？

软件中断嘛，并不是那么十万火急，有空再处理它好了。

什么时候有空？不能让它一直等吧？

Linux 系统中，各种硬件中断频繁发生，至少定时器中断每 10ms 发生一次，那取个巧？

在处理完硬件中断后，再去处理软件中断？就这么办！

③ 有哪些软件中断？

查内核源码 include/linux/interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the
                numbering. Sigh! */
    RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

    NR_SOFTIRQS
};
```

怎么触发软件中断？最核心的函数是 raise_softirq，简单地理解就是设置 softirq_veq[nr]的标记位：

```
extern void raise_softirq(unsigned int nr);
```

怎么设置软件中断的处理函数：

extern void open_softirq(int nr, void (*action) (struct soft_action*));

```
extern void open_softirq(int nr, void (*action)(struct softirq_action *));
```

后面讲到的中断下半部 tasklet 就是使用软件中断实现的。

18.2.2 中断处理原则 1：不能嵌套

官方资料：中断处理不能嵌套

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58aa3d2d0cc

中断处理函数需要调用 C 函数，这就需要用到栈。

- 中断 A 正在处理的过程中，假设又发生了中断 B，那么在栈里要保存 A 的现场，然后处理 B。
- 在处理 B 的过程中又发生了中断 C，那么在栈里要保存 B 的现场，然后处理 C。

如果中断嵌套突然暴发，那么栈将越来越大，栈终将耗尽。

所以，为了防止这种情况发生，也是为了简单化中断的处理，在 Linux 系统上中断无法嵌套：即当前中断 A 没处理完之前，不会响应另一个中断 B(即使它的优先级更高)。

18.2.3 中断处理原则 2：越快越好

妈妈在家中照顾小孩时，门铃响起，她开门取快递：这就是中断的处理。她取个快递敢花上半天吗？不怕小孩出意外吗？

同理，在 Linux 系统中，中断的处理也是越快越好。

在单芯片系统中，假设中断处理很慢，那应用程序在这段时间内就无法执行：系统显得很迟顿。

在 SMP 系统中，假设中断处理很慢，那么正在处理这个中断的 CPU 上的其他线程也无法执行。

在中断的处理过程中，该 CPU 是不能进行进程调度的，所以中断的处理要越快越好，尽早让其他中断能被处理——进程调度靠定时器中断来实现。

在 Linux 系统中使用中断是挺简单的，为某个中断 irq 注册中断处理函数 handler，可以使用 request_irq 函数：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
```

在 handler 函数中，代码尽可能高效。

但是，处理某个中断要做的事情就是很多，没办法加快。比如对于按键中断，我们需要等待几十毫秒消除机械抖动。难道要在 handler 中等待吗？对于计算机来说，这可是一个段很长的时间。

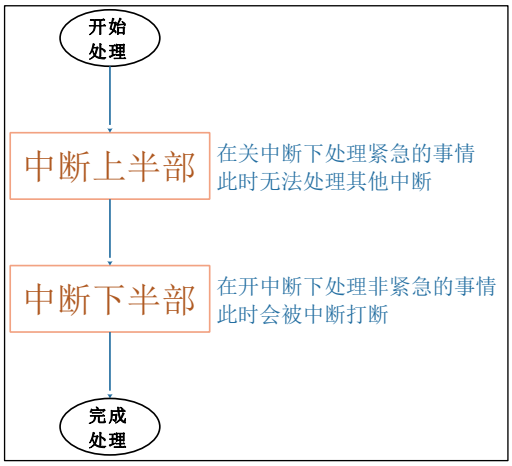
怎么办？

18.2.4 要处理的事情实在太多，拆分为：上半部、下半部

当一个中断要耗费很多时间来处理时，它的坏处是：在这段时间内，其他中断无法被处理。换句话说，在这段时间内，系统是关中断的。

如果某个中断就是要做那么多事，我们能不能把它拆分成两部分：紧急的、不紧急的？

在 handler 函数里只做紧急的事，然后就重新开中断，让系统得以正常运行；那些不紧急的事，以后再处理，处理时是开中断的。



中断下半部的实现有很多种方法，讲 2 种主要的：tasklet(小任务)、work queue(工作队列)。

18.2.5 下半部要做的事情耗时不是太长：tasklet

假设我们把中断分为上半部、下半部。发生中断时，上半部下半部的代码何时、如何被调用？

当下半部比较耗时但是能忍受，并且它的处理比较简单时，可以用 tasklet

来处理下半部。tasklet 是使用软件中断来实现。

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the
                        numbering. Sigh! */
    RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

    NR_SOFTIRQS
};
```

tasklet软件中断，
用来处理中断下半部

写字太多，不如贴代码，代码一目了然：

```
__irq_svc () at arch/arm/kernel/entry-armv.S:219
    gic_handle_irq at drivers/irqchip/irq-gic.c:364
        handle_domain_irq at ./include/linux/irqdesc.h:168
            __handle_domain_irq at kernel/irq/irqdesc.c:627
                irq_enter();
                    __irq_enter()
                        preempt_count_add(HARDIRQ_OFFSET);  // preempt_count++,表示开始处理中断
                generic_handle_irq(irq); // 会调用reqeust_irq注册的中断函数: 上半部
                irq_exit();
                    preempt_count_sub(HARDIRQ_OFFSET);      // preempt_count--, 表示硬件中断处理完毕
                    if (!in_interrupt() && local_softirq_pending()) // in_interrupt: preempt_count > 0
                        invoke_softirq();
                            __do_softirq();
                                __local_bh_disable_ip(_RET_IP_, SOFTIRQ_OFFSET);  // preempt_count++,开始软件中断
                                local_irq_enable();         // 开中断
                                // 执行softirq_vec数组上各类软中断: 下半部
                                local_irq_disable();        // 关中断, 马上就会再开中断
                                __local_bh_enable(SOFTIRQ_OFFSET);              // preempt_count--,完成软件中断
```

进出硬件中断

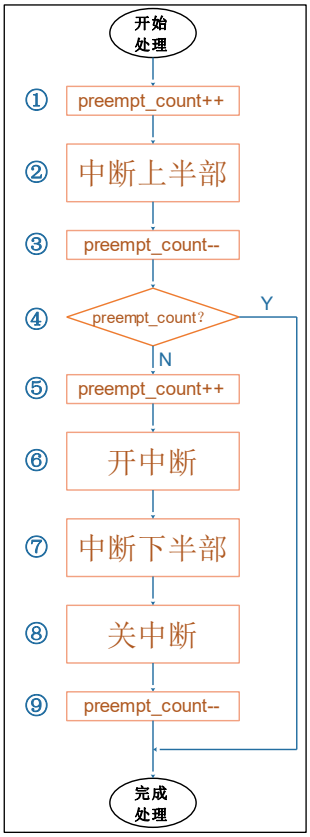
上半部

用preempt_count避免软中断多次执行

处理软中断时开中断

下半部

使用流程图简化一下：



假设硬件中断 A 的上半部函数为 irq_top_half_A，下半部为 irq_bottom_half_A。

使用情景化的分析，才能理解上述代码的精华。

- 硬件中断 A 处理过程中，没有其他中断发生：

  一开始，preempt_count = 0；

上述流程图①～⑨依次执行，上半部、下半部的代码各执行一次。

- 硬件中断 A 处理过程中，又再次发生了中断 A：

一开始，preempt_count = 0；

执行到第⑥时，一开中断后，中断 A 又再次使得 CPU 跳到中断向量表。

注意：这时 preempt_count 等于 1，并且中断下半部的代码并未执行。

CPU 又从①开始再次执行中断 A 的上半部代码：

在第①步 preempt_count 等于 2；

在第③步 preempt_count 等于 1；

在第④步发现 preempt_count 等于 1，所以直接结束当前第 2 次中断的处理；

注意：重点来了，第 2 次中断发生后，打断了第一次中断的第⑦步处理。当第 2 次中断处理完毕，CPU 会继续去执行第⑦步。

可以看到，发生 2 次硬件中断 A 时，它的上半部代码执行了 2 次，但是下半部代码只执行了一次。

所以，同一个中断的上半部、下半部，在执行时是多对一的关系。

- 硬件中断 A 处理过程中，又再次发生了中断 B：

一开始，preempt_count = 0；

执行到第⑥时，一开中断后，中断 B 又再次使得 CPU 跳到中断向量表。

注意：这时 preempt_count 等于 1，并且中断 A 下半部的代码并未执行。

CPU 又从①开始再次执行中断 B 的上半部代码：

在第①步 preempt_count 等于 2；

在第③步 preempt_count 等于 1；

在第④步发现 preempt_count 等于 1，所以直接结束当前第 2 次中断的处理；

注意：重点来了，第 2 次中断发生后，打断了第一次中断 A 的第⑦步处理。当第 2 次中断 B 处理完毕，CPU 会继续去执行第⑦步。

在第⑦步里，它会去执行中断 A 的下半部，也会去执行中断 B 的下半部。

所以，多个中断的下半部，是汇集在一起处理的。

总结：

① 中断的处理可以分为上半部，下半部

② 中断上半部，用来处理紧急的事，它是在关中断的状态下执行的

③ 中断下半部，用来处理耗时的、不那么紧急的事，它是在开中断的状态下执行的

④ 中断下半部执行时，有可能会被多次打断，有可能会再次发生同一个中断

⑤ 中断上半部执行完后，触发中断下半部的处理

⑥ 中断上半部、下半部的执行过程中，不能休眠：中断休眠的话，以后谁来调度进程啊？

18.2.6 下半部要做的事情太多并且很复杂：工作队列

在中断下半部的执行过程中，虽然是开中断的，期间可以处理各类中断。但是毕竟整个中断的处理还没走完，这期间 APP 是无法执行的。

假设下半部要执行 1、2 分钟，在这 1、2 分钟里 APP 都是无法响应的。

这谁受得了？所以，如果中断要做的事情实在太耗时，那就不能用软件中断来做，而应该用内核线程来做：在中断上半部唤醒内核线程。内核线程和 APP 都一样竞争执行，APP 有机会执行，系统不会卡顿。

这个内核线程是系统帮我们创建的，一般是 kworker 线程，内核中有很多这样的线程：

```
book@book-virtual-machine:~$ ps -A |grep kworker
    4 ?        00:00:00 kworker/0:0H
   18 ?        00:00:00 kworker/1:0H
   24 ?        00:00:00 kworker/2:0H
   30 ?        00:00:00 kworker/3:0H
   36 ?        00:00:00 kworker/4:0H
   42 ?        00:00:00 kworker/5:0H
```

kworker 线程要去“工作队列”(work queue)上取出一个一个“工作”(work)，来执行它里面的函数。

那我们怎么使用 work、work queue 呢？

① 创建 work：

你得先写出一个函数，然后用这个函数填充一个 work 结构体。比如：

```
static DECLARE_WORK(aer_recover_work, aer_recover_work_func);
                    work结构体          函数
```

② 要执行这个函数时，把 work 提交给 work queue 就可以了：

```
schedule_work(&aer_recover_work);
```

上述函数会把 work 提供给系统默认的 work queue：system_wq，它是一个队列。

③ 谁来执行 work 中的函数？

不用我们管，schedule_work 函数不仅仅是把 work 放入队列，还会把 kworker 线程唤醒。此线程抢到时间运行时，它就会从队列中取出 work，执行里面的函数。

④ 谁把 work 提交给 work queue？

在中断场景中，可以在中断上半部调用 schedule_work 函数。

总结：

- 很耗时的中断处理，应该放到线程里去
- 可以使用 work、work queue
- 在中断上半部调用 schedule_work 函数，触发 work 的处理
- 既然是在线程中运行，那对应的函数可以休眠。

18.2.7 新技术：threaded irq

使用线程来处理中断，并不是什么新鲜事。使用 work 就可以实现，但是需要定义 work、调用 schedule_work，好麻烦啊。

太懒了太懒了，就这 2 步你们都不愿意做。

好，内核是为懒人服务的，再杀出一个函数：

```
extern int __must_check
request_threaded_irq(unsigned int irq, irq_handler_t handler,
                     irq_handler_t thread_fn,
                     unsigned long flags, const char *name, void *dev);
```

哪个中断？

上半部函数，可以为空

在线程里运行的函数

你可以只提供 thread_fn，系统会为这个函数创建一个内核线程。发生中断时，内核线程就会执行这个函数。

说你懒是开玩笑，内核开发者也不会那么在乎懒人。

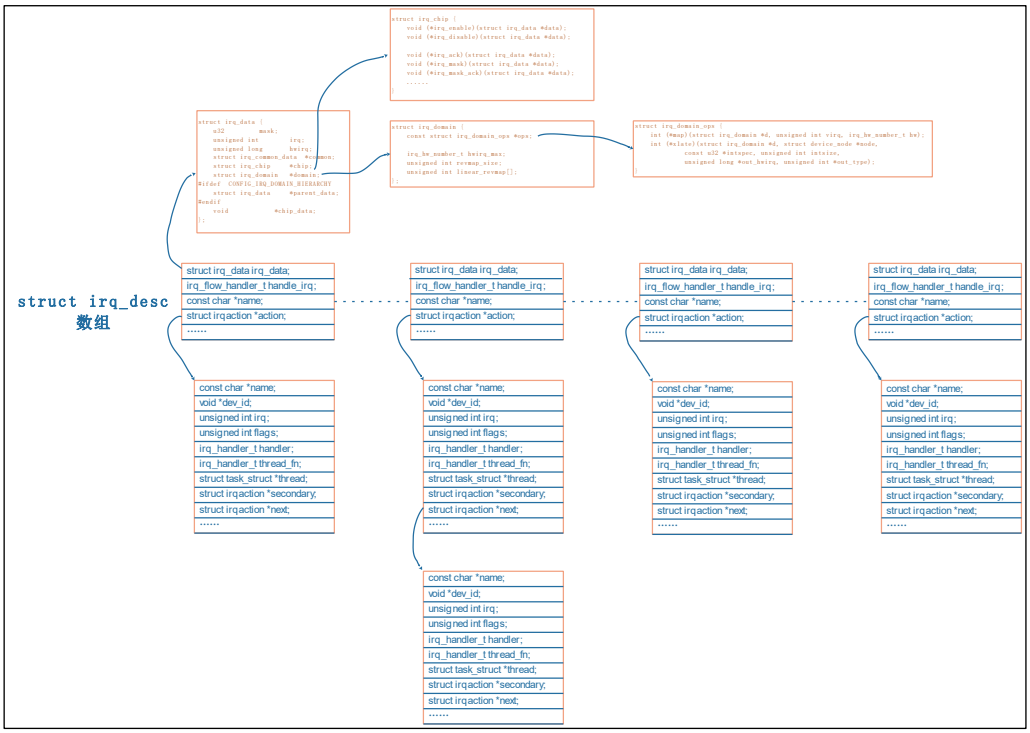
以前用 work 来线程化地处理中断，一个 worker 线程只能由一个 CPU 执行，多个中断的 work 都由同一个 worker 线程来处理，在单 CPU 系统中也只能忍着了。但是在 SMP 系统中，明明有那么多 CPU 空着，你偏偏让多个中断挤在这个 CPU 上？

新技术 threaded irq，为每一个中断都创建一个内核线程；多个中断的内核线程可以分配到多个 CPU 上执行，这提高了效率。

18.3 Linux 中断系统中的重要数据结构

本节内容，可以从 request_irq(include/linux/interrupt.h)函数一路分析得到。

能弄清楚下面这个图，对 Linux 中断系统的掌握也基本到位了。

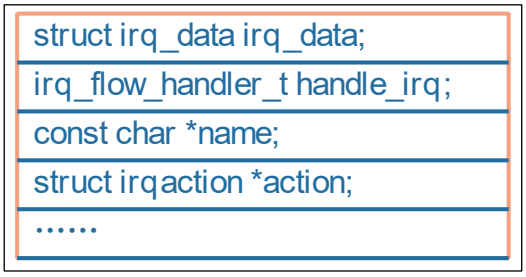


最核心的结构体是 irq_desc，之前为了易于理解，我们说在 Linux 内核中有一个中断数组，对于每一个硬件中断，都有一个数组项，这个数组就是 irq_desc 数组。

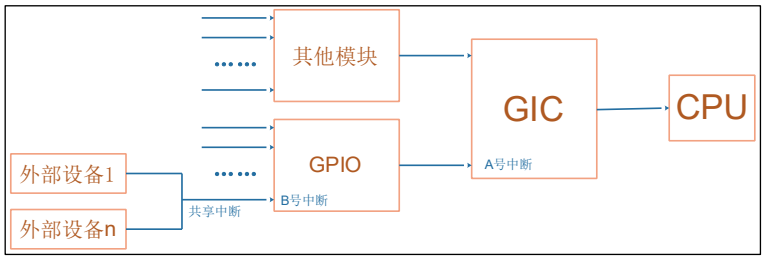
注意：如果内核配置了 CONFIG_SPARSE_IRQ，那么它就会用基数树(radix tree)来代替 irq_desc 数组。SPARSE 的意思是“稀疏”，假设大小为 1000 的数组中只用到 2 个数组项，那不是浪费嘛？所以在中断比较“稀疏”的情况下可以用基数树来代替数组。

18.3.1 irq_desc 数组

irq_desc 结构体在 include/linux/irqdesc.h 中定义，主要内容如下图：



每一个 irq_desc 数组项中都有一个函数：handle_irq，还有一个 action 链表。要理解它们，需要先看中断结构图：



外部设备 1、外部设备 n 共享一个 GPIO 中断 B，多个 GPIO 中断汇聚到 GIC(通用中断控制器)的 A 号中断，GIC 再去中断 CPU。那么软件处理时就是反过来，先读取 GIC 获得中断号 A，再细分出 GPIO 中断 B，最后判断是哪一个外部芯片发生了中断。

所以，中断的处理函数来源有三：

① GIC 的处理函数：

假设 irq_desc[A].handle_irq 是 XXX_gpio_irq_handler(XXX 指厂家)，这个函数需要读取芯片的 GPIO 控制器，细分发生的是哪一个 GPIO 中断(假设是 B)，再去调用 irq_desc[B]. handle_irq。

注意：irq_desc[A].handle_irq 细分出中断后 B，调用对应的 irq_desc[B].handle_irq。

显然中断 A 是 CPU 感受到的顶层的中断，GIC 中断 CPU 时，CPU 读取 GIC 状态得到中断 A。

② 模块的中断处理函数：

比如对于 GPIO 模块向 GIC 发出的中断 B，它的处理函数是

irq_desc[B].handle_irq。

BSP 开发人员会设置对应的处理函数，一般是 handle_level_irq 或 handle_edge_irq，从名字上看是用来处理电平触发的中断、边沿触发的中断。注意：导致 GPIO 中断 B 发生的原因很多，可能是外部设备 1，可能是外部设备 n，可能只是某一个设备，也可能是多个设备。所以 irq_desc[B].handle_irq 会调用某个链表里的函数，这些函数由外部设备提供。这些函数自行判断该中断是否自己产生，若是则处理。

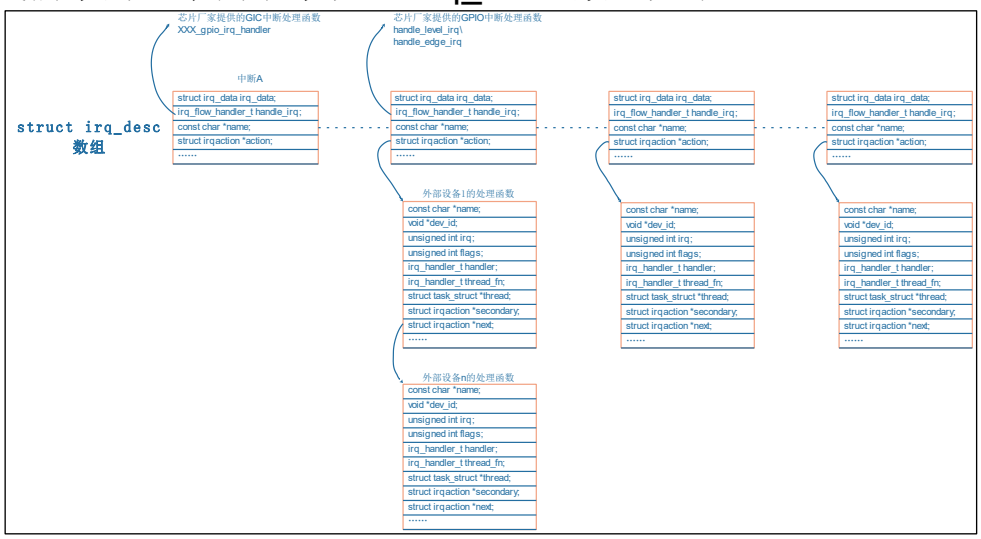
③ 外部设备提供的处理函数：

这里说的“外部设备”可能是芯片，也可能总是简单的按键。它们的处理函数由自己驱动程序提供，这是最熟悉这个设备的“人”：它知道如何判断设备是否发生了中断，如何处理中断。

对于共享中断，比如 GPIO 中断 B，它的中断来源可能有多个，每个中断源对应一个中断处理函数。所以 irq_desc[B]中应该有一个链表，存放着多个中断源的处理函数。

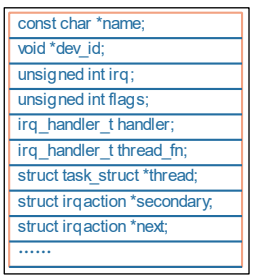
一旦程序确定发生了 GPIO 中断 B，那么就会从链表里把那些函数取出来，一一执行。这个链表就是 action 链表。

对于我们举的这个例子来说，irq_desc 数组如下：



18.3.2 irqaction 结构体

irqaction 结构体在 include/linux/interrupt.h 中定义，主要内容如下图：



当调用 request_irq、request_threaded_irq 注册中断处理函数时，内核就会构造一个 irqaction 结构体。在里面保存 name、dev_id 等，最重要的是 handler、thread_fn、thread。

handler 是中断处理的上半部函数，用来处理紧急的事情。

thread_fn 对应一个内核线程 thread，当 handler 执行完毕，Linux 内核会唤醒对应的内核线程。在内核线程里，会调用 thread_fn 函数。

- 可以提供 handler 而不提供 thread_fn，就退化为一般的 request_irq 函数。
- 可以不提供 handler 只提供 thread_fn，完全由内核线程来处理中断。
- 也可以既提供 handler 也提供 thread_fn，这就是中断上半部、下半部。

里面还有一个名为 sedondary 的 irqaction 结构体，它的作用以后再分析。

在 reqeust_irq 时可以传入 dev_id，为何需要 dev_id？作用有 2：

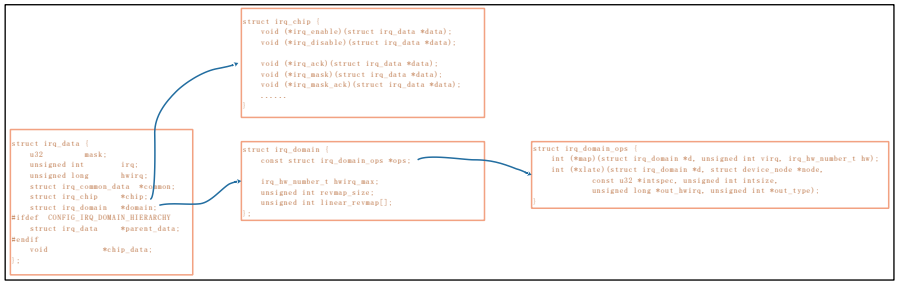
① 中断处理函数执行时，可以使用 dev_id

② 卸载中断时要传入 dev_id，这样才能在 action 链表中根据 dev_id 找到对应项

所以在共享中断中必须提供 dev_id，非共享中断可以不提供。

18.3.3 irq_data 结构体

irq_data 结构体在 include/linux/irq.h 中定义，主要内容如下图：



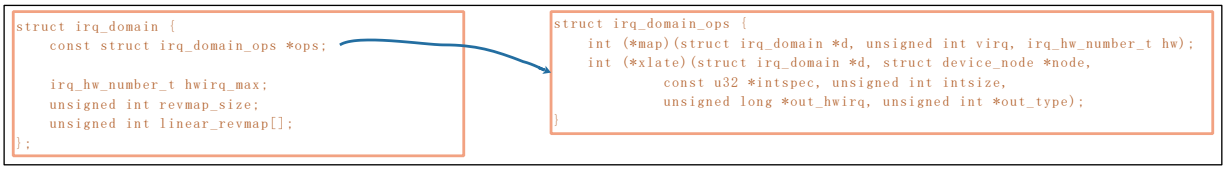
它就是个中转站，里面有 irq_chip 指针 irq_domain 指针，都是指向别的结构体。

比较有意思的是 irq、hwirq，irq 是软件中断号，hwirq 是硬件中断号。比如上面我们举的例子，在 GPIO 中断 B 是软件中断号，可以找到 irq_desc[B] 这个数组项；GPIO 里的第 x 号中断，这就是 hwirq。

谁来建立 irq、hwirq 之间的联系呢？由 irq_domain 来建立。irq_domain 会把本地的 hwirq 映射为全局的 irq，什么意思？比如 GPIO 控制器里有第 1 号中断，UART 模块里也有第 1 号中断，这两个“第 1 号中断”是不一样的，它们属于不同的“域”——irq_domain。

18.3.4 irq_domain 结构体

irq_domain 结构体在 include/linux/irqdomain.h 中定义，主要内容如下图：



当我们后面从设备树讲起，如何在设备树中指定中断，设备树的中断如何被转换为 irq 时，irq_domain 将会起到极大的作为。

这里基于入门的解度简单讲讲，在设备树中你会看到这样的属性：

```
interrupt-parent = <&gpio1>;
interrupts = <5 IRQ_TYPE_EDGE_RISING>;
```

它表示要使用 gpio1 里的第 5 号中断，hwirq 就是 5。

但是我们在驱动中会使用 request_irq(irq, handler)这样的函数来注册中断，irq 是什么？它是软件中断号，它应该从“gpio1 的第 5 号中断”转换得来。

谁把 hwirq 转换为 irq？由 gpio1 的相关数据结构，就是 gpio1 对应的 irq_domain 结构体。

irq_domain 结构体中有一个 irq_domain_ops 结构体，里面有各种操作函数，主要是：

- xlate

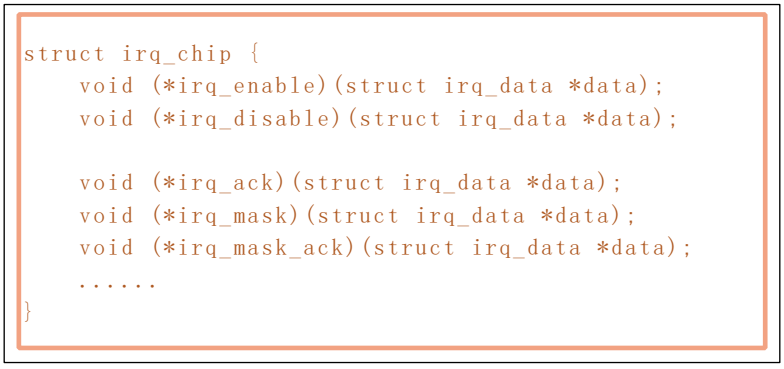
  用来解析设备树的中断属性，提取出 hwirq、type 等信息。

- map

  把 hwirq 转换为 irq。

18.3.5 irq_chip 结构体

irq_chip 结构体在 include/linux/irq.h 中定义，主要内容如下图：



这个结构体跟“chip”即芯片相关，里面各成员的作用在头文件中也列得很清楚，摘录部分如下：

```
* @irq_startup:    start up the interrupt (defaults to ->enable if NULL)
* @irq_shutdown:   shut down the interrupt (defaults to ->disable if NULL)
* @irq_enable:     enable the interrupt (defaults to chip->unmask if NULL)
* @irq_disable:    disable the interrupt
* @irq_ack:        start of a new interrupt
* @irq_mask:       mask an interrupt source
* @irq_mask_ack:   ack and mask an interrupt source
* @irq_unmask:     unmask an interrupt source
* @irq_eoi:        end of interrupt
```

我们在 request_irq 后，并不需要手工去使能中断，原因就是系统调用对应的 irq_chip 里的函数帮我们使能了中断。

我们提供的中断处理函数中，也不需要执行主芯片相关的清中断操作，也是系统帮我们调用 irq_chip 中的相关函数。

但是对于外部设备相关的清中断操作，还是需要我们自己做的。

就像上面图里的“外部设备 1“、“外部设备 n”，外设备千变万化，内核里可没有对应的清除中断操作。

18.4 在设备树中指定中断_在代码中获得中断

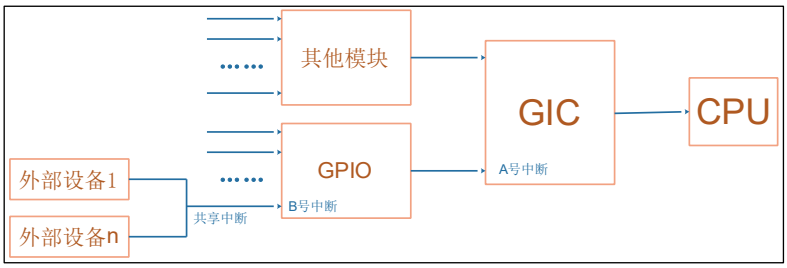
18.4.1 设备树里中断节点的语法

参考文档：

内核 Documentation\devicetree\bindings\interrupt-controller\interrupts.txt

1 设备树里的中断控制器

中断的硬件框图如下：



在硬件上，“中断控制器”只有 GIC 这一个，但是我们在软件上也可以把上图中的“GPIO”称为“中断控制器”。很多芯片有多个 GPIO 模块，比如 GPIO1、GPIO2 等等。所以软件上的“中断控制器”就有很多个：GIC、GPIO1、GPIO2 等等。

GPIO1 连接到 GIC，GPIO2 连接到 GIC，所以 GPIO1 的父亲是 GIC，GPIO2 的父亲是 GIC。

假设 GPIO1 有 32 个中断源，但是它把其中的 16 个汇聚起来向 GIC 发出一个中断，把另外 16 个汇聚起来向 GIC 发出另一个中断。这就意味着 GPIO1 会用到 GIC 的两个中断，会涉及 GIC 里的 2 个 hwirq。

这些层级关系、中断号(hwirq)，都会在设备树中有所体现。

在设备树中，中断控制器节点中必须有一个属性：interrupt-controller，表明它是“中断控制器”。

还必须有一个属性：#interrupt-cells，表明引用这个中断控制器的话需要多少个 cell。

#interrupt-cells 的值一般有如下取值：

- #interrupt-cells=<1>

别的节点要使用这个中断控制器时，只需要一个 cell 来表明使用“哪一个

中断”。

- #interrupt-cells=<2>

别的节点要使用这个中断控制器时，需要一个 cell 来表明使用“哪一个中断”；

还需要另一个 cell 来描述中断，一般是表明触发类型：

```
第 2 个 cell 的 bits[3:0] 用来表示中断触发类型(trigger type and level flags)：
  1 = low-to-high edge triggered，上升沿触发
  2 = high-to-low edge triggered，下降沿触发
  4 = active high level-sensitive，高电平触发
  8 = active low level-sensitive，低电平触发
```

示例如下：

```
vic: intc@10140000 {
    compatible = "arm,versatile-vic";
    interrupt-controller;
    #interrupt-cells = <1>;
    reg = <0x10140000 0x1000>;
};
```

如果中断控制器有级联关系，下级的中断控制器还需要表明它的“interrupt-parent”是谁，用了 interrupt-parent”中的哪一个“interrupts”，请看下一小节。

2 设备树里使用中断

一个外设，它的中断信号接到哪个“中断控制器”的哪个“中断引脚”，这个中断的触发方式是怎样的？

这 3 个问题，在设备树里使用中断时，都要有所体现。

- interrupt-parent=<&XXXX>

你要用哪一个中断控制器里的中断？

- interrupts

你要用哪一个中断？

Interrupts 里要用几个 cell，由 interrupt-parent 对应的中断控制器决定。在中断控制器里有“#interrupt-cells”属性，它指明了要用几个 cell 来描述中断。

比如：

```
i2c@7000c000 {
    gpioext: gpio-adnp@41 {
        compatible = "ad,gpio-adnp";

        interrupt-parent = <&gpio>;
        interrupts = <160 1>;

        gpio-controller;
        #gpio-cells = <1>;

        interrupt-controller;
        #interrupt-cells = <2>;
    };
```

```
......
};
```

- 新写法：interrupts-extended

一个“interrupts-extended”属性就可以既指定“interrupt-parent”，也指定“interrupts”，比如：

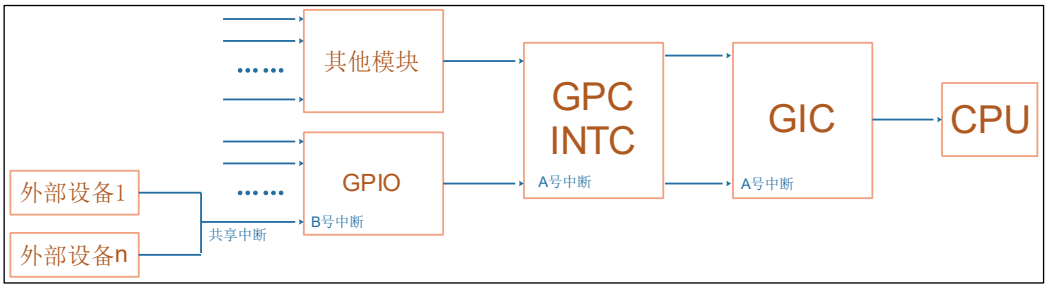
```
interrupts-extended = <&intc1 5 1>, <&intc2 1 0>;
```

18.4.2 设备树里中断节点的示例

以 100ASK_IMX6ULL 开发板为例，在 arch/arm/boot/dts 目录下可以看到 2 个文件：imx6ull.dtsi、100ask_imx6ull-14x14.dts，把里面有关中断的部分内容抽取出来。

```
intc: interrupt-controller@00a01000 {
        compatible = "arm,cortex-a7-gic";                    ---> drivers/irqchip/irq-gic.c
        #interrupt-cells = <3>;
        interrupt-controller;
        reg = <0x00a01000 0x1000>,
              <0x00a02000 0x100>;
};

soc {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        interrupt-parent = <&gpc>;

        gpc: gpc@020dc000 {
                compatible = "fsl,imx6ul-gpc", "fsl,imx6q-gpc";   ---> arch/arm/mach-imx/gpc.c
                reg = <0x020dc000 0x4000>;
                interrupt-controller;
                #interrupt-cells = <3>;
                interrupts = <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-parent = <&intc>;
                fsl,mf-mix-wakeup-irq = <0xfc00000 0x7d00 0x0 0x1400640>;
        };

        gpio1: gpio@0209c000 {
                compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";  ---> drivers/gpio/gpio-mxc.c
                reg = <0x0209c000 0x4000>;
                interrupts = <GIC_SPI 66 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 67 IRQ_TYPE_LEVEL_HIGH>;
                gpio-controller;
                #gpio-cells = <2>;
                interrupt-controller;
                #interrupt-cells = <2>;
        };                                   以上都是厂家BSP工程师实现的(imx6ull.dtsi)

        spidev: icm20608@0{
                compatible = "invensense,icm20608";                ---> drivers/iio/imu/inv_mpu6050/inv_mpu_spi.c
                interrupt-parent = <&gpio1>;
                interrupts = <1 1>;                                使用
                spi-max-frequency = <8000000>;
                reg = <0>;
        };
}
```

从设备树反推 IMX6ULL 的中断体系，如下，比之前的框图多了一个“GPC INTC”：



GPC INTC 的英文是：General Power Controller, Interrupt Controller。它提供中断屏蔽、中断状态查询功能，实际上这些功能在 GIC 里也实现了，个人觉得有点多余。除此之外，它还提供唤醒功能，这才是保留它的原因。

18.4.3 在代码中获得中断

之前我们提到过，设备树中的节点有些能被转换为内核里的platform_device，有些不能，回顾如下：

① 根节点下含有 compatile 属性的子节点，会转换为 platform_device

② 含有特定 compatile 属性的节点的子节点，会转换为 platform_device

如果一个节点的 compatile 属性，它的值是这 4 者之一："simple-bus","simple-mfd","isa","arm,amba-bus",

那么它的子结点(需含 compatile 属性)也可以转换为 platform_device。

③ 总线 I2C、SPI 节点下的子节点：不转换为 platform_device

某个总线下到子节点，应该交给对应的总线驱动程序来处理，它们不应该被转换为 platform_device。

1 对于 platform_device

一个节点能被转换为 platform_device，如果它的设备树里指定了中断属性，那么可以从 platform_device 中获得“中断资源”，函数如下，可以使用下列函数获得 IORESOURCE_IRQ 资源，即中断号：

```
/**
 * platform_get_resource - get a resource for a device
 * @dev: platform device
 * @type: resource type   // 取哪类资源？IORESOURCE_MEM、IORESOURCE_REG
*                     // IORESOURCE_IRQ 等
 * @num: resource index  // 这类资源中的哪一个？
 */
struct resource *platform_get_resource(struct platform_device *dev,
                                       unsigned int type,
                                       unsigned int num);
```

2 对于 I2C 设备、SPI 设备

对于 I2C 设备节点，I2C 总线驱动在处理设备树里的 I2C 子节点时，也会处理其中的中断信息。一个 I2C 设备会被转换为一个 i2c_client 结构体，中断号会保存在 i2c_client 的 irq 成员里,代码如下(drivers/i2c/i2c-core.c)：

```
static int i2c_device_probe(struct device *dev)
{
    struct i2c_client   *client = i2c_verify_client(dev);
    struct i2c_driver   *driver;
    int status;

    if (!client)
        return 0;

    if (!client->irq) {
        int irq = -ENOENT;

        if (dev->of_node) {
            irq = of_irq_get_byname(dev->of_node, "irq");
            if (irq == -EINVAL || irq == -ENODATA)
                irq = of_irq_get(dev->of_node, 0);
        } else if (ACPI_COMPANION(dev)) {
            irq = acpi_dev_gpio_irq_get(ACPI_COMPANION(dev), 0);
        }
        if (irq == -EPROBE_DEFER)
            return irq;
        if (irq < 0)
            irq = 0;
```

从设备树里解析出中断号

对于 SPI 设备节点，SPI 总线驱动在处理设备树里的 SPI 子节点时，也会处

理其中的中断信息。一个 SPI 设备会被转换为一个 spi_device 结构体，中断号会保存在 spi_device 的 irq 成员里，代码如下(drivers/spi/spi.c)：

```
static int spi_drv_probe(struct device *dev)
{
    const struct spi_driver     *sdrv = to_spi_driver(dev->driver);
    struct spi_device       *spi = to_spi_device(dev);
    int ret;

    ret = of_clk_set_defaults(dev->of_node, false);
    if (ret)
        return ret;

    if (dev->of_node) {
        spi->irq = of_irq_get(dev->of_node, 0);
        if (spi->irq == -EPROBE_DEFER)
            return -EPROBE_DEFER;
        if (spi->irq < 0)
            spi->irq = 0;
    }
```

从设备树里解析出中断

3 调用 of_irq_get 获得中断号

如果你的设备节点既不能转换为 platform_device，它也不是 I2C 设备，不是 SPI 设备，那么在驱动程序中可以自行调用 of_irq_get 函数去解析设备树，得到中断号。

4 对于 GPIO

参考：drivers/input/keyboard/gpio_keys.c

可以使用 gpio_to_irq 或 gpiod_to_irq 获得中断号。

举例，假设在设备树中有如下节点：

```
gpio-keys {
        compatible = "gpio-keys";
        pinctrl-names = "default";

        user {
                label = "User Button";
                gpios = <&gpio5 1 GPIO_ACTIVE_HIGH>;
                gpio-key,wakeup;
                linux,code = <KEY_1>;
        };
};
```

那么可以使用下面的函数获得引脚和 flag：

```
button->gpio = of_get_gpio_flags(pp, 0, &flags);
bdata->gpiod = gpio_to_desc(button->gpio);
```

再去使用 gpiod_to_irq 获得中断号：

```
irq = gpiod_to_irq(bdata->gpiod);
```

18.5 编写使用中断的按键驱动程序

写在前面的话：对于 GPIO 按键，我们并不需要去写驱动程序，使用内核自带的驱动程序 drivers/input/keyboard/gpio_keys.c 就可以，然后你需要做的只是修改设备树指定引脚及键值。

但是我还是要教你怎么从头写按键驱动，特别是如何使用中断。因为中断是引入其他基础知识的前提，后面要讲的这些内容都离不开中断：休眠-唤醒、POLL

机制、异步通知、定时器、中断的线程化处理。

这些基础知识是更复杂的驱动程序的基础要素，以后的复杂驱动也就是对硬件操作的封装彼此不同，但是用到的基础编程知识是一样的。

18.5.1 编程思路

1 设备树相关

查看原理图确定按键使用的引脚，再在设备树中添加节点，在节点里指定中断信息。

例子：

```
gpio_keys_100ask {
    compatible = "100ask,gpio_key";
    gpios = <&gpio5 1 GPIO_ACTIVE_HIGH
            &gpio4 14 GPIO_ACTIVE_HIGH>;
    pinctrl-names = "default";
    pinctrl-0 = <&key1_pinctrl
                &key2_pinctrl>;
};
```

2 驱动代码相关

- 首先要获得中断号，参考上面《18.4.3 在代码中获得中断》；
- 然后编写中断处理函数；
- 最后 request_irq。

18.5.2 先编写驱动程序

参考：内核源码 drivers/input/keyboard/gpio_keys.c

使用 GIT 命令载后，源码 gpio_key_drv.c 位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\06_gpio_irq\01_simple
```

1 从设备树获得 GPIO

```
count = of_gpio_count(node);
for (i = 0; i < count; i++)
    gpio_keys_100ask[i].gpio = of_get_gpio_flags(node, i, &flag);
```

2 从 GPIO 获得中断号

```
gpio_keys_100ask[i].irq = gpio_to_irq(gpio_keys_100ask[i].gpio);
```

3 申请中断

```
err = request_irq(gpio_keys_100ask[i].irq, gpio_key_isr, \
IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "100ask_gpio_key", &gpio_keys_100ask
[i]);
```

4 中断函数

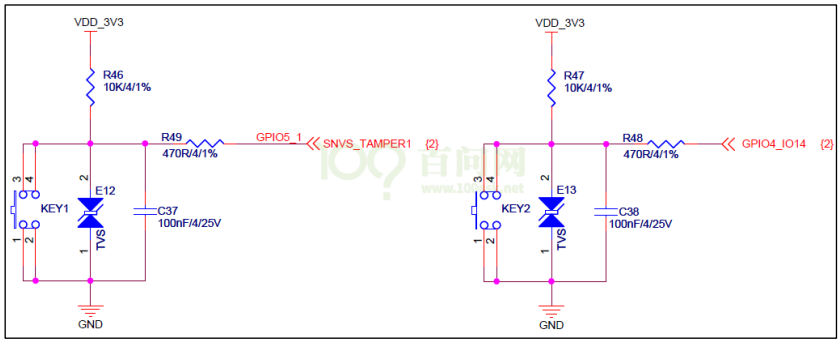
```
static irqreturn_t gpio_key_isr(int irq, void *dev_id)
{
    struct gpio_key *gpio_key = dev_id;
```

```
    int val;
    val = gpiod_get_value(gpio_key->gpiod);


    printk("key %d %d\n", gpio_key->gpio, val);

    return IRQ_HANDLED;
}
```

18.6 IMX6ULL 设备树修改及上机实验

本实验的内核版本：
https://e.coding.net/weidongshan/imx-linux4.9.88_commit_6020a20c1277c6b511e5673eecd8523e376031c8

18.6.1 查看原理图确定按键引脚



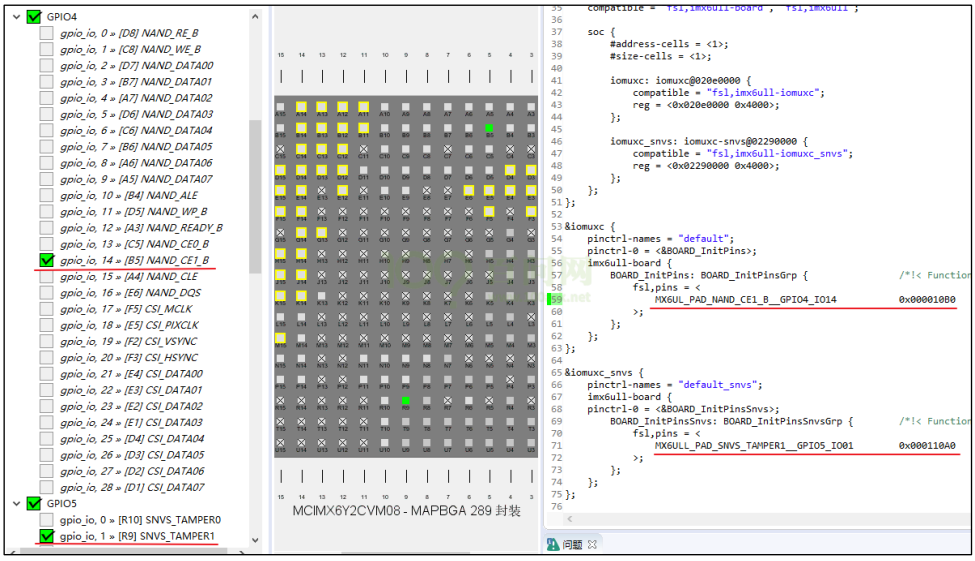
18.6.2 修改设备树

对于一个引脚要用作中断时：

a) 要通过 PinCtrl 把它设置为 GPIO 功能；

b) 表明自身：是哪一个 GPIO 模块里的哪一个引脚

运行 NXP 提供的图形化设备树配置工具“i.MX Pins Tool v6”，点击左侧选中 GPIO5_IO01、GPIO4_IO14，如下图所示：



按上图右侧去修改设备树 arch/arm/boot/dts/100ask_imx6ull-14x14.dts，修改结果放 GIT 中。

使用 GIT 命令载后，源码“修改后 100ask_imx6ull-14x14.dts”位于这个目录下(使用之前要改名为“100ask_imx6ull-14x14.dts”并上传到内核的 arch/arm/boot/dts 目录)：

01_all_series_quickstart
05_嵌入式 Linux 驱动开发基础知识\source\06_gpio_irq\01_simple\device_tree

主要内容摘录如下：

- GPIO5_IO01 的 pinctrl 定义：

```
&iomuxc_snvs {
    pinctrl-names = "default_snvs";
    pinctrl-0 = <&pinctrl_hog_2>;
    imx6ul-evk {
        key1_100ask: key1_100ask{    /*!< Function assigned for the core: Cortex-A7[ca
7] */
            fsl,pins = <
                MX6ULL_PAD_SNVS_TAMPER1__GPIO5_IO01       0x000110A0
            >;
        };
```

- GPIO4_IO14 的 pinctrl 定义：

```
&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>;
    imx6ul-evk {
        key2_100ask: key2_100ask{  /*!< Function assigned for the core: Cortex-A7[ca
7] */
            fsl,pins = <
                MX6UL_PAD_NAND_CE1_B__GPIO4_IO14          0x000010B0
            >;
        };
```

定义这 2 个按键的节点：

```
gpio_keys_100ask {
    compatible = "100ask,gpio_key";
    gpios = <&gpio5 1 GPIO_ACTIVE_HIGH
             &gpio4 14 GPIO_ACTIVE_LOW>;

    pinctrl-names = "default";
    pinctrl-0 = <&key1_100ask &key2_100ask>;
};
```

把原来的 GPIO 按键节点禁止掉：

```
gpio-keys {
    compatible = "gpio-keys";
    pinctrl-names = "default";
    status = "disabled";    // 这句是新加的
```

18.6.3 上机实验

实验步骤如下：

第1步 编译设备树，把 100ask_imx6ull-14x14.dtb 放到板子的/boot 目录，重启开发板。

第2步 编译驱动程序，安装驱动程序，操作按键。

大概命令列出如下：

- 1．在电脑上设置工具链

```
export ARCH=arm
export CROSS_COMPILE= arm-buildroot-linux-gnueabihf-
export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf_sdk-buildroot/bin
```

- 2．进入内核目录后执行：

```
make dtbs   // 生成 arch/arm/boot/dts/100ask_imx6ull-14x14.dtb，请把它放到板子的/boot目录
```

- 3．编译驱动：先进入驱动程序目录，执行 make 即可，把生成的 gpio_key_drv.ko 放到开发板上
- 4．重启开发板后，在板子上执行：

```
[root@100ask:~]# echo "7 4 1 7" > /proc/sys/kernel/printk
[root@100ask:~]# insmod gpio_key_drv.ko
```

- 5．按下、松开按键，可以看到输出信息：

```
[   48.396584] key 110 0
[   48.569403] key 110 1
[   49.321805] key 129 0
[   49.498734] key 129 1
```

参考资料

- 中断处理不能嵌套：

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58aa3d2d0cc

- genirq: add threaded interrupt handler support

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3aa551c9b4c40018f0e261a178e3d25478dc04a9

- Linux RT(2)一硬实时 Linux(RT-Preempt Patch)的中断线程化

https://www.veryarm.com/110619.html

- Linux 中断管理 (1)Linux 中断管理机制

https://www.cnblogs.com/arnoldlu/p/8659981.html

```
Breakpoint 1, gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keyboard/gpio_keys.c:393
393 {
(gdb) bt
#0  gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keyboard/gpio_keys.c:393
#1  0x80270528 in __handle_irq_event_percpu (desc=0x8616e300, flags=0x86517edc) at kernel/irq/handle.c:145
#2  0x802705cc in handle_irq_event_percpu (desc=0x8616e300) at kernel/irq/handle.c:185
#3  0x80270640 in handle_irq_event (desc=0x8616e300) at kernel/irq/handle.c:202
```

```
#4  0x802738e8 in handle_level_irq (desc=0x8616e300) at kernel/irq/chip.c:518
#5  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/i
rqdesc.h:150
#6  generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#7  0x805005e0 in mxc_gpio_irq_handler (port=0xc8, irq_stat=2252237104) at drivers/gp
io/gpio-mxc.c:274
#8  0x805006fc in mx3_gpio_irq_handler (desc=<optimized out>) at drivers/gpio/gpio-mx
c.c:291
#9  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/i
rqdesc.h:150
#10 generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#11 0x8026fd0c in __handle_domain_irq (domain=0x86006000, hwirq=32, lookup=true, regs
=0x86517fb0) at kernel/irq/irqdesc.c:627
#12 0x80201484 in handle_domain_irq (regs=<optimized out>, hwirq=<optimized out>, dom
ain=<optimized out>) at ./include/linux/irqdesc.h:168
#13 gic_handle_irq (regs=0xc8) at drivers/irqchip/irq-gic.c:364
#14 0x8020b704 in __irq_usr () at arch/arm/kernel/entry-armv.S:464
```

第19章 驱动程序基石

19.1 休眠与唤醒

19.1.1 适用场景

在前面引入中断时，我们曾经举过一个例子：

妈妈怎么知道卧室里小孩醒了？

- 休眠-唤醒：进去房间陪小孩一起睡觉，小孩醒了会吵醒她

不累，但是妈妈干不了活了

当应用程序必须等待某个事件发生，比如必须等待按键被按下时，可以使用“休眠-唤醒”机制：

① APP 调用 read 等函数试图读取数据，比如读取按键；

② APP 进入内核态，也就是调用驱动中的对应函数，发现有数据则复制到用户空间并马上返回；

③ 如果 APP 在内核态，也就是在驱动程序中发现没有数据，则 APP 休眠；

④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

⑤ APP 继续运行它的内核态代码，也就是驱动程序中的函数，复制数据到用户空间并马上返回。

驱动中有数据时，图 19.1 中红线就是 APP1 的执行过程，涉及用户态、内核态：

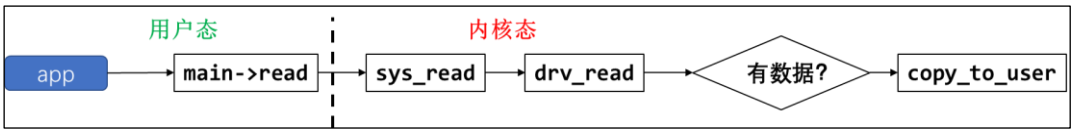


图 19.1 app1 执行过程

驱动中没有数据时，APP1 在内核态执行到 drv_read 时会休眠。所谓休眠就是把自己的状态改为非 RUNNING，这样内核的调度器就不会让它运行。当按下按键，驱动程序中的中断服务程序被调用，它会记录数据，并唤醒 APP1。所以唤醒就是把程序的状态改为 RUNNING，这样内核的调度器有合适的时间就会让它运行。

当 APP1 再次运行时，就会继续执行 drv_read 中剩下的代码，把数据复制回用户空间，返回用户空间。APP1 的执行过程如下图的红色实线所示，它被分成了 2 段：

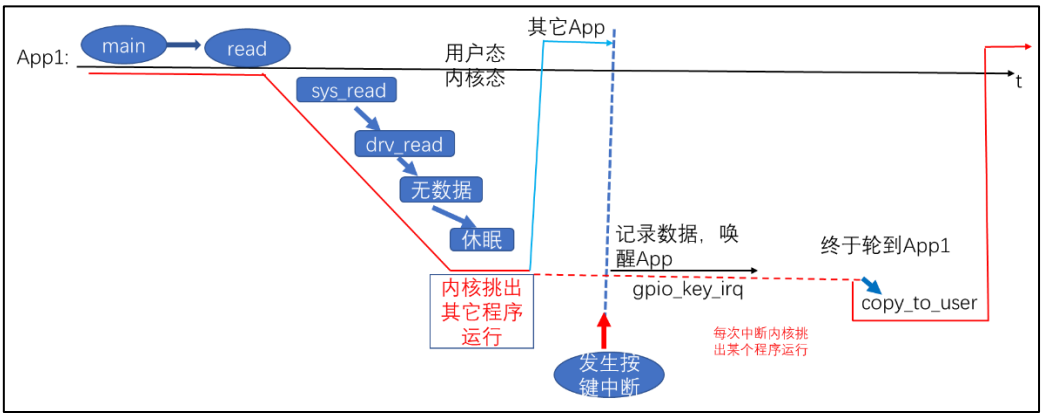


图 19.2 APP1 的执行过程

值得注意的是，上面 2 个图中红线部分都属于 APP1 的“上下文”，或者这样说：红线所涉及的代码，都是 APP1 调用的。但是按键的中断服务程序，不属于 APP1 的“上下文”，这是突如其来的，当中断发生时，APP1 正在休眠呢。

在 APP1 的“上下文”，也就是在 APP1 的执行过程中，它是可以休眠的。

在中断的处理过程中，也就是 gpio_key_irq 的执行过程中，它不能休眠：“中断”怎么能休眠？“中断”休眠了，谁来调度其他 APP 啊？

所以，请记住：**在中断处理函数中，不能休眠**，也就不能调用会导致休眠的函数。

19.1.2 内核函数

1 休眠函数

参考内核源码：include\linux\wait.h。

表 19-1 wait 内核函数

函数	说明
wait_event_interruptible(wq, condition)	休眠，直到 condition 为真； 休眠期间是可被打断的，可以被信号打断
wait_event(wq, condition)	休眠，直到 condition 为真； 退出的唯一条件是 condition 为真，信号也不好使
wait_event_interruptible_timeout(wq, condition, timeout)	休眠，直到 condition 为真或超时； 休眠期间是可被打断的，可以被信号打断
wait_event_timeout(wq, condition, timeout)	休眠，直到 condition 为真； 退出的唯一条件是 condition 为真，信号也不好使

比较重要的参数就是：

① wq：waitqueue，等待队列

a) 休眠时除了把程序状态改为非 RUNNING 之外，还要把进程/进程放入 wq 中，以后中断服务程序要从 wq 中把它取出来唤醒。

b) 没有 wq 的话，茫茫人海中，中断服务程序去哪里找到你？

② condition

这可以是一个变量，也可以是任何表达式。表示“一直等待，直到 condition 为真”。

2 唤醒函数

参考内核源码：include\linux\wait.h。

函数	说明
wake_up_interruptible(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，只唤醒其中的一个线程
wake_up_interruptible_nr(x, nr)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，只唤醒其中的 nr 个线程
wake_up_interruptible_all(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，唤醒其中的所有线程
wake_up(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，只唤醒其中的一个线程
wake_up_nr(x, nr)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，只唤醒其中 nr 个线程
wake_up_all(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，唤醒其中的所有线程

19.1.3 驱动框架

驱动框架如下：

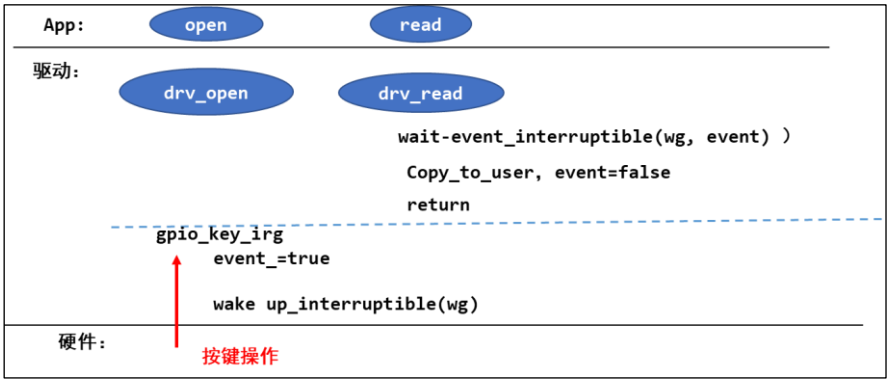


图 19.3 驱动框架

要休眠的线程，放在 wq 队列里，中断处理函数从 wq 队列里把它取出来唤醒。

所以，我们要做这几件事：

① 初始化 wq 队列

② 在驱动的 read 函数中，调用 wait_event_interruptible：

它本身会判断 event 是否为 FALSE，如果为 FASLE 表示无数据，则休眠。

当从 wait_event_interruptible 返回后，把数据复制回用户空间。

③ 在中断服务程序里：

设置 event 为 TRUE，并调用 wake_up_interruptible 唤醒线程。

19.1.4 编程

使用 GIT 命令载后，源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\02_read_key_irq\ 和 03_read_key_irq_circle_buffer
```

03_read_key_irq_circle_buffer 使用了环型缓冲区，可以避免按键丢失。

1 驱动程序关键代码

- 02_read_key_irq\gpio_key_drv.c 中，要先定义“wait queue”：

```
41 static DECLARE_WAIT_QUEUE_HEAD(gpio_key_wait);
```

- 在驱动的读函数里调用 wait_event_interruptible：

```
44 static ssize_t gpio_key_drv_read (struct file *file, char __user *buf, size_t siz
e, loff_t *offset)
45 {
46     //printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
47     int err;
48
49     wait_event_interruptible(gpio_key_wait, g_key);
50     err = copy_to_user(buf, &g_key, 4);
51     g_key = 0;
52
53     return 4;
54 }
```

第 49 行并不一定会进入休眠，它会先判断 g_key 是否为 TRUE。

执行到第 50 行时，表示要么有了数据(g_key 为 TRUE)，要么有信号等待处理(本节课程不涉及信号)。

假设 g_key 等于 0，那么 APP 会执行到上述代码第 49 行时进入休眠状态。它被谁唤醒？被控制的中断服务程序：

```
64 static irqreturn_t gpio_key_isr(int irq, void *dev_id)
65 {
66     struct gpio_key *gpio_key = dev_id;
67     int val;
68     val = gpiod_get_value(gpio_key->gpiod);
69
70
71     printk("key %d %d\n", gpio_key->gpio, val);
72     g_key = (gpio_key->gpio << 8) | val;
73     wake_up_interruptible(&gpio_key_wait);
74
75     return IRQ_HANDLED;
76 }
```

上述代码中，第 72 行确定按键值 g_key，g_key 也就变为 TRUE 了。

然后在第 73 行唤醒 gpio_key_wait 中的第 1 个线程。

注意这 2 个函数，一个没有使用“&”，另一个使用了“&”：

```
wait_event_interruptible(gpio_key_wait, g_key);
wake_up_interruptible(&gpio_key_wait);
```

2 应用程序

应用程序并不复杂，调用 open、read 即可，代码在 button_test.c 中：

```
25      /* 2. 打开文件 */
26      fd = open(argv[1], O_RDWR);
27      if (fd == -1)
28      {
29              printf("can not open file %s\n", argv[1]);
30              return -1;
31      }
32
33      while (1)
34      {
35              /* 3. 读文件 */
36              read(fd, &val, 4);
37              printf("get button : 0x%x\n", val);
38      }
```

在 33 行～38 行的循环中，APP 基本上都是休眠状态。你可以执行 top 命令查看 CPU 占用率。

19.1.5 上机实验

跟上一节视频类似，需要先修改设备树，请使用上一节视频的设备树文件。然后安装驱动程序，运行测试程序。

```
[root@100ask:~]# insmod -f gpio_key_drv.ko
[root@100ask:~]# ls /dev/100ask_gpio_key
/dev/100ask_gpio_key
[root@100ask:~]# ./button_test /dev/100ask_gpio_key  &
[root@100ask:~]# top
```

19.1.6 使用环形缓冲区改进驱动程序

使用 GIT 命令载后，源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\03_read_key_irq_circle_buffer
```

使用环形缓冲区，可以在一定程序上避免按键数据丢失，关键代码如下：

```
/* 环形缓冲区 */
#define BUF_LEN 128
static int g_keys[BUF_LEN];
static int r, w;                    // r,w是读写位置

#define NEXT_POS(x) ((x+1) % BUF_LEN)

static int is_key_buf_empty(void)
{
    return (r == w);  // 一开始r,w都是0, r==w表示空
}

static int is_key_buf_full(void)
{
    return (r == NEXT_POS(w));   // 下一个写的位置等于r，表示满
}                                // 容量为128的buffer,
                                 // 存有127个数据时我们就认为满了
static void put_key(int key)
{
    if (!is_key_buf_full())
    {
        g_keys[w] = key;     // 把数据放入w位置
        w = NEXT_POS(w);     // 移动w
    }
}

static int get_key(void)
{
    int key = 0;
    if (!is_key_buf_empty())
    {
        key = g_keys[r];     // 从r位置读数据
        r = NEXT_POS(r);     // 移动r
    }
    return key;
}
```

使用环形缓冲区之后，休眠函数可以这样写：

```
86      wait_event_interruptible(gpio_key_wait, !is_key_buf_empty());
87      key = get_key();
88      err = copy_to_user(buf, &key, 4);
```

唤醒函数可以这样写：

```
111     key = (gpio_key->gpio << 8) | val;
112     put_key(key);
113     wake_up_interruptible(&gpio_key_wait);
```

19.2 POLL 机制

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\06_gpio_irq\04_read_key_irq_poll
```

19.2.1 适用场景

在前面引入中断时，我们曾经举过一个例子：妈妈怎么知道卧室里小孩醒了？

- poll 方式：妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟

要浪费点时间，但是可以继续干活。妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

使用休眠-唤醒的方式等待某个事件发生时，有一个缺点：等待的时间可能很久。我们可以加上一个超时时间，这时就可以使用 poll 机制。

① APP 不知道驱动程序中是否有数据，可以先调用 poll 函数查询一下，poll 函数可以传入超时时间；

② APP 进入内核态，调用到驱动程序的 poll 函数，如果有数据的话立刻返回；

③ 如果发现没有数据时就休眠一段时间；

④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

⑤ 当超时时间到了之后，内核也会唤醒 APP；

⑥ APP 根据 poll 函数的返回值就可以知道是否有数据，如果有数据就调用 read 得到数据。

19.2.2 使用流程

妈妈进入房间时，会先看小孩醒没醒，闹钟响之后走出房间之前又会再看小孩醒没醒。

注意：看了 2 次小孩！

POLL 机制也是类似的，流程如下：

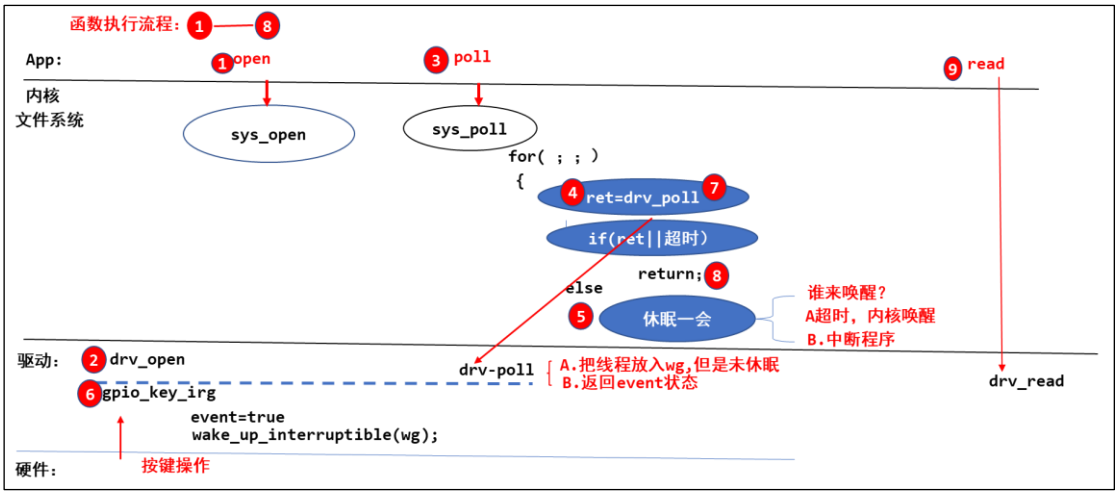


图 19.4 poll 机制

函数执行流程如上图①～⑧所示，重点从③开始看。假设一开始无按键数据：

③PP 调用 poll 之后，进入内核态；

④致驱动程序的 drv_poll 被调用：

注意，drv_poll 要把自己这个线程挂入等待队列 wq 中；假设不放入队列里，那以后发生中断时，中断服务程序去哪里找到你嘛？

drv_poll 还会判断一下：有没有数据啊？返回这个状态。

⑤当前没有数据，则休眠一会；

⑥过程中，按下了按键，发生了中断：

- 在中断服务程序里记录了按键值，并且从 wq 中把线程唤醒了。

⑦从休眠中被唤醒，继续执行 for 循环，再次调用 drv_poll：

- drv_poll 返回数据状态

⑧哦，你有数据，那从内核态返回到应用态吧

⑨APP 调用 read 函数读数据

如果一直没有数据，调用流程也是类似的，重点从③开始看，如下：

③ APP 调用 poll 之后，进入内核态；

④ 导致驱动程序的 drv_poll 被调用：

注意，drv_poll 要把自己这个线程挂入等待队列 wq 中；假设不放入队列里，那以后发生中断时，中断服务程序去哪里找到你嘛？

drv_poll 还会判断一下：有没有数据啊？返回这个状态。

⑤ 假设当前没有数据，则休眠一会；

⑥ 在休眠过程中，一直没有按下了按键，超时时间到：内核把这个线程唤醒；

⑦ 线程从休眠中被唤醒，继续执行 for 循环，再次调用 drv_poll：drv_poll 返回数据状态

⑧ 哦，你还是没有数据，但是超时时间到了，那从内核态返回到应用态吧

⑨ APP 不能调用 read 函数读数据

注意几点：

- drv_poll 要把线程挂入队列 wq，但是并不是在 drv_poll 中进入休眠，而是在调用 drv_poll 之后休眠
- drv_poll 要返回数据状态
- APP 调用一次 poll，有可能会导致 drv_poll 被调用 2 次
- 线程被唤醒的原因有 2：中断发生了去队列 wq 中把它唤醒，超时时间到了内核把它唤醒
- APP 要判断 poll 返回的原因：有数据，还是超时。有数据时再去调用 read 函数。

19.2.3 驱动编程

使用 poll 机制时，驱动程序的核心就是提供对应的 drv_poll 函数。在 drv_poll 函数中要做 2 件事：

① 把当前线程挂入队列 wq：poll_wait

a) APP 调用一次 poll，可能导致 drv_poll 被调用 2 次，但是我们并不需要把当前线程挂入队列 2 次。

b) 可以使用内核的函数 poll_wait 把线程挂入队列，如果线程已经在队列里了，它就不会再次挂入。

② 返回设备状态：

APP 调用 poll 函数时，有可能是查询“有没有数据可以读”：POLLIN，也有可能是查询“你有没有空间给我写数据”：POLLOUT。所以 drv_poll 要返回自己的当前状态：(POLLIN | POLLRDNORM) 或 (POLLOUT | POLLWRNORM)。

a) POLLRDNORM 等同于 POLLIN，为了兼容某些 APP 把它们一起返回。

b) POLLWRNORM 等同于 POLLOUT ，为了兼容某些 APP 把它们一起返回。

APP 调用 poll 后，很有可能会休眠。对应的，在按键驱动的中断服务程序中，也要有唤醒操作。

驱动程序中 poll 的代码如下：

```
static unsigned int gpio_key_drv_poll(struct file *fp, poll_table * wait)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    poll_wait(fp, &gpio_key_wait, wait);
    return is_key_buf_empty() ? 0 : POLLIN | POLLRDNORM;
}
```

19.2.4 应用编程

注意：APP 可以调用 poll 或 select 函数，这 2 个函数的作用是一样的。

poll/select 函数可以监测多个文件，可以监测多种事件：

事件类型	说明
POLLIN	有数据可读
POLLRDNORM	等同于 POLLIN
POLLRDBAND	Priority band data can be read，有优先级较较高的“band data”可读

	Linux 系统中很少使用这个事件
POLLPRI	高优先级数据可读
POLLOUT	可以写数据
POLLWRNORM	等同于 POLLOUT
POLLWRBAND	Priority data may be written
POLLERR	发生了错误
POLLHUP	挂起
POLLNVAL	无效的请求，一般是 fd 未 open

在调用 poll 函数时，要指明：

- 你要监测哪一个文件：哪一个 fd
- 你想监测这个文件的哪种事件：是 POLLIN、还是 POLLOUT

最后，在 poll 函数返回时，要判断状态。

应用程序代码如下：

```
struct pollfd fds[1];
int timeout_ms = 5000;
int ret;

fds[0].fd = fd;
fds[0].events = POLLIN;

ret = poll(fds, 1, timeout_ms);
if ((ret == 1) && (fds[0].revents & POLLIN))
{
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

19.2.5 现场编程

驱动程序、应用程序的核心都在上面的章节列了出来，怎么写程序？观看视频体验更佳，更能抓住重点。

19.2.6 上机实验

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\06_gpio_irq\04_read_key_irq_poll
```

首先，把“04_read_key_irq_poll”整个目录上传单 Ubuntu 中，修改里面的 Makefile 指定内核路径，如下修改：

```
KERN_DIR =  /home/book/100ask_imx6ull-sdk/Linux-4.9.88
```

然后执行“make”命令，生成驱动程序“gpio_key_drv.ko”、应用程序“button_test”。

接着，还需要修改设备树，参照下图修改 Ubuntu 中的设备树文件“Linux-4.9.88/arch/arm/boot/dts/100ask_imx6ull-14x14.dts”：

```
101     gpio_keys_100ask {
102         compatible = "100ask,gpio_key";
103                 gpios = <&gpio5 1 GPIO_ACTIVE_LOW
104                  &gpio4 14 GPIO_ACTIVE_LOW>;
105
106         pinctrl-names = "default";
107     };
108
109     gpio-keys {
110         compatible = "gpio-keys";
111         pinctrl-names = "default";
112         status = "disabled";
113
114         user1 {
115             label = "User1 Button";
116             gpios = <&gpio5 1 GPIO_ACTIVE_LOW>;
117             gpio-key,wakeup;
118             linux,code = <KEY_1>;
119         };
```

图 19.5 POLL 机制实验的设备树

修改完设备树后，在内核目录下载执行“make dtbs”生产新的设备树文件“arch/arm/boot/dts/100ask_imx6ull-14x14.dtb”，把它放到开发板的“/boot”目录下，重启开发板。

最后，在开发板上通过 NFS 挂载 Ubuntu 的目录后，执行如下命令测试：

```
[root@100ask:~]#
[root@100ask:~]# ls /sys/firmware/devicetree/base/gpio_keys_100ask/ 1.确定设备树已经修改
compatible  gpios  name  pinctrl-names
[root@100ask:~]#
[root@100ask:~]#
[root@100ask:~]# insmod /mnt/04_read_key_irq_poll/gpio_key_drv.ko 2.安装驱动程序
[  115.288579] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_init line 224
[  115.305827] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_probe line 139
[root@100ask:~]# ls /dev/100ask_gpio_key -l 3.确定生成了设备节点
crw------- 1 root root 240, 0 Aug  2 06:23 /dev/100ask_gpio_key
[root@100ask:~]#
[root@100ask:~]# /mnt/04_read_key_irq_poll/button_test 4.直接执行测试程序，查看用法
Usage: /mnt/04_read_key_irq_poll/button_test <dev>
[root@100ask:~]#
[root@100ask:~]# /mnt/04_read_key_irq_poll/button_test /dev/100ask_gpio_key 5.执行测试程序
[  140.939056] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_poll line 96
[  145.940745] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_poll line 96
timeout[  145.950800] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_poll line 96

[  146.077690] key 129 0   6.按下、松开按键，观察输出信息
[  146.080591] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_poll line 96
get button : 0x8100[  146.089573] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_p
oll line 96

[  146.265985] key 129 1
[  146.269197] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_poll line 96
get button : 0x8101[  146.277946] /home/book/tmp/04_read_key_irq_poll/gpio_key_drv.c gpio_key_drv_p
oll line 96
```

图 19.6 POLL 机制上机实验

19.2.7 POLL 机制的内核代码详解

Linux APP 系统调用，基本都可以在它的名字前加上“sys_”前缀，这就是它在内核中对应的函数。比如系统调用 open、read、write、poll，与之对应的内核函数为：sys_open、sys_read、sys_write、sys_poll。

对于系统调用 poll 或 select，它们对应的内核函数都是 sys_poll。分析

sys_poll，即可理解 poll 机制。

1 sys_poll 函数

sys_poll 位于 fs/select.c 文件中，代码如下：

```
SYSCALL_DEFINE3(poll, struct pollfd __user *, ufds, unsigned int, nfds,
        int, timeout_msecs)
{
    struct timespec64 end_time, *to = NULL;
    int ret;

    if (timeout_msecs >= 0) {
        to = &end_time;
        poll_select_set_timeout(to, timeout_msecs / MSEC_PER_SEC,
            NSEC_PER_MSEC * (timeout_msecs % MSEC_PER_SEC));
    }

    ret = do_sys_poll(ufds, nfds, to);
……
```

- SYSCALL_DEFINE3 是一个宏，它定义于 include/linux/syscalls.h，展开后就有 sys_poll 函数。
- sys_poll 对超时参数稍作处理后，直接调用 do_sys_poll。

2 do_sys_poll 函数

do_sys_poll 位于 fs/select.c 文件中，我们忽略其他代码，只看关键部分：

```
int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
        struct timespec64 *end_time)
{
……
    poll_initwait(&table);
    fdcount = do_poll(head, &table, end_time);
    poll_freewait(&table);
……
}
```

poll_initwait 函数非常简单，它初始化一个 poll_wqueues 变量 table：

```
poll_initwait
init_poll_funcptr(&pwq->pt, __pollwait);
pt->qproc = qproc;
```

即 `table->pt->qproc = __pollwait`，__pollwait 将在驱动的 poll 函数里用到。do_poll 函数才是核心，继续看代码。

3 do_poll 函数

do_poll 函数位于 fs/select.c 文件中，这是 POLL 机制中最核心的代码，贴图如下：

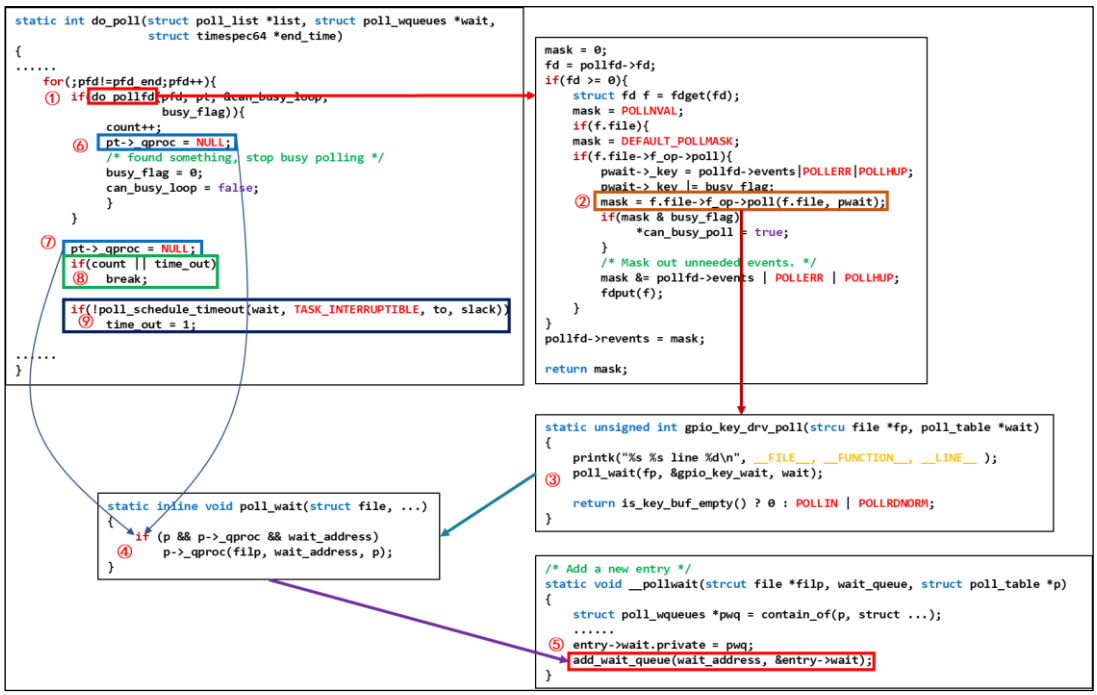


图 19.7 do_poll 机制

① 从这里开始，将会导致驱动程序的 poll 函数被第一次调用。

沿着②③④⑤，你可以看到：驱动程序里的 poll_wait 会调用__pollwait 函数把线程放入某个队列。

当执行完①之后，在⑥或⑦处，pt->_qproc 被设置为 NULL，所以第二次调用驱动程序的 poll 时，不会再次把线程放入某个队列里。

⑧ 如果驱动程序的 poll 返回有效值，则 count 非 0，跳出循环；

⑨ 否则休眠一段时间；当休眠时间到，或是被中断唤醒时，会再次循环、再次调用驱动程序的 poll。

回顾 APP 的代码，APP 可以指定“想等待某些事件”，poll 函数返回后，可以知道“发生了哪些事件”：

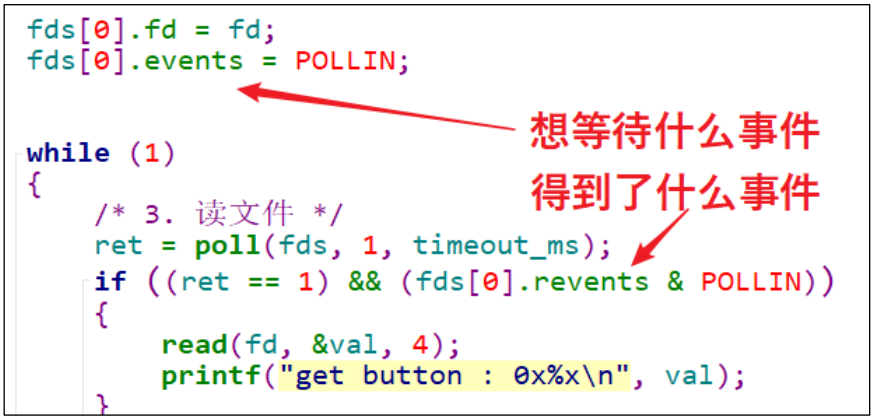


图 19.8 POLL 机制的 APP 核心代码

驱动程序里怎么体现呢？在上上一个图中，看②位置处，细说如下：

```
        if (f.file) {
            mask = DEFAULT_POLLMASK;
            if (f.file->f_op->poll) {
                pwait->_key = pollfd->events|POLLERR|POLLHUP;
                pwait->_key |= busy_flag;
                mask = f.file->f_op->poll(f.file, pwait);
                if (mask & busy_flag)
                    *can_busy_poll = true;
            }
            /* Mask out unneeded events. */
            mask &= pollfd->events | POLLERR | POLLHUP;
            fdput(f);
        }
    }
    pollfd->revents = mask;
```

1. 驱动程序返回的状态

2. 是否是APP期待的？

3. 写入revents

图 19.9 POLL 机制的内核核心代码

19.3 异步通知

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\05_read_key_irq_poll_fasync
```

19.3.1 适用场景

在前面引入中断时，我们曾经举过一个例子：

妈妈怎么知道卧室里小孩醒了？

- 异步通知：妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈
  - 妈妈、小孩互不耽误。

使用休眠-唤醒、POLL 机制时，都需要休眠等待某个事件发生时，它们的差别在于后者可以指定休眠的时长。

在现实生活中：妈妈可以不陪小孩睡觉，小孩醒了之后可以主动通知妈妈。

如果 APP 不想休眠怎么办？也有类似的方法：驱动程序有数据时主动通知 APP，APP 收到信号后执行信息处理函数。

什么叫“异步通知”？

- 你去买奶茶：
  - 你在旁边等着，眼睛盯着店员，生怕别人插队，他一做好你就知道：你是主动等待他做好，这叫“同步”。
  - 你付钱后就去玩手机了，店员做好后他会打电话告诉你：你是被动获得结果，这叫“异步”。

19.3.2 使用流程

驱动程序怎么通知 APP：发信号，这只有 3 个字，却可以引发很多问题：

① 谁发：驱动程序发
② 发什么：信号
③ 发什么信号：SIGIO
④ 怎么发：内核里提供有函数
⑤ 发给谁：APP，APP 要把自己告诉驱动
⑥ APP 收到后做什么：执行信号处理函数

⑦ 信号处理函数和信号，之间怎么挂钩：APP 注册信号处理函数

小孩通知妈妈的事情有很多：饿了、渴了、想找人玩。

Linux 系统中也有很多信号，在 Linux 内核源文件 include\uapi\asm-generic\signal.h 中，有很多信号的宏定义：

```
#define SIGHUP       1
#define SIGINT       2
#define SIGQUIT      3
#define SIGILL       4
#define SIGTRAP      5
#define SIGABRT      6
#define SIGIOT       6
#define SIGBUS       7
#define SIGFPE       8
#define SIGKILL      9
#define SIGUSR1     10
#define SIGSEGV     11
#define SIGUSR2     12
#define SIGPIPE     13
#define SIGALRM     14
#define SIGTERM     15
#define SIGSTKFLT   16
#define SIGCHLD     17
#define SIGCONT     18
#define SIGSTOP     19
#define SIGTSTP     20
#define SIGTTIN     21
#define SIGTTOU     22
#define SIGURG      23
#define SIGXCPU     24
#define SIGXFSZ     25
#define SIGVTALRM   26
#define SIGPROF     27
#define SIGWINCH    28
#define SIGIO       29   <-- 驱动常用信号，表示有IO事件
#define SIGPOLL     SIGIO
```

就 APP 而言，你想处理 SIGIO 信息，那么需要提供信号处理函数，并且要跟 SIGIO 挂钩。这可以通过一个 signal 函数来“给某个信号注册处理函数”，用法如下：

```
#include <signal.h>

typedef void (*sighandler_t)(int); // 1. 先编写函数

sighandler_t signal(int signum, sighandler_t handler); // 2. 注册
                    哪个信号？      信号处理函数
```

APP 还要做什么事？想想这几个问题：

a) 内核里有那么多驱动，你想让哪一个驱动给你发 SIGIO 信号？

APP 要打开驱动程序的设备节点。

b) 驱动程序怎么知道要发信号给你而不是别人？

APP 要把自己的进程 ID 告诉驱动程序。

c) APP 有时候想收到信号，有时候又不想收到信号：

应该可以把 APP 的意愿告诉驱动。

驱动程序要做什么？发信号。

a) APP 设置进程 ID 时，驱动程序要记录下进程 ID；

b) APP 还要使能驱动程序的异步通知功能，驱动中有对应的函数：

APP 打开驱动程序时，内核会创建对应的 file 结构体，file 中有 f_flags；f_flags 中有一个 FASYNC 位，它被设置为 1 时表示使能异步通知功能。当 f_flags 中的 FASYNC 位发生变化时，驱动程序的 fasync 函数被调用。

d) 发生中断时，有数据时，驱动程序调用内核辅助函数发信号。

这个辅助函数名为 kill_fasync。完美！

APP 收到信号后，是怎么执行信号处理函数的？这个，很难，有兴趣的话就看本节最后的文档。初学者没必要看。

综上所述，使用异步通知，也就是使用信号的流程如下图所示：

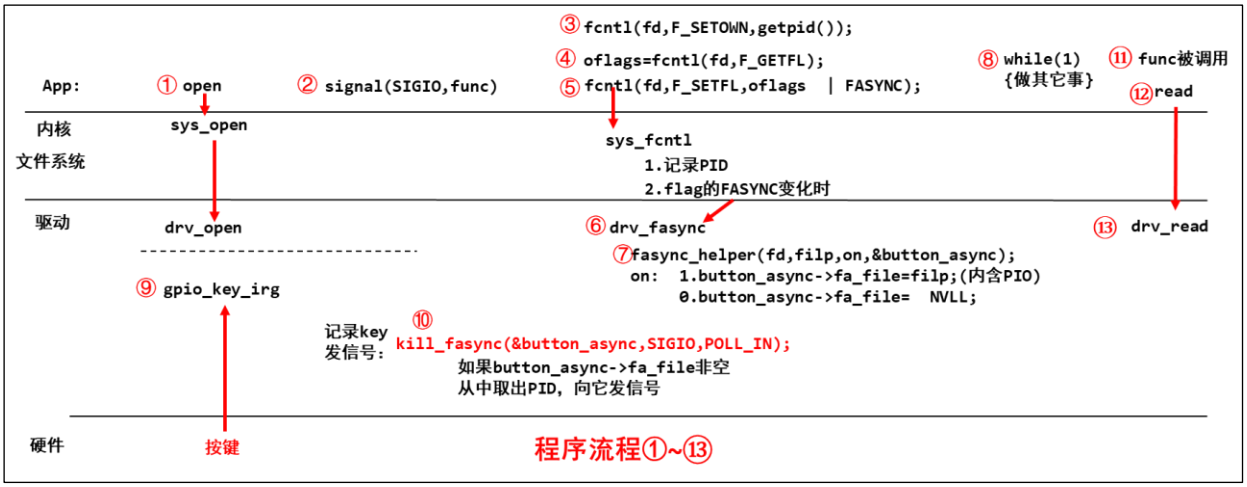


图 19.10 异步通知的信号流程

重点从②开始：

② APP 给 SIGIO 这个信号注册信号处理函数 func，以后 APP 收到 SIGIO 信号时，这个函数会被自动调用；

③ 把 APP 的 PID(进程 ID)告诉驱动程序，这个调用不涉及驱动程序，在内核的文件系统层次记录 PID；

④ 读取驱动程序文件 Flag；

⑤ 设置 Flag 里面的 FASYNC 位为 1：当 FASYNC 位发生变化时，会导致驱动程序的 fasync 被调用；

⑥⑦ 调用 faync_helper，它会根据 FAYSNC 的值决定是否设置 button_async->fa_file=驱动文件 filp：

驱动文件 filp 结构体里面含有之前设置的 PID。

⑧ APP 可以做其他事；

⑨⑩ 按下按键，发生中断，驱动程序的中断服务程序被调用，里面调用 kill_fasync 发信号；

⑪⑫⑬ APP 收到信号后，它的信号处理函数被自动调用，可以在里面调用 read 函数读取按键。

19.3.3 驱动编程

使用异步通知时，驱动程序的核心有 2：

① 提供对应的 drv_fasync 函数；

② 并在合适的时机发信号。

drv_fasync 函数很简单，调用 fasync_helper 函数就可以，如下：

```
static struct fasync_struct *button_async;
```

```
static int drv_fasync (int fd, struct file *filp, int on)
{
    return fasync_helper (fd, filp, on, &button_async);
}
```

fasync_helper 函数会分配、构造一个 fasync_struct 结构体 button_async：

- 驱动文件的 flag 被设置为 FAYNC 时：

```
button_async->fa_file = filp;  // filp 表示驱动程序文件，里面含有之前设置的 PID
```

- 驱动文件被设置为非 FASYNC 时：

```
button_async->fa_file = NULL;
```

以后想发送信号时，使用 button_async 作为参数就可以，它里面“可能”含有 PID。

什么时候发信号呢？在本例中，在 GPIO 中断服务程序中发信号。

怎么发信号呢？代码如下：

```
kill_fasync (&button_async, SIGIO, POLL_IN);
```

① 第 1 个参数：button_async->fa_file 非空时，可以从中得到 PID，表示发给哪一个 APP；

② 第 2 个参数表示发什么信号：SIGIO；

③ 第 3 个参数表示为什么发信号：POLL_IN，有数据可以读了。(APP 用不到这个参数)

19.3.4 应用编程

应用程序要做的事情有这几件：

① 编写信号处理函数：

```
static void sig_func(int sig)
{
    int val;
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

② 注册信号处理函数：

```
signal(SIGIO, sig_func);
```

③ 打开驱动：

```
fd = open(argv[1], O_RDWR);
```

④ 把进程 ID 告诉驱动：

```
fcntl(fd, F_SETOWN, getpid());
```

⑤ 使能驱动的 FASYNC 功能：

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
```

19.3.5 现场编程

驱动程序、应用程序的核心都在上面的章节列了出来，怎么写程序？观看视频体验更佳，更能抓住重点。

19.3.6 上机实验

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\05_read_key_irq_poll_fasync
```

首先，把“05_read_key_irq_poll_fasync”整个目录上传单 Ubuntu 中，修改里面的 Makefile 指定内核路径，如下修改：

```
KERN_DIR =  /home/book/100ask_imx6ull-sdk/Linux-4.9.88
```

然后执行“make”命令，生成驱动程序“gpio_key_drv.ko”、应用程序“button_test”。

接着，还需要修改设备树，参照下图修改 Ubuntu 中的设备树文件“Linux-4.9.88/arch/arm/boot/dts/100ask_imx6ull-14x14.dts”：

```
101     gpio_keys_100ask {
102         compatible = "100ask,gpio_key";
103                 gpios = <&gpio5 1 GPIO_ACTIVE_LOW
104                  &gpio4 14 GPIO_ACTIVE_LOW>;
105
106         pinctrl-names = "default";
107     };
108
109     gpio-keys {
110         compatible = "gpio-keys";
111         pinctrl-names = "default";
112         status = "disabled";
113
114         user1 {
115             label = "User1 Button";
116             gpios = <&gpio5 1 GPIO_ACTIVE_LOW>;
117             gpio-key,wakeup;
118             linux,code = <KEY_1>;
119         };
```

图 19.11 异步通知机制实验的设备树

修改完设备树后，在内核目录下载执行“make dtbs”生产新的设备树文件“arch/arm/boot/dts/100ask_imx6ull-14x14.dtb”，把它放到开发板的“/boot”目录下，重启开发板。

最后，在开发板上通过 NFS 挂载 Ubuntu 的目录后，执行如下命令测试：

```
[root@100ask:~]#
[root@100ask:~]# ls /sys/firmware/devicetree/base/gpio_keys_100ask/ 1.确定设备树已经修改
compatible  gpios  name  pinctrl-names
[root@100ask:~]#
[root@100ask:~]# insmod /mnt/05_read_key_irq_poll_fasync/gpio_key_drv.ko 2.安装驱动程序
[  870.582134] /home/book/nfs_rootfs/05_read_key_irq_poll_fasync/gpio_key_drv.c gpio_key_init line 236
[  870.598863] /home/book/nfs_rootfs/05_read_key_irq_poll_fasync/gpio_key_drv.c gpio_key_probe line 151
[root@100ask:~]#
[root@100ask:~]# ls /dev/100ask_gpio_key 3.确定生成了设备节点
/dev/100ask_gpio_key
[root@100ask:~]#
[root@100ask:~]# [  875.973498] RTL871X: RTW_ADAPTIVITY_EN_[  875.977268] AUTO, chplan:0x20, Regulation:3
,3
[  875.981770] RTL871X: RTW_ADAPTIVITY_MODE_[  875.985676] NORMAL
[  878.410786] RTL871X: nolinked power save leave

[root@100ask:~]# [  880.274193] RTL871X: nolinked power save enter

[root@100ask:~]# /mnt/05_read_key_irq_poll_fasync/button_test 4.直接执行测试程序，查看用法
Usage: /mnt/05_read_key_irq_poll_fasync/button_test <dev>
[root@100ask:~]#
[root@100ask:~]# /mnt/05_read_key_irq_poll_fasync/button_test /dev/100ask_gpio_key 5.执行测试程序
www.100ask.net
www.100ask.net
[  894.871846] key 129 0 6.按下、松开按键，观察输出信息
get button : 0x8100
www.100ask.net
[  895.054998] key 129 1
get button : 0x8101
```

图 19.12 异步通知机制上机实验

19.3.7 异步通知机制内核代码详解

异步通知的本质是“发信号”，涉及 2 个对象：发送者、接收者。发送者可以是驱动程序，可以是进程；接收者必定是进程。驱动程序要想给进程发送信号，有 2 个问题需要解决：

① 使能驱动程序的“异步”功能，即：允许它发出信号

② 告诉驱动程序，发信号时，发给“谁”

应用编程时，需要执行如下操作：

- 打开驱动：

```
fd = open(“/dev/xxx”, O_RDWR);
```

- 把进程 ID 告诉驱动：

```
fcntl(fd, F_SETOWN, getpid());
```

- 使能驱动的 FASYNC 功能：

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
```

对于 F_SETOWN、F_GETFL、F_SETFL，内核或驱动程序如何处理？

APP 执行 fcntl 系统调用时，会导致内核“fs/fcntl.c”的如下函数被调用：

```
361: SYSCALL_DEFINE3(fcntl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
362: {
363:     struct fd f = fdget_raw(fd);
364:     long err = -EBADF;
365:
366:     if (!f.file)
367:         goto ↓out;
368:
369:     if (unlikely(f.file->f_mode & FMODE_PATH)) {
370:         if (!check_fcntl_cmd(cmd))
371:             goto ↓out1;
372:     }
373:
374:     err = security_file_fcntl(f.file, cmd, arg);
375:     if (!err)
376:         err = do_fcntl(fd, cmd, arg, f.file);
377:
378: out1:
379:     fdput(f);
380: out:
381:     return err;
382: } « end SYSCALL_DEFINE3 »
383:
```

图 19.13 异步通知机制系统调用接口

在“do_fcntl”函数中，对于“F_SETOWN”，一路查看代码，发现最终如下设置：

```
84: static void f_modown(struct file *filp, struct pid *pid, enum pid_type type,
85:                       int force)
86: {
87:     write_lock_irq(&filp->f_owner.lock);
88:     if (force || !filp->f_owner.pid) {
89:         put_pid(filp->f_owner.pid);
90:         filp->f_owner.pid = get_pid(pid);
91:         filp->f_owner.pid_type = type;
92:
93:         if (pid) {
94:             const struct cred *cred = current_cred();
95:             filp->f_owner.uid = cred->uid;
96:             filp->f_owner.euid = cred->euid;
97:         }
98:     }
99:     write_unlock_irq(&filp->f_owner.lock);
100: }
101:
```

在filp里记录PID

图 19.14 异步通知机制 F_SETOWN 内部实现

在“do_fcntl”函数中，对于“F_GETFL”，仅仅是返回“filp->f_flags”；对于“F_SETFL”，会调用“setfl”函数进一步处理。代码如下：

```
248: static long do_fcntl(int fd, unsigned int cmd, unsigned long arg,
249:         struct file *filp)
250: {
251:     long err = -EINVAL;
252:
253:     switch (cmd) {
254:     case F_DUPFD:
255:         err = f_dupfd(arg, filp, 0);
256:         break;
257:     case F_DUPFD_CLOEXEC:
258:         err = f_dupfd(arg, filp, O_CLOEXEC);
259:         break;
260:     case F_GETFD:
261:         err = get_close_on_exec(fd) ? FD_CLOEXEC : 0;
262:         break;
263:     case F_SETFD:
264:         err = 0;
265:         set_close_on_exec(fd, arg & FD_CLOEXEC);
266:         break;
267:     case F_GETFL:
268:         err = filp->f_flags;
269:         break;
270:     case F_SETFL:
271:         err = setfl(fd, filp, arg);
272:         break;
```

图 19.15 异步通知机制 F_GETFL/F_SETFL 内部实现

“setfl”函数会比较“filp->f_flags”中的“FASYNC”位，发现它发生了变化时，就会调用驱动程序的 faysnc 函数：

```
65:
66:     /*
67:      * ->fasync() is responsible for setting the FASYNC bit.
68:      */
69:     if (((arg ^ filp->f_flags) & FASYNC) && filp->f_op->fasync) {
70:         error = filp->f_op->fasync(fd, filp, (arg & FASYNC) != 0);
71:         if (error < 0)
72:             goto out;
73:         if (error > 0)
74:             error = 0;
75:     }
```

图 19.16 异步通知机制 F_SETFL 内部实现

驱动程序的 faysnc 函数代码如下：

```
103: static int gpio_key_drv_fasync(int fd, struct file *file, int on)
104: {
105:     if (fasync_helper(fd, file, on, &button_fasync) >= 0)
106:         return 0;
107:     else
108:         return -EIO;
109: }
110:
```

图 19.17 驱动程序中的异步通知代码

它使用 fasync_helper 函数来设置指针 button_fasync，简化后的示例代码如下：

```
if (on)
{
    struct fasync_struct *new;
    new = fasync_alloc();
    new->magic = FASYNC_MAGIC;
    new->fa_file = filp;
    new->fa_fd = fd;
    button_fasync = new;
}
else
{
    kfree(button_fasync);
    button_fasync = NULL;
}
```

所以，启动了 FASYNC 功能的话，驱动程序的 button_fasync 就被设置了，它指向的 fasync_struct 结构体里含有 filp，filp 里含有 PID(接收方的 PID)。

在驱动程序的中断函数里，使用如下代码发出中断：

```
kill_fasync(&button_fasync, SIGIO, POLL_IN);
```

它的核心就是从 button_fasync 指针中，取出 fasync_struct 结构体，从这个结构体的 fa_file 中得到接收方的 PID，然后使用“send_sigio”函数发送信号。

图 19.18 驱动程序中的发信号的过程

“send_sigio”函数的实质是：根据 PID 找到进程在内核的 task_struct 结构体，修改里面的某些成员表示收到了信号。

APP 收到信号后，它的信号处理函数时如何被调用的呢？信号相当于 APP 的中断，处理过程也跟中断的处理过程类似：保存现场、处理信号，恢复现场。

APP 进入内核态时，内核在 APP 的栈里保存“APP 的运行环境”：APP 在用户态进入内核态瞬间各个寄存器的值，包括“运行地址”(即恢复运行时从哪里继续运行)。

APP 退出内核态时，内核会从 APP 的栈里恢复“APP 的运行环境”，比如 APP 将从之前保存的“运行地址”继续运行。

APP 收到信号瞬间，APP 必定处于内核态，因为信号的发送函数“send_sigio”要么由驱动程序调用，要么由 APP 通过系统调用来间接调用，函数“send_sigio”处于内核态。APP 从内核态返回到用户态前，内核发现 APP 有信号在等待处理时，会修改 APP 的栈，增加一个新的“运行环境”：新环境里“运行地址”是信号处

理函数的地址。这样，APP 从内核态返回用户态时，运行的是信号处理函数。信号处理函数执行完后，会再次返回到内核态，在内核态里再使用旧的“运行环境”恢复 APP 的运行。

19.4 阻塞与非阻塞

所谓阻塞，就是等待某件事情发生。比如调用 read 读取按键时，如果没有按键数据则 read 函数不会返回，它会让线程休眠等待。

使用 poll 时，如果传入的超时时间不为 0，这种访问方法也是阻塞的。

使用 poll 时，可以设置超时时间为 0，这样即使没有数据它也会立刻返回，这就是非阻塞方式。能不能让 read 函数既能工作于阻塞方式，也可以工作于非阻塞方式？可以！

APP 调用 open 函数时，传入 O_NONBLOCK，就表示要使用非阻塞方式；默认是阻塞方式。

注意：对于普通文件、块设备文件，O_NONBLOCK 不起作用。

注意：对于字符设备文件，O_NONBLOCK 起作用的前提是驱动程序针对 O_NONBLOCK 做了处理。

只能在 open 时表明 O_NONBLOCK 吗？在 open 之后，也可以通过 fcntl 修改为阻塞或非阻塞。

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\06_read_key_irq_poll_fasync_block
```

19.4.1 应用编程

- open 时设置：

```
int  fd = open(“/dev/xxx”, O_RDWR | O_NONBLOCK);  /* 非阻塞方式 */
int  fd = open(“/dev/xxx”, O_RDWR );  /* 阻塞方式 */
```

- open 之后设置：

```
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);  /* 非阻塞方式 */
fcntl(fd, F_SETFL, flags & ~O_NONBLOCK);  /* 阻塞方式 */
```

19.4.2 驱动编程

- 以 drv_read 为例：

```
static ssize_t drv_read(struct file *fp, char __user *buf, size_t count, loff_t *ppo
s)
 {
if (queue_empty(&as->queue) && fp->f_flags & O_NONBLOCK)
        return -EAGAIN;

    wait_event_interruptible(apm_waitqueue, !queue_empty(&as->queue));
```

```
    ……
}
```

从驱动代码也可以看出来，当 APP 打开某个驱动时，在内核中会有一个 struct file 结构体对应这个驱动，这个结构体中有 f_flags，就是打开文件时的标记位；可以设置 f_flasgs 的 O_NONBLOCK 位，表示非阻塞；也可以清除这个位表示阻塞。

驱动程序要根据这个标记位决定事件未就绪时是休眠和还是立刻返回。

19.4.3 驱动开发原则

驱动程序程序“只提供功能，不提供策略”。就是说驱动程序可以提供休眠唤醒、查询等等各种方式,，驱动程序只提供这些能力，怎么用由 APP 决定。

19.5 定时器

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\07_read_key_irq_poll_fasync_block_timer
```

19.5.1 内核函数

所谓定时器，就是闹钟，时间到后你就要做某些事。有 2 个要素：时间、做事，换成程序员的话就是：超时时间、函数。

在内核中使用定时器很简单，涉及这些函数(参考内核源码 include\linux\timer.h)：

- setup_timer(timer, fn, data)：
  - 设置定时器，主要是初始化 timer_list 结构体，设置其中的函数、参数。
- void add_timer(struct timer_list *timer)：
  - 向内核添加定时器。timer->expires 表示超时时间。
  - 当超时时间到达，内核就会调用这个函数：timer->function(timer->data)。
- int mod_timer(struct timer_list *timer, unsigned long expires)：
  - 修改定时器的超时时间，
  - 它等同于：del_timer(timer); timer->expires = expires; add_timer(timer);
  - 但是更加高效。
- int del_timer(struct timer_list *timer)：
  - 删除定时器。

19.5.2 定时器时间单位

编译内核时，可以在内核源码根目录下用“ls -a”看到一个隐藏文件，它就是内核配置文件。打开后可以看到如下这项：

```
CONFIG_HZ=100
```

这表示内核每秒中会发生 100 次系统滴答中断(tick)，这就像人类的心跳一样，这是 Linux 系统的心跳。每发生一次 tick 中断，全局变量 jiffies 就会累加 1。

CONFIG_HZ=100 表示每个滴答是 10ms。

定时器的时间就是基于 jiffies 的，我们修改超时时间时，一般使用这 2 种方法：

① 在 add_timer 之前，直接修改：

```
timer.expires = jiffies + xxx;   // xxx 表示多少个滴答后超时，也就是 xxx*10ms
timer.expires = jiffies + 2*HZ;  // HZ 等于 CONFIG_HZ，2*HZ 就相当于 2 秒
```

② 在 add_timer 之后，使用 mod_timer 修改：

```
mod_timer(&timer, jiffies + xxx);   // xxx 表示多少个滴答后超时，也就是 xxx*10ms
mod_timer(&timer, jiffies + 2*HZ);  // HZ 等于 CONFIG_HZ，2*HZ 就相当于 2 秒
```

19.5.3 使用定时器处理按键抖动

在实际的按键操作中，可能会有机械抖动：

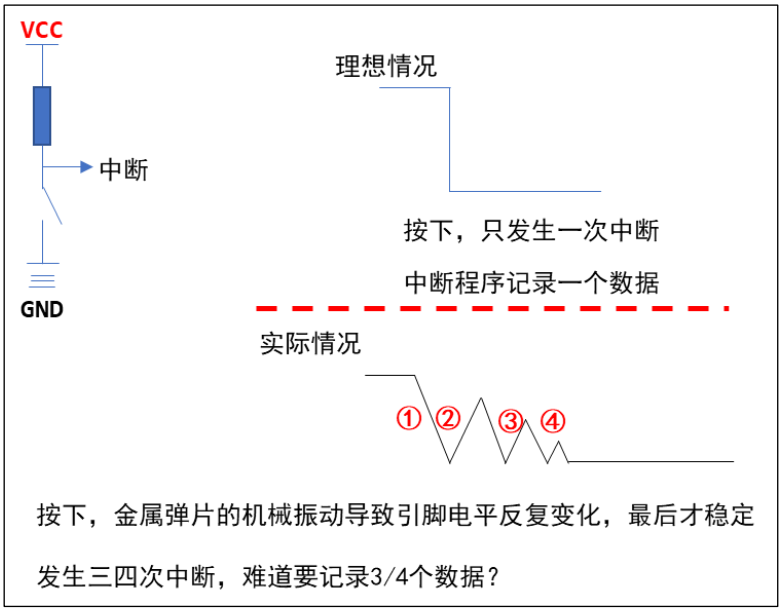


图 19.19 按键震动

按下或松开一个按键，它的 GPIO 电平会反复变化，最后才稳定。一般是几十毫秒才会稳定。

如果不处理抖动的话，用户只操作一次按键，中断程序可能会上报多个数据。怎么处理？

- 在按键中断程序中，可以循环判断几十毫秒，发现电平稳定之后再上报
- 使用定时器

显然第 1 种方法太耗时，违背“中断要尽快处理”的原则，你的系统会很卡。怎么使用定时器？看下图：

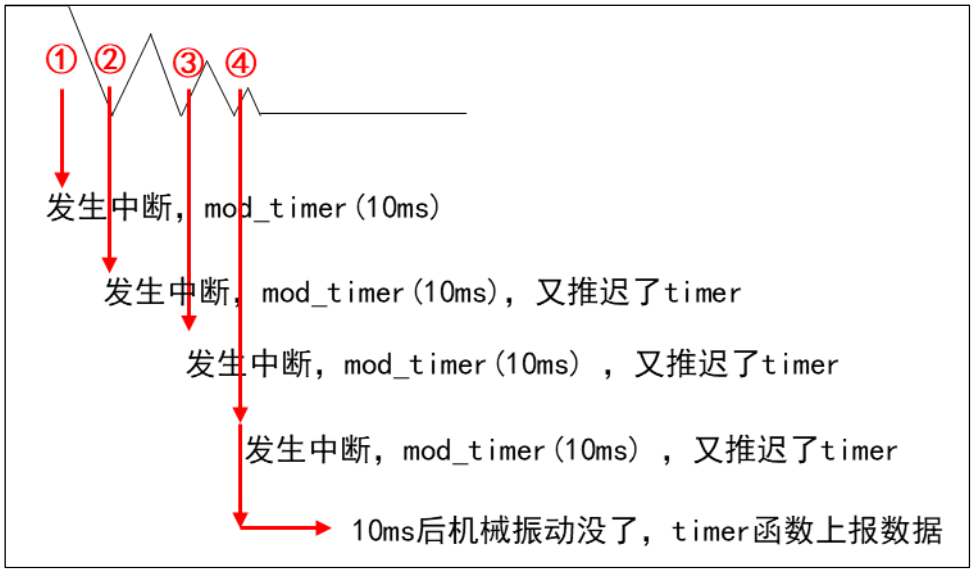


图 19.20 震动触发定时器

核心在于：在 GPIO 中断中并不立刻记录按键值，而是修改定时器超时时间，10ms 后再处理。

- 如果 10ms 内又发生了 GPIO 中断，那就认为是抖动，这时再次修改超时时间为 10ms。
- 只有 10ms 之内再无 GPIO 中断发生，那么定时器的函数才会被调用。在定时器函数中记录按键值。

19.5.4 现场编程

驱动程序、应用程序的核心都在上面的章节列了出来，怎么写程序？观看视频体验更佳，更能抓住重点。

19.5.5 深入研究：定时器的内部机制

初学者会用定时器就行，本节不用看。

怎么实现定时器，逻辑上很简单：每发生一次硬件中断时，硬件中断处理完后就会看看有没有软件中断要处理。

定时器就是通过软件中断来实现的，它属于 TIMER_SOFTIRQ 软中断。

对于 TIMER_SOFTIRQ 软中断，初始化代码如下：

```
void __init init_timers(void)
{
    init_timer_cpus();
    init_timer_stats();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

当发生硬件中断时，硬件中断处理完后，内核会调用软件中断的处理函数。对于 TIMER_SOFTIRQ，会调用 run_timer_softirq，它的函数如下：

```
run_timer_softirq
  __run_timers(base);
    while (time_after_eq(jiffies, base->clk)) {
        ……
expire_timers(base, heads + levels);
    fn = timer->function;
    data = timer->data;
    call_timer_fn(timer, fn, data);
        fn(data);
```

简单地说，add_timer 函数会把 timer 放入内核里某个链表；

在 TIMER_SOFTIRQ 的处理函数中，会从链表中把这些超时的 timer 取出来，执行其中的函数。

怎么判断是否超时？jiffies 大于或等于 timer->expires 时，timer 就超时。

内核中有很多 timer，如果高效地找到超时的 timer？这是比较复杂的，可以看看这文章：

https://blog.csdn.net/tianmohust/article/details/8707162

我们以后如果要深入讲解 timer 的话，会用视频来讲解。

19.5.6 深入研究：找到系统滴答

在开发板执行以下命令，可以看到 CPU0 下有一个数值变化特别快，它就是滴答中断：

```
# cat /proc/interrupts
           CPU0
 16:       2532      GPC  55 Level     i.MX Timer Tick
 19:         22      GPC  33 Level     2010000.ecspi
 20:        384      GPC  26 Level     2020000.serial
 21:          0      GPC  98 Level     sai
```

- 以 100ASK_IMX6ULL 为做，滴答中断名字就是“i.MX Timer Tick”。在 Linux 内核源码目录下执行以下命令：

```
$ grep "i.MX Timer Tick" * -nr
drivers/clocksource/timer-imx-gpt.c:319:        act->name = "i.MX Timer Tick";
```

- 打开 timer-imx-gpt.c 319 行左右，可得如下源码：

```
act->name = "i.MX Timer Tick";
act->flags = IRQF_TIMER | IRQF_IRQPOLL;
act->handler = mxc_timer_interrupt;
act->dev_id = ced;

return setup_irq(imxtm->irq, act);
```

- mxc_timer_interrupt 应该就是滴答中断的处理函数，代码如下：

```
static irqreturn_t mxc_timer_interrupt(int irq, void *dev_id)
{
    struct clock_event_device *ced = dev_id;
    struct imx_timer *imxtm = to_imx_timer(ced);
    uint32_t tstat;

    tstat = readl_relaxed(imxtm->base + imxtm->gpt->reg_tstat);
```

```
    imxtm->gpt->gpt_irq_acknowledge(imxtm);

    ced->event_handler(ced);

    return IRQ_HANDLED;
}
```

在上述代码中没看到对 jiffies 的累加操作啊，应该是在 ced->event_handler(ced)中进行。

ced->event_handler(ced)是哪一个函数？不太好找，我使用 QEMU 来调试内核，在 mxc_timer_interrupt 中打断点跟踪代码(以后的课程会讲怎么用 QEMU 调试内核)，发现它对应 tick_handle_periodic。

tick_handle_periodic 位于 kernel\time\tick-common.c 中，它里面的调用关系如下：

```
tick_handle_periodic
  tick_periodic(cpu);
    do_timer(1);
       jiffies_64 += ticks;  // jiffies 就是 jiffies_64
```

你为何说 jiffies 就是 jiffies_64？在 arch\arm\kernel\vmlinux.lds.S 有如下代码：

```
#ifndef __ARMEB__
jiffies = jiffies_64;
#else
jiffies = jiffies_64 + 4;
#endif
```

上述代码说明了，对于大字节序的 CPU，jiffies 指向 jiffies_64 的高 4 字节；对于小字节序的 CPU，jiffies 指向 jiffies_64 的低 4 字节。

对 jiffies_64 的累加操作，就是对 jiffies 的累加操作。

19.6 中断下半部 tasklet

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\08_read_key_irq_poll_fasync_block_timer_tasklet
```

在前面我们介绍过中断上半部、下半部。中断的处理有几个原则：

- 不能嵌套；
- 越快越好。

在处理当前中断时，即使发生了其他中断，其他中断也不会得到处理，所以中断的处理要越快越好。但是某些中断要做的事情稍微耗时，这时可以把中断拆分为上半部、下半部。

在上半部处理紧急的事情，在上半部的处理过程中，中断是被禁止的；

在下半部处理耗时的事情，在下半部的处理过程中，中断是使能的。

中断上半部、下半部的关系机制，请回顾第《>>>>>>>>第五篇 18.2.5 下半部要做的事情耗时不是太长：tasklet》。

19.6.1 内核函数

1 定义 tasklet

中断下半部使用结构体 tasklet_struct 来表示，它在内核源码 include\linux\interrupt.h 中定义：

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

- 其中的 state 有 2 位：
  - bit0 表示 TASKLET_STATE_SCHED

    等于 1 时表示已经执行了 tasklet_schedule 把该 tasklet 放入队列了；tasklet_schedule 会判断该位，如果已经等于 1 那么它就不会再次把 tasklet 放入队列。
  - bit1 表示 TASKLET_STATE_RUN

    等于 1 时，表示正在运行 tasklet 中的 func 函数；函数执行完后内核会把该位清 0。
- 其中的 count 表示该 tasklet 是否使能：等于 0 表示使能了，非 0 表示被禁止了。对于 count 非 0 的 tasklet，里面的 func 函数不会被执行。

使用中断下半部之前，要先实现一个 tasklet_struct 结构体，这可以用这 2 个宏来定义结构体：

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

- 使用 DECLARE_TASKLET 定义的 tasklet 结构体，它是使能的；
- 使用 DECLARE_TASKLET_DISABLED 定义的 tasklet 结构体，它是禁止的；使用之前要先调用 tasklet_enable 使能它。

也可以使用函数来初始化 tasklet 结构体：

```
extern void tasklet_init(struct tasklet_struct *t,
            void (*func)(unsigned long), unsigned long data);
```

2 使能/禁止 tasklet

```
static inline void tasklet_enable(struct tasklet_struct *t);
static inline void tasklet_disable(struct tasklet_struct *t);
```

- tasklet_enable 把 count 增加 1；
- tasklet_disable 把 count 减 1。

3 调度 tasklet

```
static inline void tasklet_schedule(struct tasklet_struct *t);
```

- 把 tasklet 放入链表，并且设置它的 TASKLET_STATE_SCHED 状态为 1。

4 kill tasklet

```
extern void tasklet_kill(struct tasklet_struct *t);
```

- 如果一个 tasklet 未被调度，tasklet_kill 会把它的 TASKLET_STATE_SCHED 状态清 0；
- 如果一个 tasklet 已被调度，tasklet_kill 会等待它执行完华，再把它的 TASKLET_STATE_SCHED 状态清 0。

通常在卸载驱动程序时调用 tasklet_kill。

19.6.2 tasklet 使用方法

先定义 tasklet，需要使用时调用 tasklet_schedule，驱动卸载前调用 tasklet_kill。

tasklet_schedule 只是把 tasklet 放入内核队列，它的 func 函数会在软件中断的执行过程中被调用。

19.6.3 tasklet 内部机制

tasklet 属于 TASKLET_SOFTIRQ 软件中断，入口函数为 tasklet_action，这在内核 kernel\softirq.c 中设置：

```
void __init softirq_init(void)
{
    int cpu;

    for_each_possible_cpu(cpu) {
        per_cpu(tasklet_vec, cpu).tail =
            &per_cpu(tasklet_vec, cpu).head;
        per_cpu(tasklet_hi_vec, cpu).tail =
            &per_cpu(tasklet_hi_vec, cpu).head;
    }

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

图 19.21 softirq.c

当驱动程序调用 tasklet_schedule 时，会设置 tasklet 的 state 为 TASKLET_STATE_SCHED，并把它放入某个链表：

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) // 1. 如果未设置为SCHED
        __tasklet_schedule(t);                              设置为SCHED并放入队列
}

void __tasklet_schedule(struct tasklet_struct *t)
{
    unsigned long flags;

    local_irq_save(flags);  2. 放入队列
    t->next = NULL;
    *__this_cpu_read(tasklet_vec.tail) = t;
    __this_cpu_write(tasklet_vec.tail, &(t->next));
    raise_softirq_irqoff(TASKLET_SOFTIRQ);
    local_irq_restore(flags);  3. 出发TASKLET软中断
}
```

图 19.22 放入队列

当发生硬件中断时，内核处理完硬件中断后，会处理软件中断。对于 TASKLET_SOFTIRQ 软件中断，会调用 tasklet_action 函数。

执行过程还是挺简单的：从队列中找到 tasklet，进行状态判断后执行 func 函数，从队列中删除 tasklet。

从这里可以看出：

① tasklet_schedule 调度 tasklet 时，其中的函数并不会立刻执行，而只是把 tasklet 放入队列；

② 调用一次 tasklet_schedule，只会导致 tasklnet 的函数被执行一次；

③ 如果 tasklet 的函数尚未执行，多次调用 tasklet_schedule 也是无效的，只会放入队列一次。

tasklet_action 函数解析如下：

```
static __latent_entropy void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();

    while (list) {
        struct tasklet_struct *t = list;

        list = list->next;

        if (tasklet_trylock(t)) {
            if (!atomic_read(&t->count)) {
                if (!test_and_clear_bit(TASKLET_STATE_SCHED,
                                        &t->state))
                    BUG();
                t->func(t->data);
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }

        local_irq_disable();
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        __raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_enable();
    } « end while list »
} « end tasklet_action »
```

1. 从列表中去出每一项
2. 判断 如果不是SCHED状态，就是有BUG
3. 执行
4. 从队列中取出

图 19.23 tasklet_action

19.7 工作队列

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\
source\06_gpio_irq\09_read_key_irq_poll_fasync_block_timer_tasklet_workqueue
```

前面讲的定时器、下半部 tasklet，它们都是在中断上下文中执行，它们无法休眠。当要处理更复杂的事情时，往往更耗时。这些更耗时的工作放在定时器或是下半部中，会使得系统很卡；并且循环等待某件事情完成也太浪费 CPU 资源了。

如果使用线程来处理这些耗时的工作，那就可以解决系统卡顿的问题：因为线程可以休眠。

在内核中，我们并不需要自己去创建线程，可以使用“工作队列”(workqueue)。内核初始化工作队列是，就为它创建了内核线程。以后我们要使用“工作队列”，只需要把“工作”放入“工作队列中”，对应的内核线程就会取出“工作”，执行里面的函数。

在 2.xx 的内核中，工作队列的内部机制比较简单；在现在 4.x 的内核中，工作队列的内部机制做得复杂无比，但是用法是一样的。

工作队列的应用场合：要做的事情比较耗时，甚至可能需要休眠，那么可以使用工作队列。

缺点：多个工作(函数)是在某个内核线程中依序执行的，前面函数执行很慢，就会影响到后面的函数。

在多 CPU 的系统下，一个工作队列可以有多个内核线程，可以在一定程度上缓解这个问题。

我们先使用看看怎么使用工作队列。

19.7.1 内核函数

内核线程、工作队列(workqueue)都由内核创建了，我们只是使用。使用的核心是一个 work_struct 结构体，定义如下：

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};

typedef void (*work_func_t)(struct work_struct *work);
```

图 19.24 work_struct

使用工作队列时，步骤如下：

第1步 构造一个 work_struct 结构体，里面有函数；

第2步 把这个 work_struct 结构体放入工作队列，内核线程就会运行 work 中的函数。

1 定义 work

参考内核头文件：include\linux\workqueue.h

```
#define DECLARE_WORK(n, f)                      \
    struct work_struct n = __WORK_INITIALIZER(n, f)

#define DECLARE_DELAYED_WORK(n, f)                  \
    struct delayed_work n = __DELAYED_WORK_INITIALIZER(n, f, 0)
```

- 第 1 个宏是用来定义一个 work_struct 结构体，要指定它的函数。
- 第 2 个宏用来定义一个 delayed_work 结构体，也要指定它的函数。所以“delayed”，意思就是说要让它运行时，可以指定：某段时间之后你再执行。

如果要在代码中初始化 work_struct 结构体，可以使用下面的宏：

```
#define INIT_WORK(_work, _func)
```

2 使用 work：schedule_work

调用 schedule_work 时，就会把 work_struct 结构体放入队列中，并唤醒对应的内核线程。内核线程就会从队列里把 work_struct 结构体取出来，执行里面的函数。

3 其他函数

表 19-2 workqueue 其它函数

序号	函数	说明
1	create_workqueue	在 Linux 系统中已经有了现成的 system_wq 等工作队列， 你当然也可以自己调用 create_workqueue 创建工作队列， 对于 SMP 系统，这个工作队列会有多个内核线程与它对应， 创建工作队列时，内核会帮这个工作队列创建多个内核线程
2	create_singlethread_workqueue	如果想只有一个内核线程与工作队列对应， 可以用本函数创建工作队列， 创建工作队列时，内核会帮这个工作队列创建一个内核线程
3	destroy_workqueue	销毁工作队列
4	schedule_work	调度执行一个具体的 work， 执行的 work 将会被挂入 Linux 系统提供的工作队列
5	schedule_delayed_work	延迟一定时间去执行一个具体的任务， 功能与 schedule_work 类似，多了一个延迟时间
6	queue_work	跟 schedule_work 类似， schedule_work 是在系统默认的工作队列上执行一个 work， queue_work 需要自己指定工作队列
7	queue_delayed_work	跟 schedule_delayed_work 类似， schedule_delayed_work 是在系统默认的工作队列上执行一个 work， queue_delayed_work 需要自己指定工作队列
8	flush_work	等待一个 work 执行完毕， 如果这个 work 已经被放入队列，那么本函数等它执行完毕，并且返回 true； 如果这个 work 已经执行完华才调用本函数，那么直接返回 false
9	flush_delayed_work	等待一个 delayed_work 执行完毕， 如果这个 delayed_work 已经被放入队列，那么本函数等它执行完毕，并且返回 true； 如果这个 delayed_work 已经执行完华才调用本函数，那么直接返回 false

19.7.2 编程、上机

驱动程序、应用程序的核心都在上面的章节列了出来，怎么写程序？观看视频体验更佳，更能抓住重点。

19.7.3 内部机制

初学者知道 work_struct 中的函数是运行于内核线程的上下文，这就足够了。

在 2.xx 版本的 Linux 内核中，创建 workqueue 时就会同时创建内核线程；

在 4.xx 版本的 Linux 内核中，内核线程和 workqueue 是分开创建的，比较复杂。

1 Linux 2.x 的工作队列创建过程

代码在 kernel\workqueue.c 中：

```
init_workqueues
keventd_wq = create_workqueue("events");
    __create_workqueue((name), 0, 0)
        for_each_possible_cpu(cpu) {
            err = create_workqueue_thread(cwq, cpu);
                p = kthread_create(worker_thread, cwq, fmt, wq->name, cpu);
```

对于每一个 CPU，都创建一个名为“events/X”的内核线程，X 从 0 开始。在创建 workqueue 的同时创建内核线程。

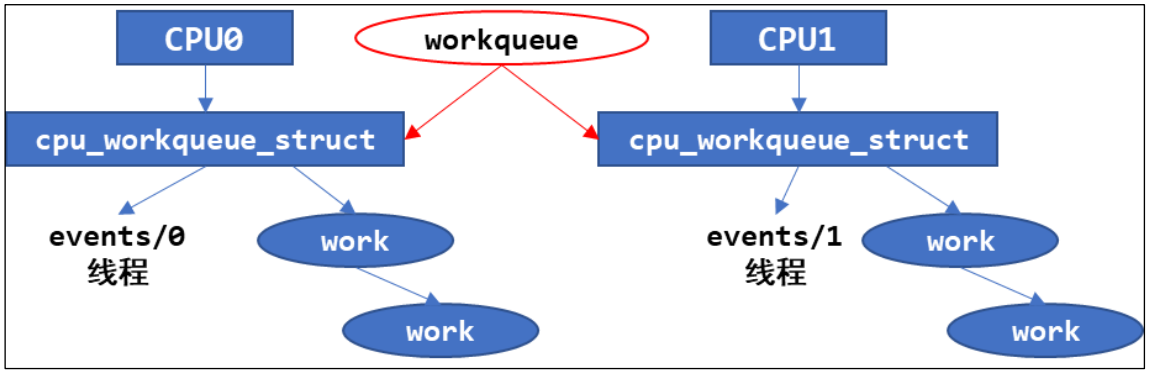


图 19.25 Linux 2.x 的工作队列

2 Linux 4.x 的工作队列创建过程

Linux4.x 中，内核线程和工作队列是分开创建的。先创建内核线程，代码在 kernel\workqueue.c 中：

```
init_workqueues
/* initialize CPU pools */
for_each_possible_cpu(cpu) {
    for_each_cpu_worker_pool(pool, cpu) {
        /* 对每一个 CPU 都创建 2 个 worker_pool 结构体，它是含有 ID 的 */
        /*  一个 worker_pool 对应普通优先级的 work，第 2 个对应高优先级的 work */
}

/* create the initial worker */
for_each_online_cpu(cpu) {
    for_each_cpu_worker_pool(pool, cpu) {
        /* 对每一个 CPU 的每一个 worker_pool，创建一个 worker */
/* 每一个 worker 对应一个内核线程 */
        BUG_ON(!create_worker(pool));
    }
}
```

create_worker 函数代码如下：

```
static struct worker *create_worker(struct worker_pool *pool)
{
    struct worker *worker = NULL;
    int id = -1;
    char id_buf[16];

    /* ID is needed to determine kthread name */
    id = ida_simple_get(&pool->worker_ida, 0, 0, GFP_KERNEL);
    if (id < 0)
        goto fail;

    worker = alloc_worker(pool->node);
    if (!worker)
        goto fail;

    worker->pool = pool;
    worker->id = id;
                                在哪个CPU上运行        poll中第几个线程
    if (pool->cpu >= 0)
        snprintf(id_buf, sizeof(id_buf), "%d:%d%s", pool->cpu, id,
             pool->attrs->nice < 0  ? "H" : "");  H: 高优先级
    else
        snprintf(id_buf, sizeof(id_buf), "u%d:%d", pool->id, id);

    worker->task = kthread_create_on_node(worker_thread, worker, pool->node,
                                          "kworker/%s", id_buf);内核线程的名字
```

图 19.26 create_worker 函数

创建好内核线程后，再创建 workqueue，代码在 kernel\workqueue.c 中：

```
init_workqueues
system_wq = alloc_workqueue("events", 0, 0);
    __alloc_workqueue_key
        wq = kzalloc(sizeof(*wq) + tbl_size, GFP_KERNEL);  // 分配 workqueue_struct
        alloc_and_link_pwqs(wq) // 跟 worker_poll 建立联系
```

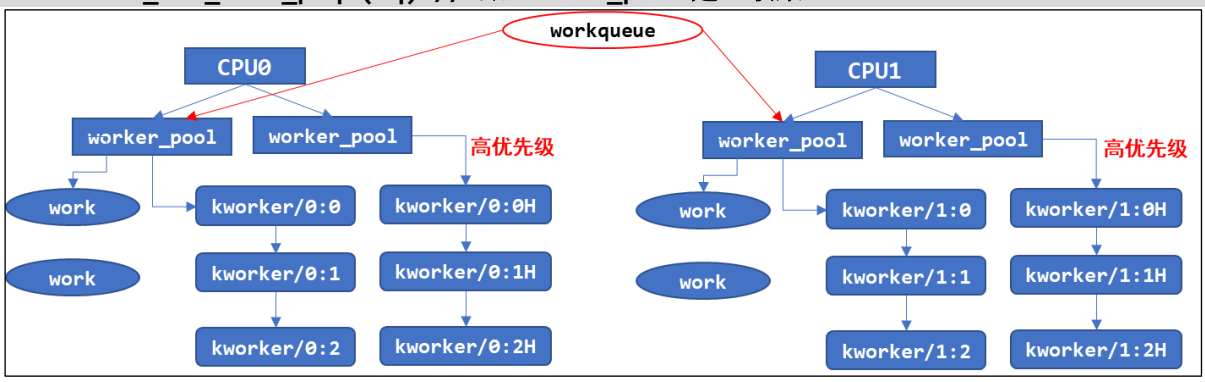


图 19.27 Linux 4.x 的工作队列

一开始时，每一个 worker_poll 下只有一个线程，但是系统会根据任务繁重程度动态创建、销毁内核线程。所以你可以在 work 中打印线程 ID，发现它可能是变化的。

参考文章：

https://zhuanlan.zhihu.com/p/91106844

https://www.cnblogs.com/vedic/p/11069249.html

https://www.cnblogs.com/zxc2man/p/4678075.html

19.8 中断的线程化处理

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
```

05_嵌入式 Linux 驱动开发基础知识\source\06_gpio_irq\10_read_key_irq_poll_fasync_block_timer_tasklet_workqueue_threadedirq

请先回顾《>>>>>>>>第五篇 18.2.7 新技术：threaded irq》。

复杂、耗时的事情，尽量使用内核线程来处理。上节视频介绍的工作队列用起来挺简单，但是它有一个缺点：工作队列中有多个 work，前一个 work 没处理完会影响后面的 work。解决方法有很多种，比如干脆自己创建一个内核线程，不跟别的 work 凑在一块了。在 Linux 系统中，对于存储设备比如 SD/TF 卡，它的驱动程序就是这样做的，它有自己的内核线程。

对于中断处理，还有另一种方法：threaded irq，线程化的中断处理。中断的处理仍然可以认为分为上半部、下半部。上半部用来处理紧急的事情，下半部用一个内核线程来处理，这个内核线程专用于这个中断。

内核提供了这个函数：

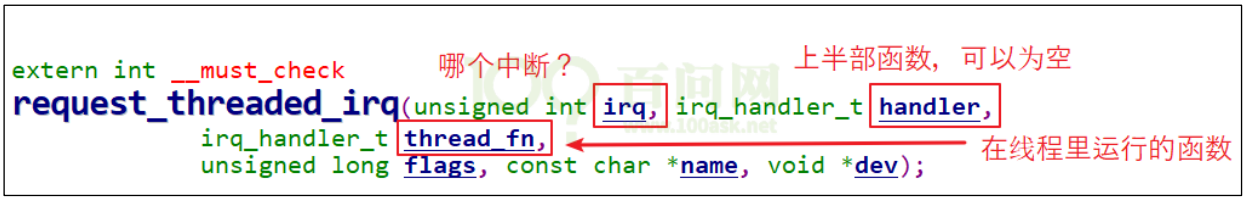


图 19.28 内核的中断线程

你可以只提供 thread_fn，系统会为这个函数创建一个内核线程。发生中断时，系统会立刻调用 handler 函数，然后唤醒某个内核线程，内核线程再来执行 thread_fn 函数。

19.8.1 内核机制

1 调用 request_threaded_irq 后内核的数据结构、

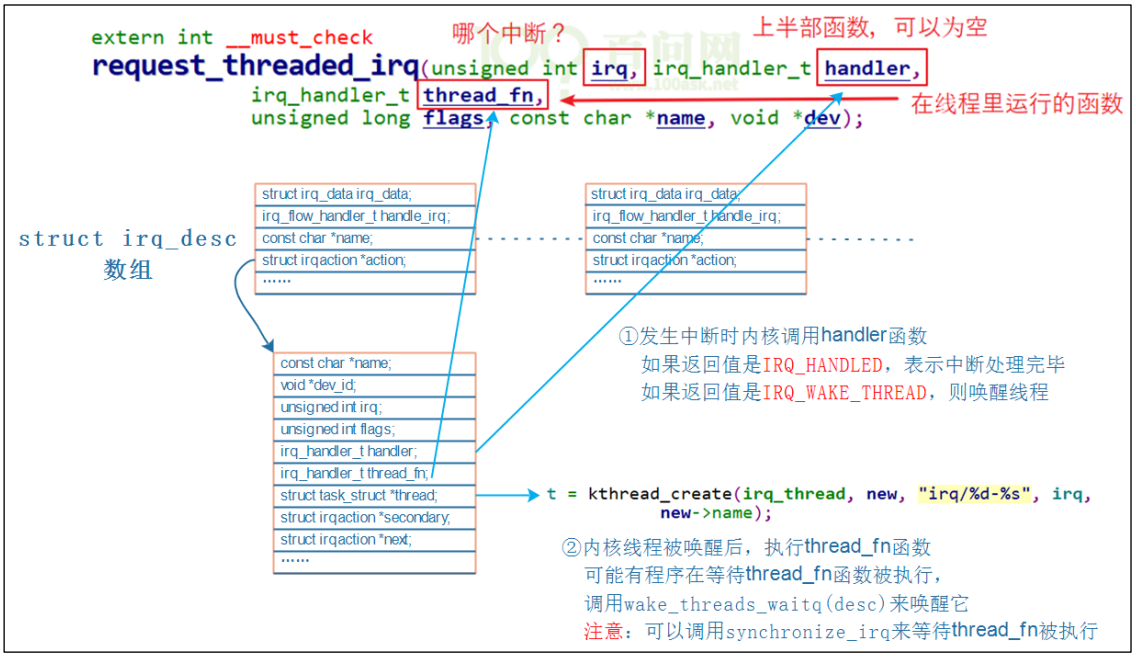


图 19.29 调用 request_threaded_irq 后

2 request_threaded_irq

request_threaded_irq 函数，肯定会创建一个内核线程。

源码在内核文件 kernel\irq\manage.c 中，

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
            irq_handler_t thread_fn, unsigned long irqflags,
            const char *devname, void *dev_id)
{
    // 分配、设置一个 irqaction 结构体
    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->thread_fn = thread_fn;
    action->flags = irqflags;
    action->name = devname;
    action->dev_id = dev_id;

    retval = __setup_irq(irq, desc, action);  // 进一步处理
}
```

- __setup_irq 函数代码如下(只摘取重要部分)：

```
    if (new->thread_fn && !nested) {
        ret = setup_irq_thread(new, irq, false);
```

- setup_irq_thread 函数代码如下(只摘取重要部分)：

```
    if (!secondary) {
        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                   new->name);
    } else {
        t = kthread_create(irq_thread, new, "irq/%d-s-%s", irq,
                   new->name);
        param.sched_priority -= 1;
    }
new->thread = t;
```

3 中断的执行过程

对于 GPIO 中断，我使用 QEMU 的调试功能找出了所涉及的函数调用，其他板子可能稍有不同。

调用关系如下，反过来看：

```
Breakpoint 1, gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keybo
ard/gpio_keys.c:393
393 {
(gdb) bt
#0  gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keyboard/gpio_k
eys.c:393
#1  0x80270528 in __handle_irq_event_percpu (desc=0x8616e300, flags=0x86517edc) at ke
rnel/irq/handle.c:145
#2  0x802705cc in handle_irq_event_percpu (desc=0x8616e300) at kernel/irq/handle.c:18
5
#3  0x80270640 in handle_irq_event (desc=0x8616e300) at kernel/irq/handle.c:202
#4  0x802738e8 in handle_level_irq (desc=0x8616e300) at kernel/irq/chip.c:518
#5  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/i
rqdesc.h:150
```

```
#6  generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#7  0x805005e0 in mxc_gpio_irq_handler (port=0xc8, irq_stat=2252237104) at drivers/gpio/gpio-mxc.c:274
#8  0x805006fc in mx3_gpio_irq_handler (desc=<optimized out>) at drivers/gpio/gpio-mxc.c:291
#9  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/irqdesc.h:150
#10 generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#11 0x8026fd0c in __handle_domain_irq (domain=0x86006000, hwirq=32, lookup=true, regs=0x86517fb0) at kernel/irq/irqdesc.c:627
#12 0x80201484 in handle_domain_irq (regs=<optimized out>, hwirq=<optimized out>, domain=<optimized out>) at ./include/linux/irqdesc.h:168
#13 gic_handle_irq (regs=0xc8) at drivers/irqchip/irq-gic.c:364
#14 0x8020b704 in __irq_usr () at arch/arm/kernel/entry-armv.S:464
```

我们只需要分析__handle_irq_event_percpu函数，它在kernel\irq\handle.c中：

```
irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, unsigned int *flags)
{
    irqreturn_t retval = IRQ_NONE;
    unsigned int irq = desc->irq_data.irq;
    struct irqaction *action;

    for_each_action_of_desc(desc, action) {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        res = action->handler(irq, action->dev_id);
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
                      irq, action->handler))
            local_irq_disable();

        switch (res) {
        case IRQ_WAKE_THREAD:
            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }

            __irq_wake_thread(desc, action);
```

1.调用上半部处理函数

2. 如果返回值是IRQ_WAKE_THREAD

就唤醒中断对应的内核线程

线程的处理函数为irq_thread，代码在kernel\irq\handle.c中：

```
while (!irq_wait_for_interrupt(action)) { // 1.休眠等待中断
    irqreturn_t action_ret;

    irq_thread_check_affinity(desc, action);

    action_ret = handler_fn(desc, action); // 2.执行thread_fn
    if (action_ret == IRQ_HANDLED)
        atomic_inc(&desc->threads_handled);
    if (action_ret == IRQ_WAKE_THREAD)
        irq_wake_secondary(desc, action);

    wake_threads_waitq(desc); // 3. 唤醒等待thread_fn的线程
}
```

19.8.2 编程、上机

调用 request_threaded_irq 函数注册中断，调用 free_irq 卸载中断。

从前面可知，我们可以提供上半部函数，也可以不提供：

- 如果不提供
  - 内核会提供默认的上半部处理函数：irq_default_primary_handler，它是直接返回 IRQ_WAKE_THREAD。
- 如果提供的话
  - 返回值必须是：IRQ_WAKE_THREAD。

在 thread_fn 中，如果中断被正确处理了，应该返回 IRQ_HANDLED。

19.9 mmap

应用程序和驱动程序之间传递数据时，可以通过 read、write 函数进行。这涉及在用户态 buffer 和内核态 buffer 之间传数据，如下图所示：

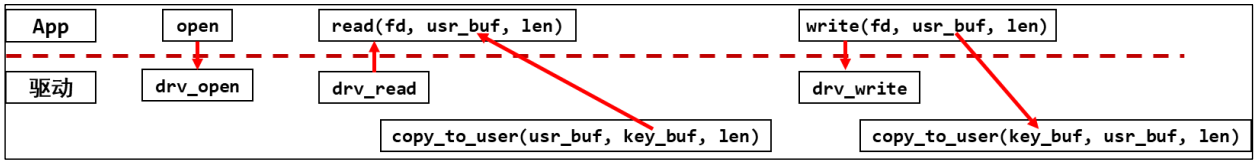


图 19.30 mmap

应用程序不能直接读写驱动程序中的 buffer，需要在用户态 buffer 和内核态 buffer 之间进行一次数据拷贝。这种方式在数据量比较小时没什么问题；但是数据量比较大时效率就太低了。比如更新 LCD 显示时，如果每次都让 APP 传递一帧数据给内核，假设 LCD 采用 1024*600*32bpp 的格式，一帧数据就有 1024*600*32/8=2.3MB 左右，这无法忍受。

改进的方法就是让程序可以直接读写驱动程序中的 buffer，这可以通过 mmap 实现(memory map)，把内核的 buffer 映射到用户态，让 APP 在用户态直接读写。

19.9.1 内存映射现象与数据结构

假设有这样的程序，名为 test.c：

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

int a;
int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage: %s <number>\n", argv[0]);
        return -1;
    }
    a = strtol(argv[1], NULL, 0);
```

```
    printf("a's address = 0x%lx, a's value = %d\n", &a, a);
    while (1)
    {
        sleep(10);
    }
    return 0;
}
```

在 PC 上如下编译(必须静态编译)：

```
gcc  -o  test  test.c  -staitc
```

分别执行 test 程序 2 次，最后执行 ps，可以看到这 2 个程序同时存在，这 2 个程序里 a 变量的地址相同，但是值不同。如下图：

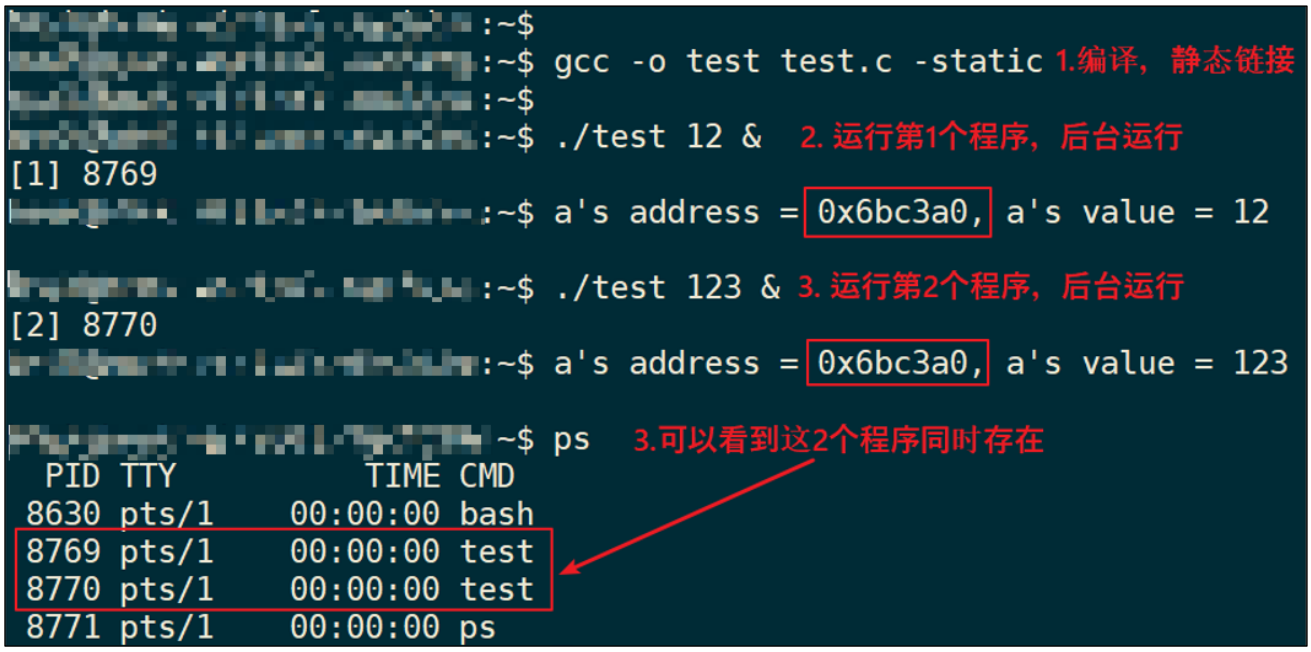


图 19.31 执行两次 test

观察到这些现象：

- 2 个程序同时运行，它们的变量 a 的地址都是一样的：0x6bc3a0；
- 2 个程序同时运行，它们的变量 a 的值是不一样的，一个是 12，另一个是 123。

疑问来了：

- 这 2 个程序同时在内存中运行，它们的值不一样，所以变量 a 的地址肯定不同；
- 但是打印出来的变量 a 的地址却是一样的。

怎么回事？

这里要引入虚拟地址的概念：CPU 发出的地址是虚拟地址，它经过 MMU(Memory Manage Unit，内存管理单元)映射到物理地址上，对于不同进程的同一个虚拟地址，MMU 会把它们映射到不同的物理地址。如下图：

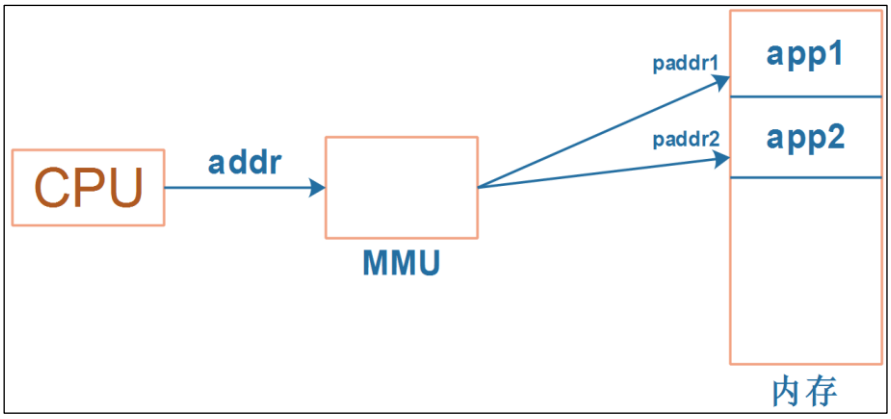
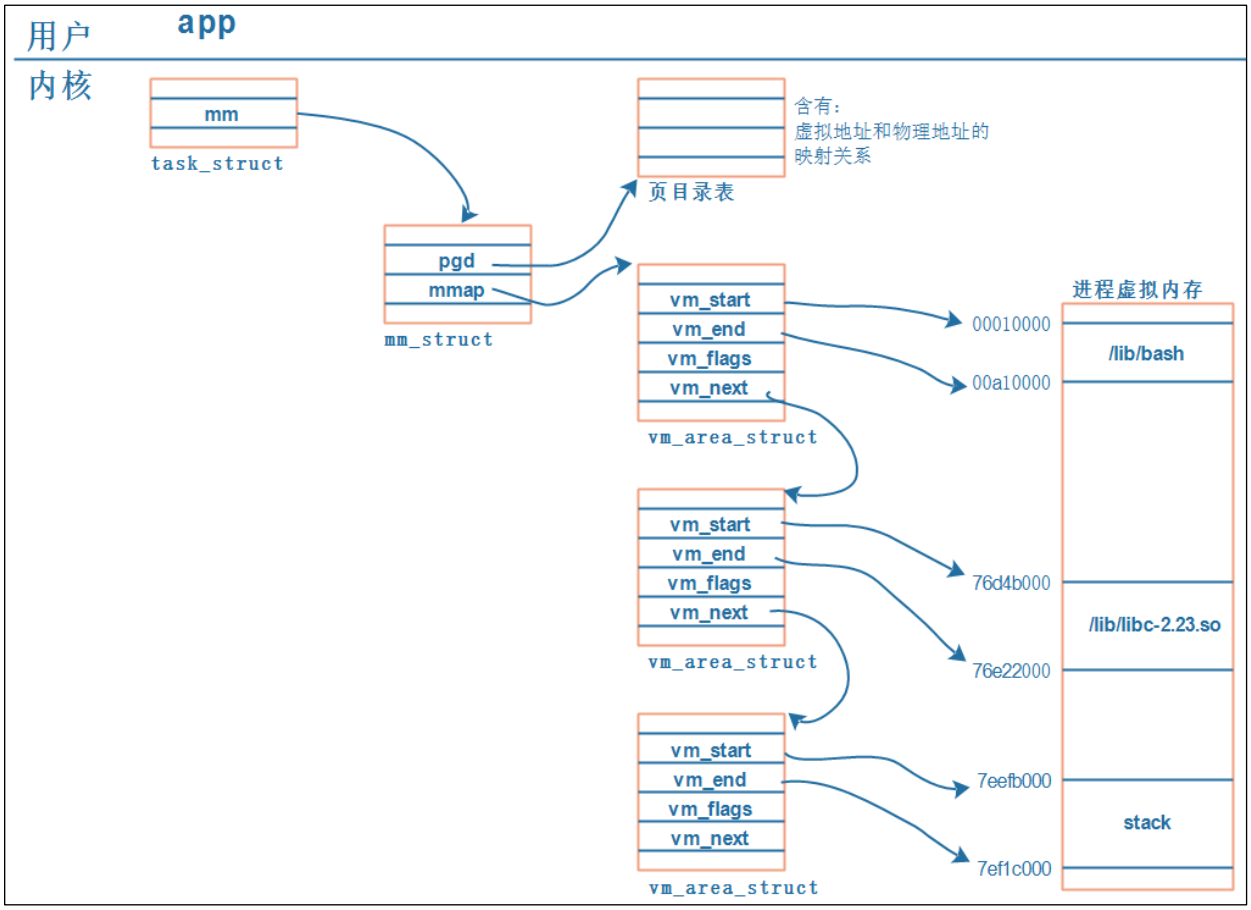


图 19.32 MMU

- 当前运行的是 app1 时，MMU 会把 CPU 发出的虚拟地址 addr 映射为物理地址 paddr1，用 paddr1 去访问内存。
- 当前运行的是 app2 时，MMU 会把 CPU 发出的虚拟地址 addr 映射为物理地址 paddr2，用 paddr2 去访问内存。
- MMU 负责把虚拟地址映射为物理地址，虚拟地址映射到哪个物理地址去？

可以执行 ps 命令查看进程 ID，然后执行“cat /proc/325/maps”得到映射关系。

每一个 APP 在内核里都有一个 tast_struct，这个结构体中保存有内存信息：mm_struct。而虚拟地址、物理地址的映射关系保存在页目录表中，如下图所示：



解析如下：

- 每个 APP 在内核中都有一个 task_struct 结构体，它用来描述一个进程；
- 每个 APP 都要占据内存，在 task_struct 中用 mm_struct 来管理进程占用的内存：
  - 内存有虚拟地址、物理地址，mm_struct 中用 mmap 来描述虚拟地址，用 pgd 来描述对应的物理地址。
  - 注意：pgd，Page Global Directory，页目录。
- 每个 APP 都有一系列的 VMA：virtual memory
  - 比如 APP 含有代码段、数据段、BSS 段、栈等等，还有共享库。这些单元会保存在内存里，它们的地址空间不同，权限不同(代码段是只读的可运行的、数据段可读可写)，内核用一系列的 vm_area_struct 来描述它们。
  - vm_area_struct 中的 vm_start、vm_end 是虚拟地址。
- vm_area_struct 中虚拟地址如何映射到物理地址去？
  - 每一个 APP 的虚拟地址可能相同，物理地址不相同，这些对应关系保存在 pgd 中。

19.9.2 ARM 架构内存映射简介

ARM 架构支持一级页表映射，也就是说 MMU 根据 CPU 发来的虚拟地址可以找到第 1 个页表，从第 1 个页表里就可以知道这个虚拟地址对应的物理地址。一级页表里地址映射的最小单位是 1M。

ARM 架构还支持二级页表映射，也就是说 MMU 根据 CPU 发来的虚拟地址先找到第 1 个页表，从第 1 个页表里就可以知道第 2 级页表在哪里；再取出第 2 级页表，从第 2 个页表里才能确定这个虚拟地址对应的物理地址。二级页表地址映射的最小单位有 4K、1K，Linux 使用 4K。

一级页表项里的内容，决定了它是指向一块物理内存，还是指向二级页表，如下图：

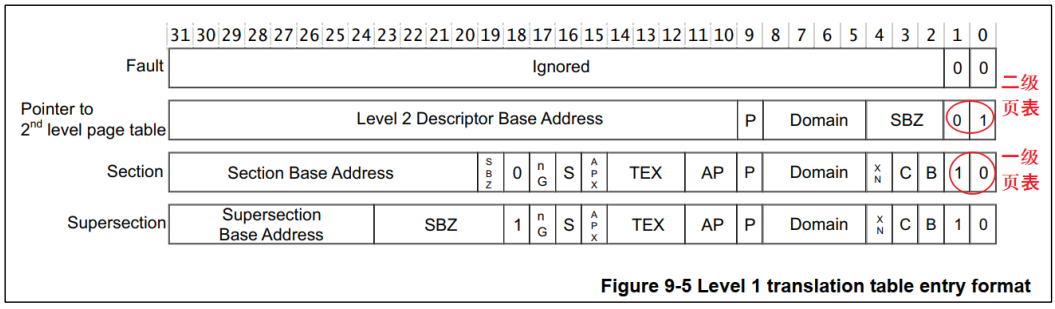


Figure 9-5 Level 1 translation table entry format

1 一级页表映射过程

一线页表中每一个表项用来设置 1M 的空间，对于 32 位的系统，虚拟地址空间有 4G，4G/1M=4096。所以一级页表要映射整个 4G 空间的话，需要 4096 个页表项。

第 0 个页表项用来表示虚拟地址第 0 个 1M(虚拟地址为 0～0xFFFFF)对应哪一块物理内存，并且有一些权限设置；

第 1 个页表项用来表示虚拟地址第 1 个 1M(虚拟地址为 0x100000～0x1FFFFF)对应哪一块物理内存，并且有一些权限设置；

依次类推。

使用一级页表时，先在内存里设置好各个页表项，然后把页表基地址告诉 MMU，就可以启动 MMU 了。

以下图为例介绍地址映射过程：

a) CPU 发出虚拟地址 vaddr，假设为 0x12345678

b) MMU 根据 vaddr[31:20]找到一级页表项：

- 虚拟地址 0x12345678 是虚拟地址空间里第 0x123 个 1M，所以找到页表里第 0x123 项，根据此项内容知道它是一个段页表项。

段内偏移是 0x45678。

c) 从这个表项里取出物理基地址：Section Base Address，假设是 0x81000000

d) 物理基地址加上段内偏移得到：0x81045678

所以 CPU 要访问虚拟地址 0x12345678 时，实际上访问的是 0x81045678 的物理地址。

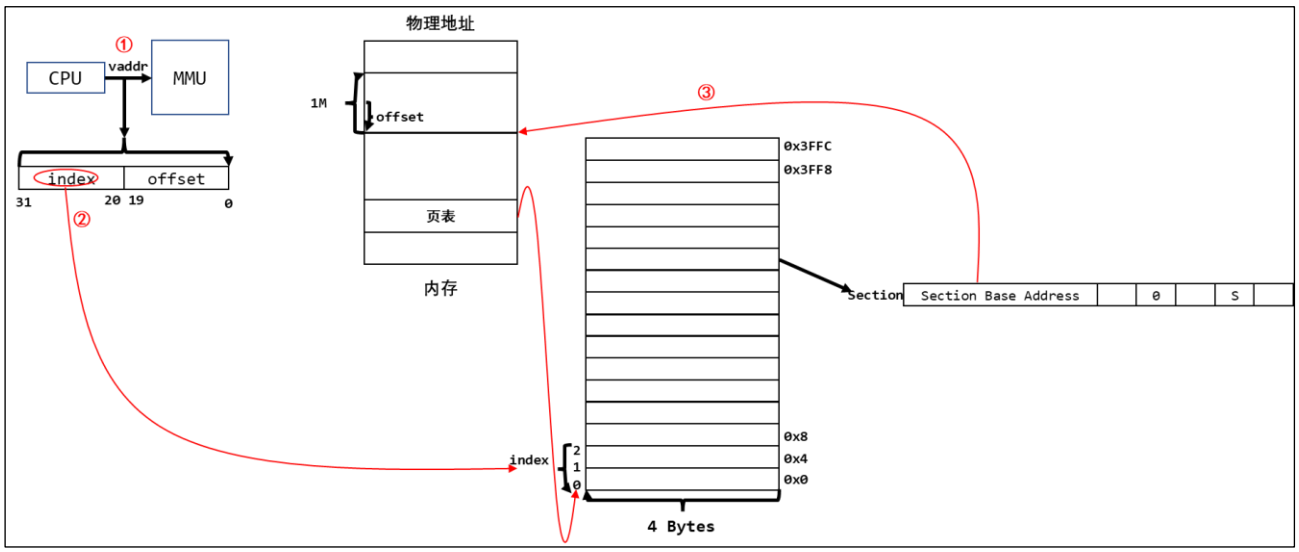


图 19.33 内存映射

2 二级页表映射过程

首先设置好一级页表、二级页表，并且把一级页表的首地址告诉 MMU。

以下图为例介绍地址映射过程：

- CPU 发出虚拟地址 vaddr，假设为 0x12345678
- MMU 根据 vaddr[31:20]找到一级页表项：

虚拟地址 0x12345678 是虚拟地址空间里第 0x123 个 1M，所以找到页表里第 0x123 项。根据此项内容知道它是一个二级页表项。

- 从这个表项里取出地址，假设是 address，这表示的是二级页表项的物理地址；
- vaddr[19:12]表示的是二级页表项中的索引 index 即 0x45，在二级页表项中找到第 0x45 项；
- 二级页表项格式如下：

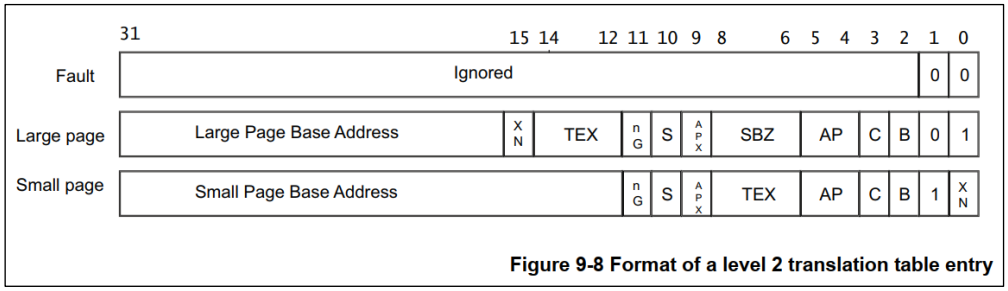


图 19.34 二级页表

里面含有这 4K 或 1K 物理空间的基地址 page base addr，假设是 0x81889000：

它跟 vaddr[11:0]组合得到物理地址：0x81889000 + 0x678 = 0x81889678。

所以 CPU 要访问虚拟地址 0x12345678 时，实际上访问的是 0x81889678 的物理地址

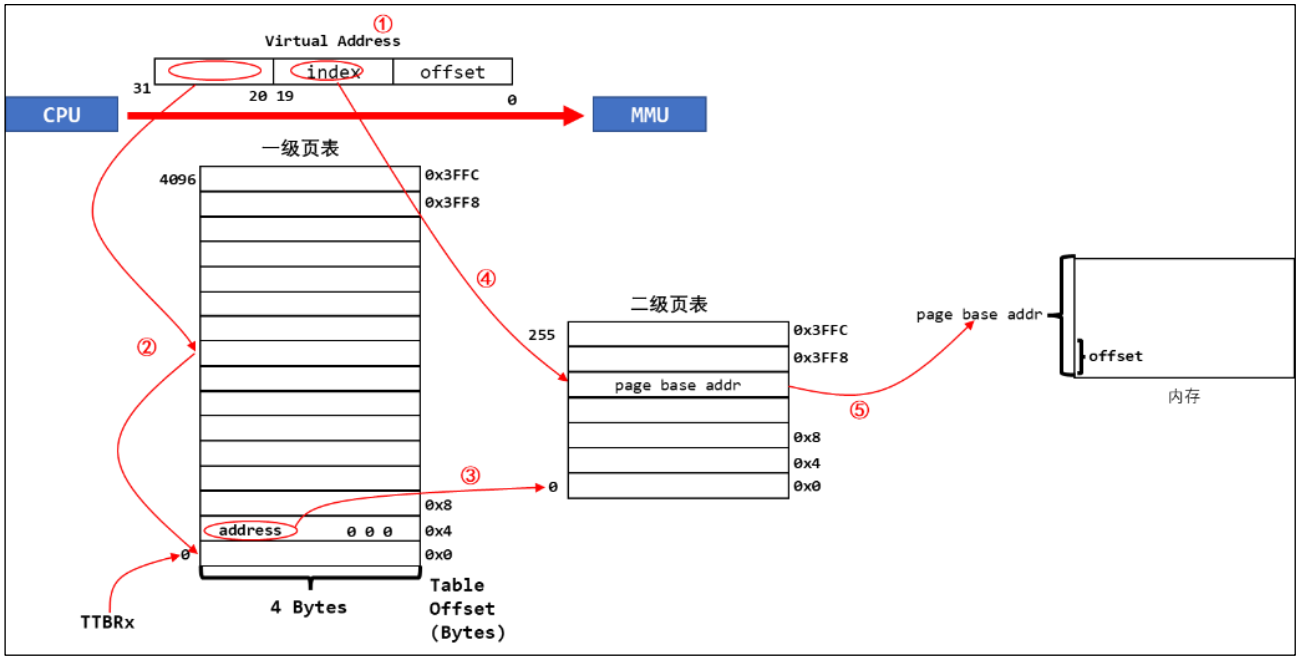


图 19.35 二级页表

19.9.3 怎么给 APP 新建一块内存映射

1 mmap 调用过程

从上面内存映射的过程可以知道，要给 APP 新开劈一块虚拟内存，并且让它指向某块内核 buffer，我们要做这些事：

① 得到一个 vm_area_struct，它表示 APP 的一块虚拟内存空间；

很幸运，APP 调用 mmap 系统函数时，内核就帮我们构造了一个 vm_area_stuct 结构体。里面含有虚拟地址的地址范围、权限。

② 确定物理地址：

你想映射某个内核 buffer，你需要得到它的物理地址，这得由你提供。

③ 给 vm_area_struct 和物理地址建立映射关系：

也很幸运，内核提供有相关函数。

APP 里调用 mmap 时，导致的内核相关函数调用过程如下：

```
app:  mmap(addr, length, prot, flags, fd, offset)
-----------------------------------------------------------------
内核: SYSCALL_DEFINE6(mmap_pgoff ...  // mm/mmap.c
          vm_mmap_pgoff(file, addr, len, prot, flags, pgoff); // mm/util.c
             do_mmap_pgoff(file, addr, len, prot, flag, pgoff, &populate);  // include/linux/mm.h
                do_mmap(file, addr, len, prot, flags, 0, pgoff, populate); // mm/mmap.c
                    addr = get_unmapped_area(file, addr, len, pgoff, flags);  // 得到未使用的虚拟地址
                   addr = mmap_region(file, addr, len, vm_flags, pgoff);      // mm/mmap.c
                              vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);  // 分配新的vm_area_struct

                              // 设置 vm_area_struct
                              vma->vm_mm = mm;
                              vma->vm_start = addr;
                              vma->vm_end = addr + len;
                              vma->vm_flags = vm_flags;
                              vma->vm_page_prot = vm_get_page_prot(vm_flags);
                              vma->vm_pgoff = pgoff;

                              // 调用驱动程序的mmap
                              error = file->f_op->mmap(file, vma);
```

我们只需要实现驱动的mmap函数：
1. 提供物理地址
2.设置属性：cache, buffer
3.给vm_area_struct和物理地址建立映射

图 19.36 mmap 调用过程

2 cache 和 buffer

本小节参考 ARM 的 cache 和写缓冲器（write buffer）：

https://blog.csdn.net/gameit/article/details/13169445

使用 mmap 时，需要有 cache、buffer 的知识。下图是 CPU 和内存之间的关系，有 cache、buffer(写缓冲器)。Cache 是一块高速内存；写缓冲器相当于一个 FIFO，可以把多个写操作集合起来一次写入内存。

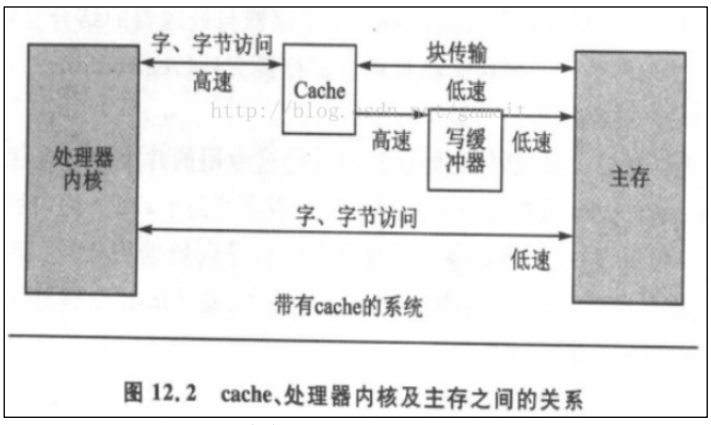


图 12.2 cache、处理器内核及主存之间的关系

图 19.37 cache

程序运行时有“局部性原理”，这又分为时间局部性、空间局部性。

- 时间局部性：

在某个时间点访问了存储器的特定位置，很可能在一小段时间里，会反复地

访问这个位置。

- 空间局部性：

  访问了存储器的特定位置，很可能在不久的将来访问它附近的位置。

  而 CPU 的速度非常快，内存的速度相对来说很慢。CPU 要读写比较慢的内存时，怎样可以加快速度？根据“局部性原理”，可以引入 cache。

① 读取内存 addr 处的数据时：

- 先看看 cache 中有没有 addr 的数据，如果有就直接从 cache 里返回数据：这被称为 cache 命中。
- 如果 cache 中没有 addr 的数据，则从内存里把数据读入，注意：它不是仅仅读入一个数据，而是读入一行数据(cache line)。
- 而 CPU 很可能会再次用到这个 addr 的数据，或是会用到它附近的数据，这时就可以快速地从 cache 中获得数据。

② 写数据：

- CPU 要写数据时，可以直接写内存，这很慢；也可以先把数据写入 cache，这很快。
- 但是 cache 中的数据终究是要写入内存的啊，这有 2 种写策略：

a) 写通(write through)：

- 数据要同时写入 cache 和内存，所以 cache 和内存中的数据保持一致，但是它的效率很低。能改进吗？可以！使用“写缓冲器”：cache 大哥，你把数据给我就可以了，我来慢慢写，保证帮你写完。
- 有些写缓冲器有“写合并”的功能，比如 CPU 执行了 4 条写指令：写第 0、1、2、3 个字节，每次写 1 字节；写缓冲器会把这 4 个写操作合并成一个写操作：写 word。对于内存来说，这没什么差别，但是对于硬件寄存器，这就有可能导致问题。
- 所以对于寄存器操作，不会启动 buffer 功能；对于内存操作，比如 LCD 的显存，可以启用 buffer 功能。

b) 写回(write back)：

- 新数据只是写入 cache，不会立刻写入内存，cache 和内存中的数据并不一致。
- 新数据写入 cache 时，这一行 cache 被标为“脏”(dirty)；当 cache 不够用时，才需要把脏的数据写入内存。

使用写回功能，可以大幅提高效率。但是要注意 cache 和内存中的数据很可能不一致。这在很多时间要小心处理：比如 CPU 产生了新数据，DMA 把数据从内存搬到网卡，这时候就要 CPU 执行命令先把新数据从 cache 刷到内存。反过来也是一样的，DMA 从网卡得过了新数据存在内存里，CPU 读数据之前先把 cache 中的数据丢弃。

是否使用 cache、是否使用 buffer，就有 4 种组合(Linux 内核文件 arch\arm\include\asm\pgtable-2level.h)：

```
#define L_PTE_MT_UNCACHED   (_AT(pteval_t, 0x00) << 2)  /* 0000 */
#define L_PTE_MT_BUFFERABLE (_AT(pteval_t, 0x01) << 2)  /* 0001 */
#define L_PTE_MT_WRITETHROUGH   (_AT(pteval_t, 0x02) << 2)  /* 0010 */
#define L_PTE_MT_WRITEBACK  (_AT(pteval_t, 0x03) << 2)  /* 0011 */
```

图 19.38 cache 和 buffer 的组合

上面 4 种组合对应下表中的各项，一一对应(下表来自 s3c2410 芯片手册，高架构的 cache、buffer 更复杂，但是这些基础知识没变)：

表 19-3 s3c2410 的 cache&buffer 组合

是否启用 cache	是否启用 buffer	说明
0	0	Non-cached, non-buffered (NCNB) 读、写都直达外设硬件
0	1	Non-cached buffered (NCB) 读、写都直达外设硬件； 写操作通过 buffer 实现，CPU 不等待写操作完成，CPU 会马上执行下一条指令
1	0	Cached, write-through mode (WT)，写通 读：cache hit 时从 cahce 读数据；cache miss 时已入一行数据到 cache； 写：通过 buffer 实现，CPU 不等待写操作完成，CPU 会马上执行下一条指令
1	1	Cached, write-back mode (WB)，写回 读：cache hit 时从 cahce 读数据；cache miss 时已入一行数据到 cache； 写：通过 buffer 实现，cache hit 时新数据不会到达硬件，而是在 cahce 中被标为“脏”；cache miss 时，通过 buffer 写入硬件，CPU 不等待写操作完成，CPU 会马上执行下一条指令

- 第 1 种是不使用 cache 也不使用 buffer，读写时都直达硬件，这适合寄存器的读写。
- 第 2 种是不使用 cache 但是使用 buffer，写数据时会用 buffer 进行优化，可能会有“写合并”，这适合显存的操作。因为对显存很少有读操作，基本都是写操作，而写操作即使被“合并”也没有关系。
- 第 3 种是使用 cache 不使用 buffer，就是“write through”，适用于只读设备：在读数据时用 cache 加速，基本不需要写。
- 第 4 种是既使用 cache 又使用 buffer，适合一般的内存读写。

3 驱动程序要做的事

驱动程序要做的事情有 3 点：

① 确定物理地址

② 确定属性：是否使用 cache、buffer

③ 建立映射关系

参考 Linux 源文件，示例代码如下：

```
// drivers/video/fbdev/mxsfb.c
static int mxsfb_mmap(struct fb_info *info, struct vm_area_struct *vma)
{
    u32 len;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset < info->fix.smem_len) {
        /* mapping framebuffer memory */
        len = info->fix.smem_len - offset;
        vma->vm_pgoff = (info->fix.smem_start + offset) >> PAGE_SHIFT;
    } else
        return -EINVAL;                          1.确定物理地址，单位：page, 4k

    len = PAGE_ALIGN(len);
    if (vma->vm_end - vma->vm_start > len)
        return -EINVAL;                          2.设置属性：cache? buffer ?

    /* make buffers bufferable */
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);

    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
                vma->vm_end - vma->vm_start, vma->vm_page_prot)) {
        dev_dbg(info->device, "mmap remap_pfn_range failed\n");
        return -ENOBUFS;
    }

    return 0;                                    3.映射
} « end mxsfb_mmap »
```

图 19.39 驱动程序要做的 3 件事

还有一个更简单的函数：

```
// drivers/video/fbdev/controlfb.c
static int controlfb_mmap(struct fb_info *info,
                          struct vm_area_struct *vma)
{
    unsigned long mmio_pgoff;
    unsigned long start;
    u32 len;

    start = info->fix.smem_start;                1.确定物理地址
    len = info->fix.smem_len;
    mmio_pgoff = PAGE_ALIGN((start & ~PAGE_MASK) + len) >> PAGE_SHIFT;
    if (vma->vm_pgoff >= mmio_pgoff) {
        if (info->var.accel_flags)
            return -EINVAL;
        vma->vm_pgoff -= mmio_pgoff;
        start = info->fix.mmio_start;
        len = info->fix.mmio_len;                2.确定属性：cache ? buffer ?
        vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    } else {
        /* framebuffer */
        vma->vm_page_prot = pgprot_cached_wthru(vma->vm_page_prot);
    }

    return vm_iomap_memory(vma, start, len);     3.映射
} « end controlfb_mmap »
```

图 19.40 controlfb_mmap 函数

19.9.4 编程

使用 GIT 命令载后，本节源码位于这个目录下：

01_all_series_quickstart
05_嵌入式 Linux 驱动开发基础知识\source\07_mmap

目的：我们在驱动程序中申请一个 8K 的 buffer，让 APP 通过 mmap 能直接访问。

1 APP 编程

APP 怎么写？open 驱动、buf=mmap(……)映射内存，直接读写 buf 就可以了，代码如下：

```
22      /* 1. 打开文件 */
23      fd = open("/dev/hello", O_RDWR);
24      if (fd == -1)
25      {
26              printf("can not open file /dev/hello\n");
27              return -1;
28      }
29
30      /* 2. mmap
31       * MAP_SHARED  : 多个 APP 都调用 mmap 映射同一块内存时，对内存的修改大家都可以看到。
32       *               就是说多个 APP、驱动程序实际上访问的都是同一块内存
33       * MAP_PRIVATE : 创建一个 copy on write 的私有映射。
34       *               当 APP 对该内存进行修改时，其他程序是看不到这些修改的。
35       *               就是当 APP 写内存时，内核会先创建一个拷贝给这个 APP，
36       *               这个拷贝是这个 APP 私有的，其他 APP、驱动无法访问。
37       */
38      buf =  mmap(NULL, 1024*8, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
39      if (buf == MAP_FAILED)
40      {
41              printf("can not mmap file /dev/hello\n");
42              return -1;
43      }
```

最难理解的是 mmap 函数 MAP_SHARED、MAP_PRIVATE 参数。使用 MAP_PRIVATE 映射时，在没有发生写操作时，APP、驱动访问的都是同一块内存；当 APP 发起写操作时，就会触发“copy on write”，即内核会先创建该内存块的拷贝，APP 的写操作在这个新内存块上进行，这个新内存块是 APP 私有的，别的 APP、驱动看不到。

仅用 MAP_SHARED 参数时，多个 APP、驱动读、写时，操作的都是同一个内存块，“共享”。

MAP_PRIVATE 映射是很有用的，Linux 中多个 APP 都会使用同一个动态库，在没有写操作之前大家都使用内存中唯一一份代码。当 APP1 发起写操作时，内核会为它复制一份代码，再执行写操作，APP1 就有了专享的、私有的动态库，在里面做的修改只会影响到 APP1。其他程序仍然共享原先的、未修改的代码。

有了这些知识后，下面的代码就容易理解了，请看代码中的注释：

```
45      printf("mmap address = 0x%x\n", buf);
46      printf("buf origin data = %s\n", buf); /* old */
47
48      /* 3. write */
49      strcpy(buf, "new");
50
51      /* 4. read & compare */
52      /* 对于 MAP_SHARED 映射:  str = "new"
53       * 对于 MAP_PRIVATE 映射: str = "old"
54       */
55      read(fd, str, 1024);
```

```
56      if (strcmp(buf, str) == 0)
57      {
58              /* 对于 MAP_SHARED 映射，APP 写的数据驱动可见
59               * APP 和驱动访问的是同一个内存块
60               */
61              printf("compare ok!\n");
62      }
63      else
64      {
65              /* 对于 MAP_PRIVATE 映射，APP 写数据时，是写入另一个内存块(是原内存块的
"拷贝")
66               */
67              printf("compare err!\n");
68              printf("str = %s!\n", str);  /* old */
69              printf("buf = %s!\n", buf);  /* new */
70      }
```

执行测试程序后，查看到它的进程号 PID，执行这样的命令查看这个程序的内存使用情况：

```
[root@100ask:~]# cat  /proc/PIC/maps
```

2 驱动编程

驱动程序要做什么？

① 分配一块 8K 的内存

使用哪一个函数分配内存？

表 19-4 分配内存函数

函数名	说明
kmalloc	分配到的内存物理地址是连续的
kzalloc	分配到的内存物理地址是连续的，内容清 0
vmalloc	分配到的内存物理地址不保证是连续的
vzalloc	分配到的内存物理地址不保证是连续的，内容清 0

我们应该使用 kmalloc 或 kzalloc，这样得到的内存物理地址是连续的，在 mmap 时后 APP 才可以使用同一个基地址去访问这块内存。(如果物理地址不连续，就要执行多次 mmap 了)。

② 提供 mmap 函数

关键在于 mmap 函数，代码如下：

```
static int hello_drv_mmap(struct file *file, struct vm_area_struct *vma)
{
    /* 获得物理地址 */
    unsigned long phy = virt_to_phys(kernel_buf); 1.得到物理地址
                                                  kernel_buf是内核使用的虚拟地址
    /* 设置属性: cache, buffer */
    vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot); 2.设置属性:
                                                                不使用cache
    /* map */  3.映射                                           使用buffer
                                  注意:按page映射
    if (remap_pfn_range(vma, vma->vm_start, phy >> PAGE_SHIFT,
                vma->vm_end - vma->vm_start, vma->vm_page_prot)) {
        printk("mmap remap_pfn_range failed\n");
        return -ENOBUFS;
    }

    return 0;
}
```

图 19.41 hello_drv_mmap

要注意的是，remap_pfn_range 中，pfn 的意思是“Page Frame Number”。在 Linux 中，整个物理地址空间可以分为第 0 页、第 1 页、第 2 页，诸如此类，这就是 pfn。假设每页大小是 4K，那么给定物理地址 phy，它的 pfn = phy / 4096 = phy >> 12。内核的 page 一般是 4K，但是也可以配置内核修改 page 的大小。所以为了通用，pfn = phy >> PAGE_SHIFT。

APP 调用 mmap 后，会导致驱动程序的 mmap 函数被调用，最终 APP 的虚拟地址和驱动程序中的物理地址就建立了映射关系。APP 可以直接访问驱动程序的 buffer。

3 上机测试

在 Ubuntu 中编译好驱动、测试程序，放到开发板。

在开发板上安装驱动、执行测试程序。

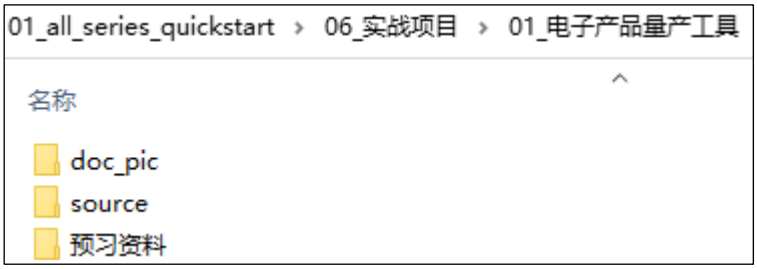
>>>>>>>>第六篇 实战项目<<<<<<<<

掌握了嵌入式 Linux 应用开发基础、驱动开发基础后，就可以做项目了。百问网的《第六篇 实战项目》是录播课程，对应的文档和源码放在 GIT 仓库里。

GIT 仓库地址：

https://e.coding.net/weidongshan/01_all_series_quickstart.git

目录的位置如下：



除了录播的项目之外，还有更多直播项目。直播的项目有 3 大类：调试 Bug、毕业设计级别项目、产品级别项目。会不断更新这些项目。

直播的视频也都录制下来，放在官网：

从零编写带 GUI 的应用：https://www.bilibili.com/video/BV1it4y1Q75z

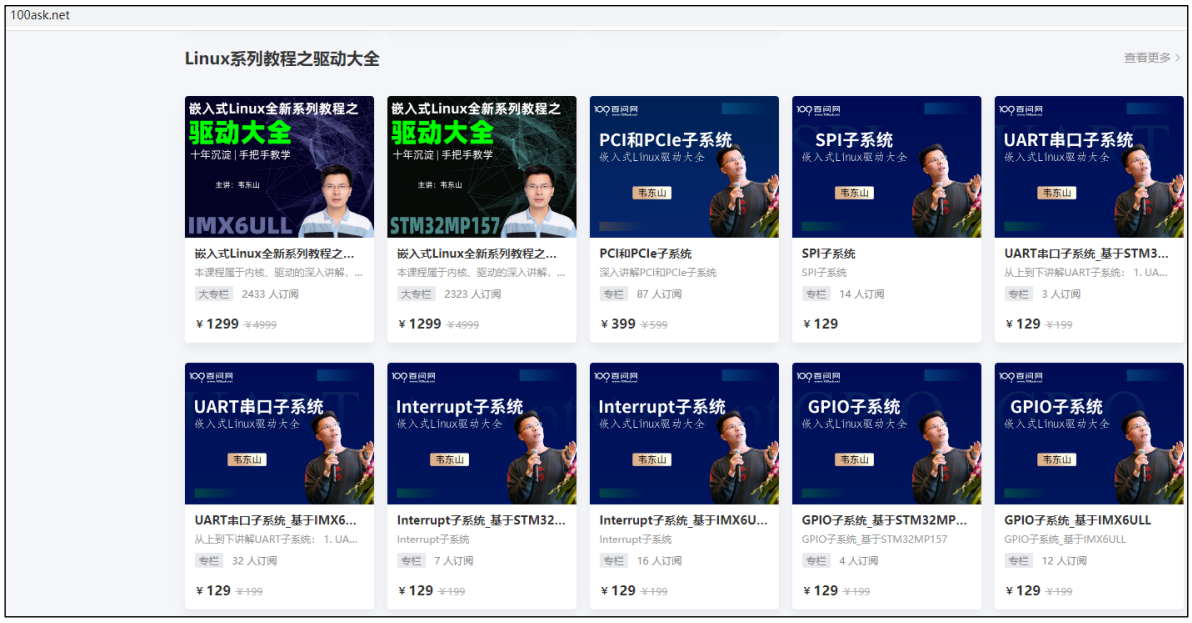
>>>>>>>第七篇 驱动大全<<<<<<<

驱动大全是收费视频，适合想深入研究某个内核主题、深入研究 Linux 驱动的人。驱动大全的录制目标，是称为“驱动开发的字典”，并不需要从头学完，用到哪部分，再学习哪部分。

驱动大全的文档、源码等所有配套资料，放在另一个 GIT 仓库：

```
https://e.coding.net/weidongshan/linux/doc_and_source_for_drivers.git
```

视频放在官网：



驱动大全使用两个板子进行录制：IMX6ULL、STM32MP157，购买一套就可以得到另一套。可以购买整套驱动大全，也可以单独都买里面的某个专题。

驱动大全尚未录制完毕，正在持续更新。

驱动大全规划的大纲如下：

>>>>>>>第八篇 其它应用<<<<<<<

第1章 Qt 应用开发(仅供测试)

QT（发音为“cute”）是跨平台的应用程序框架和小部件工具包，用于创建经典和嵌入式图形用户界面，和应用上具有很少或没有变化的各种软件和硬件平台上运行在底层代码库中，仍然是具有本机功能和速度的本机应用程序。Qt 是目前正在开发的两个 Qt 的公司，一家上市公司，以及 Qt 的项目下的开源治理，包括个人开发者和工作推进 Qt 的公司。Qt 可在商业许可和开源 GPL 2.0，GPL 3.0 和 LGPL 3.0 许可下使用。

参考网址：

- Qt 官网：https://www.qt.io/
- 维基百科：https://wiki.qt.io/
- Qt 下载：https://download.qt.io/archive/

注意：

- 安装 Qt 前序先使用 buildroot 编译 100ask_imx6ull-qt_defconfig 配置文件，生成包含 qt 环境的文件系统，具体编译过程请参考构建根文件系统，并将生成的系统烧写进 SD 卡，并启动。
- 百问网没有 QT 教程，无法对 QT 的技术问题提供支持；本章内容仅供参考。
- 此安装教程只在百问网提供的 Ubuntu18.04LTS 上进行过测试，其它版本请酌情修改。

1.1 安装 Qtcreator

QtCreator 下载网址：

https://download.qt.io/official_releases/qtcreator/

这里我们使用的 qtcreator 版本为 4.8：

```
book@100ask:~$ cd /home/book
book@100ask:~$ cp /media/03-Tools/qt-creator-opensource-linux-x86_64-4.1.0.run  /home/book
book@100ask:~$ chmod a+x qt-creator-opensource-linux-x86_64-4.1.0.run
book@100ask:~$ sudo ./qt-creator-opensource-linux-x86_64-4.1.0.run
```

此安装过程类似 Windows 下应用的安装方法，一路点击下一步即可。

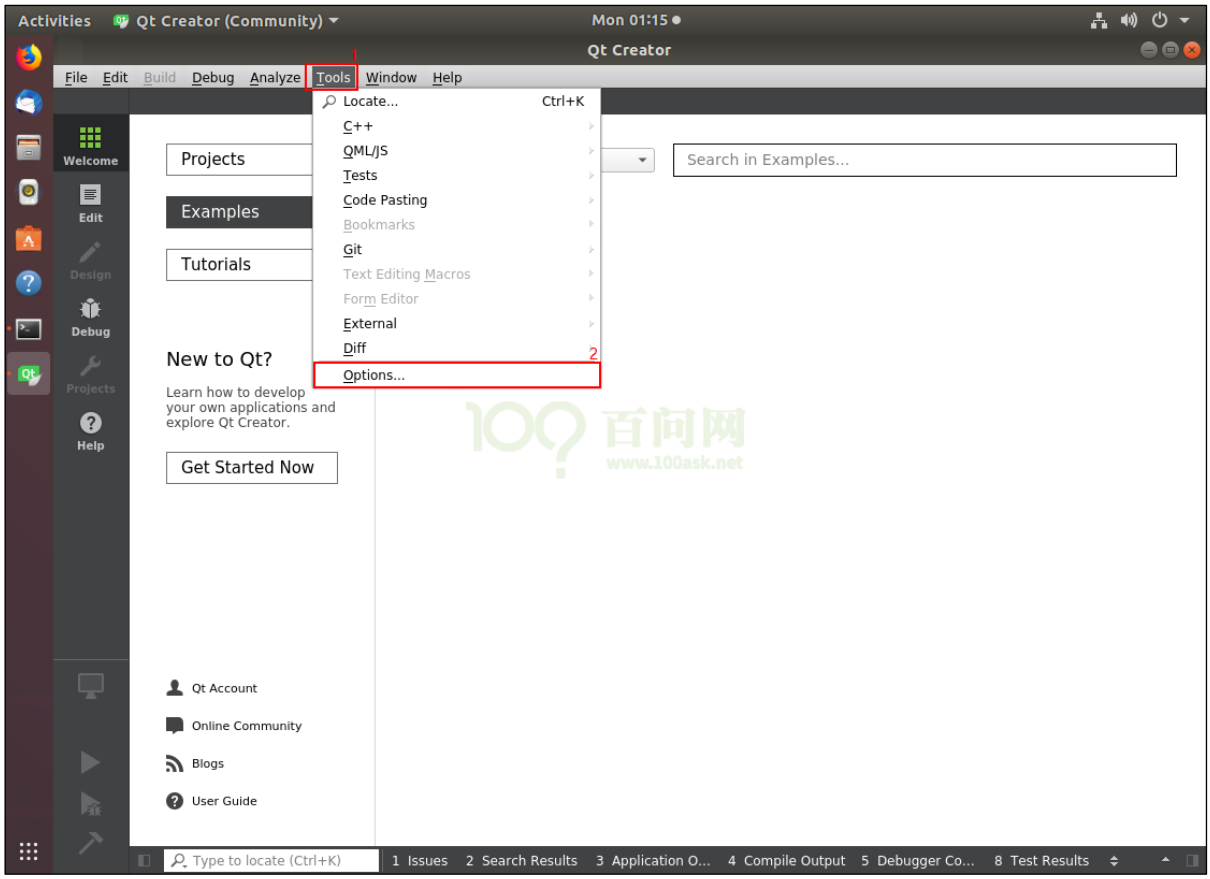
1.2 配置 QtCreator 开发环境

在 Ubuntu 桌面打开终端，进入安装目录，运行如下命令启动 QtCreator：

```
book@100ask:~$ cd  /opt/qtcreator-4.8.0/bin
book@100ask: /opt/qtcreator-4.8.0/bin$ sudo ./qtcreator
```

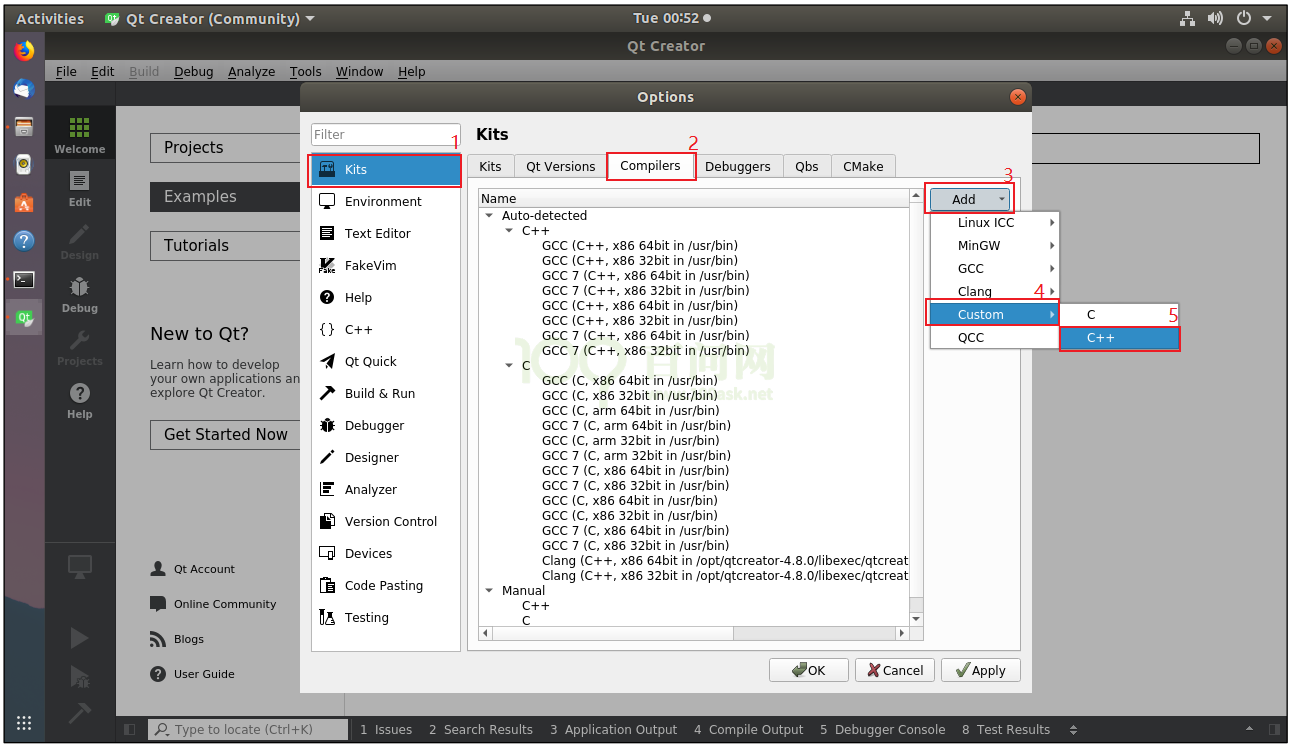
第1步 打开选项界面：

在 QtCreator 界面中，依次点击 **tools -> options**：



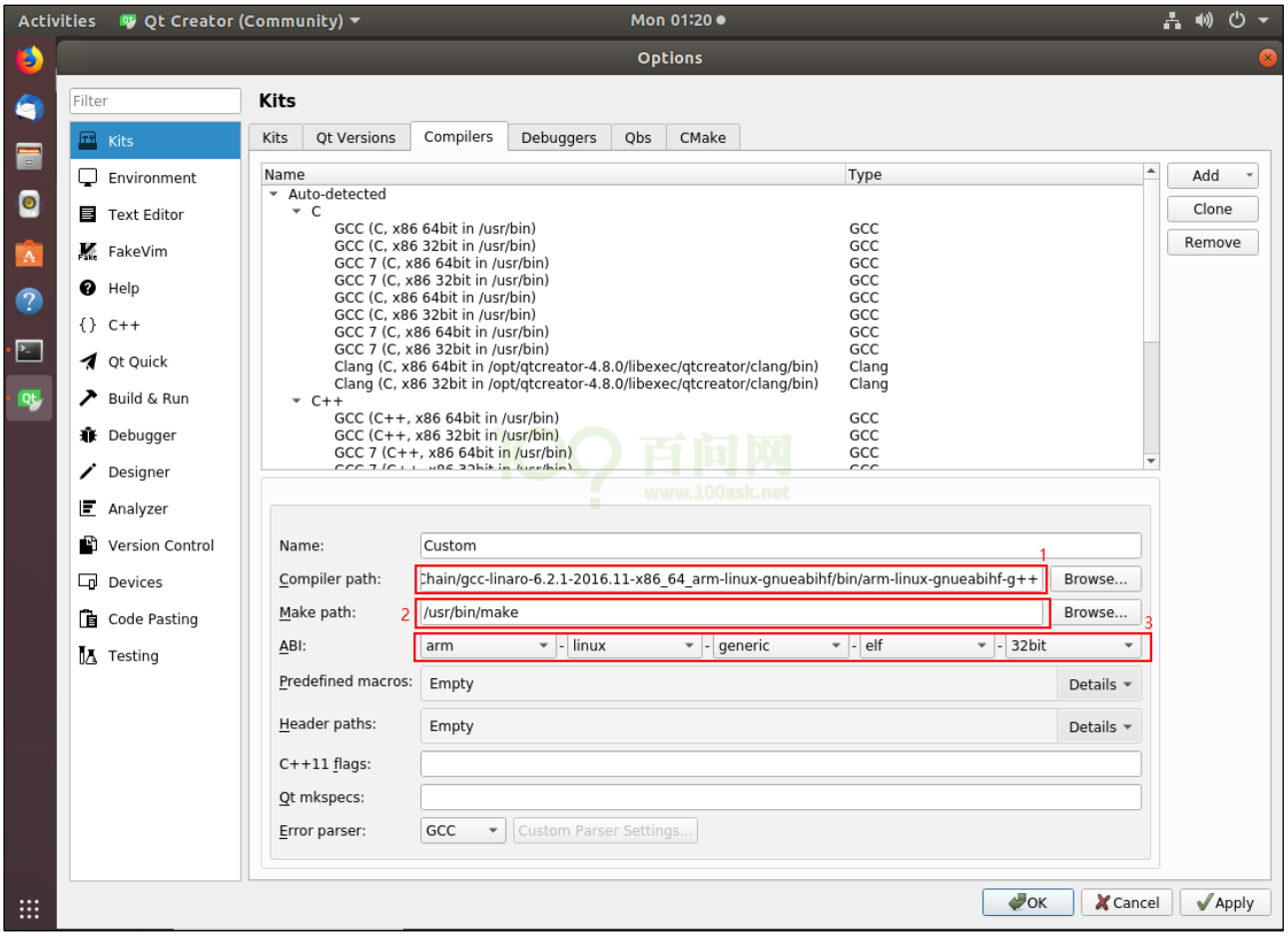
第2步 选择编译器：

在出现的选项对话框中，在左边点击 **Kits**，右边选择 **Compilers** 标签，并点击 **Add** 选择 **Custom->C++**：



第3步 设置编译器：

在弹出的对话框中填写以下内容：**Compiler path** ，**Make path** 和 **ABI**；填写完成后，点击 **Apply** 进行保存。



第4步 添加 QT 版本：

假设你已经编译出了 QT 的 SDK。

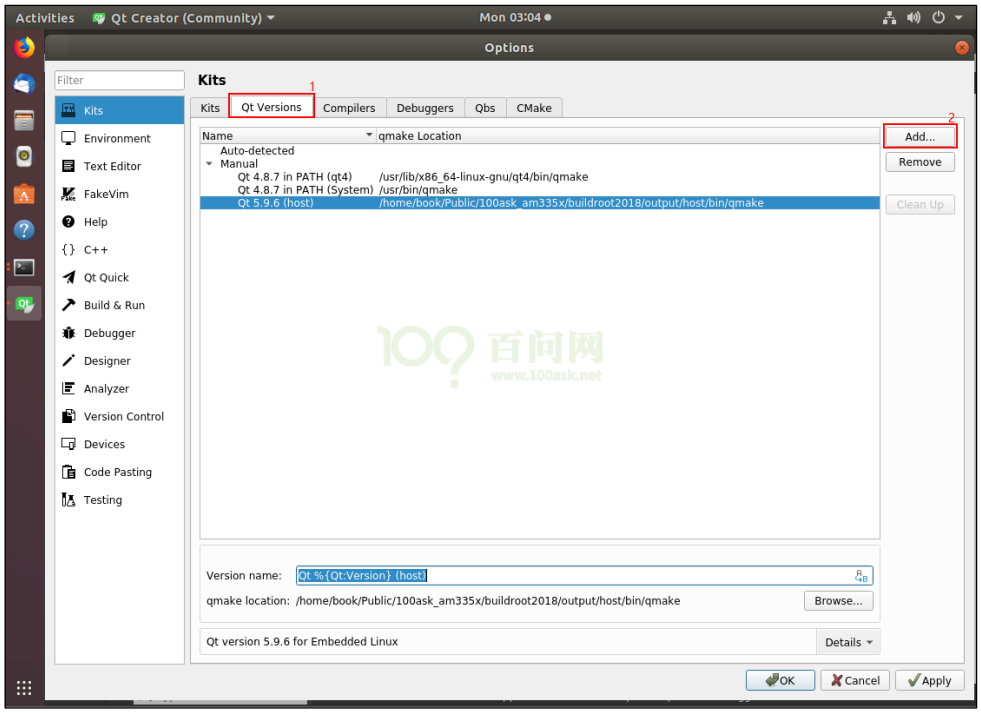
以 IMX6ULL 为例，你已经做了这些事情：

在执行“`make 100ask_imx6ull-qt_defconfig`”配置后，再执行“`make all`”可以编译出 QT 的 SDK 包。可以在/home/book/100ask_imx6ull-sdk 下执行这个命令找到 qmake，记住它的目录：

```
find  -name  qmake
```

在 QT 设置界面中，选择 Qt Version 标签，在右侧点击“Add...”，会弹出对话框，切换目录到选择 **qmake** 文件后，点击 **open** 按钮，设置完成之后，点击 **Apply** 按钮保存。

注意：qmake 文件是 buildroot 编译根文件系统后生成的，文件在 buildroot 目录下。



第5步 综合选择：

继续选择上边的 Kits 标签，点击右侧 Add，填写相应内容：

① Name：输入 100ask_imx6ull

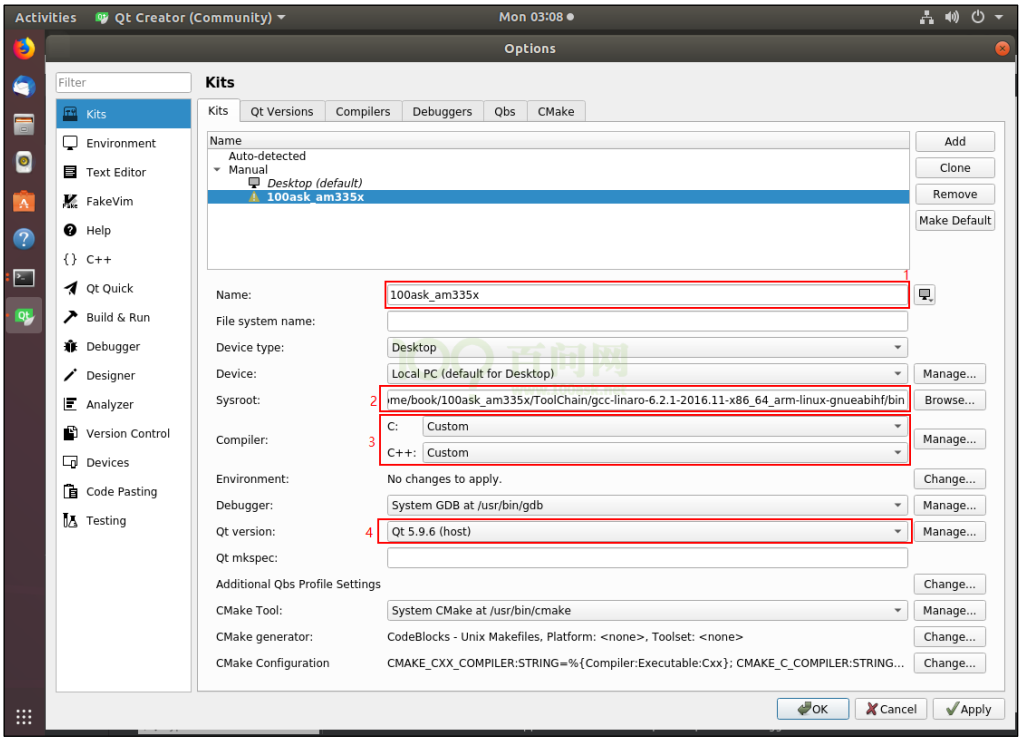
② Sysroot：输入交叉编译工具链的目录

③ compiler：c 和 c++这两个选择框里，都选择 Custom

④ Debugger：选择 None

⑤ Qtversio：选择上图中配置的“Qt5.9.6 (host) ”

配置完成后点击 apply，点击 OK。



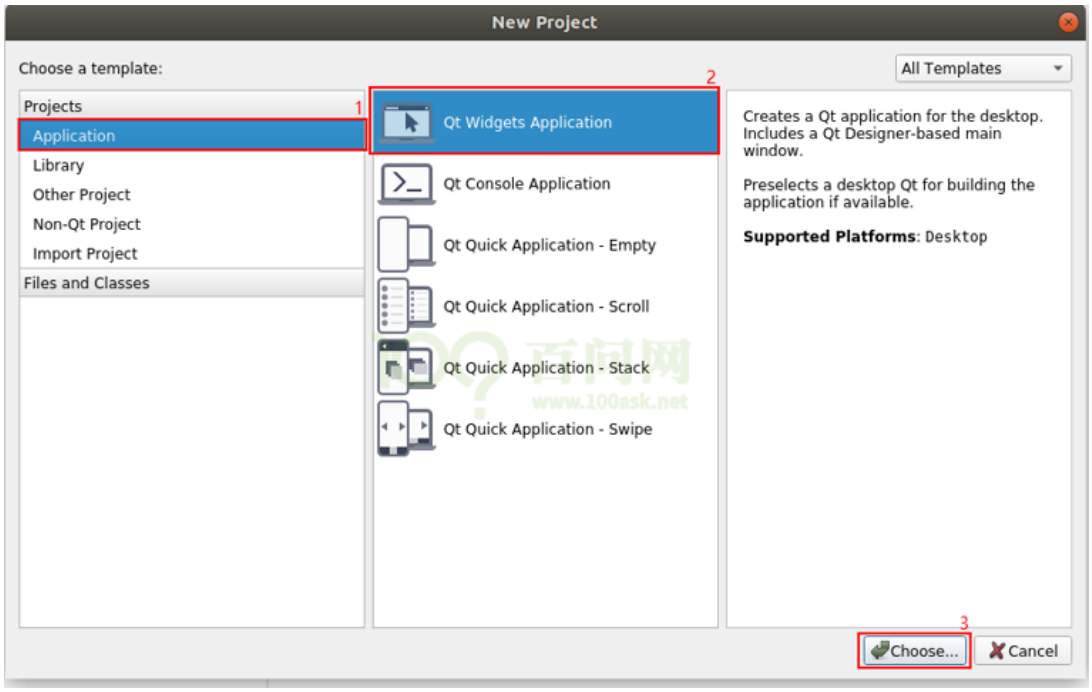
1.3 创建 Helloworld 项目

第1步 新建项目：

运行 QtCreator 后，在菜单栏选择 **File -> New File or Project**

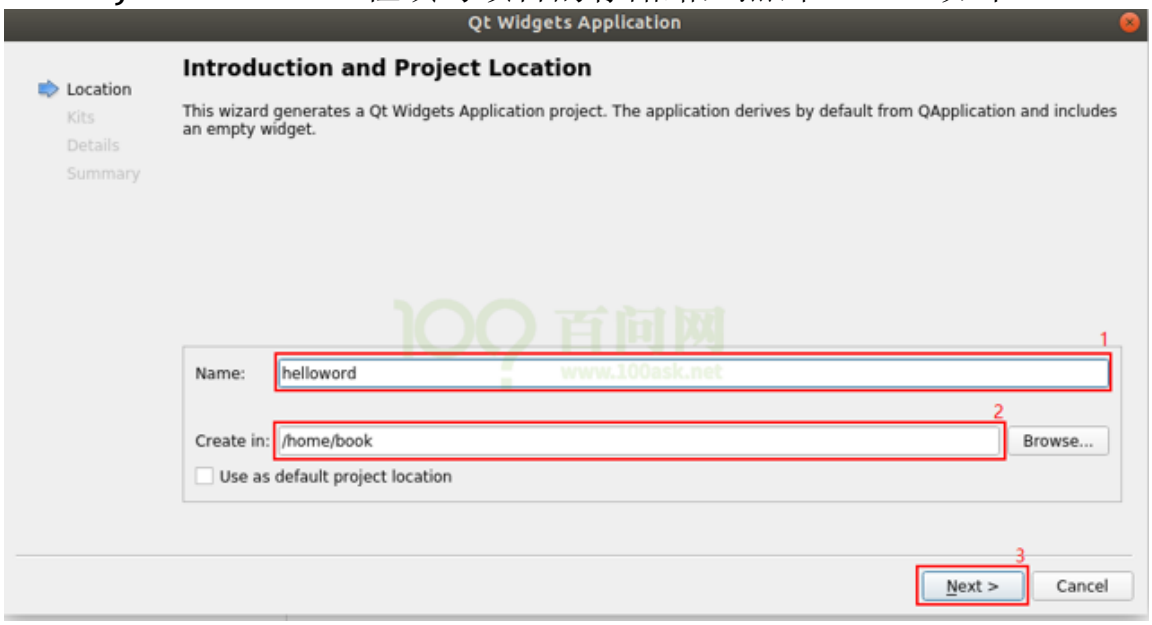
第2步 选择项目类型：

在打开的对话框中，依次选择 **Application -> Qt Widgets Application** ，点击 **Choose...** ，如下图所示：

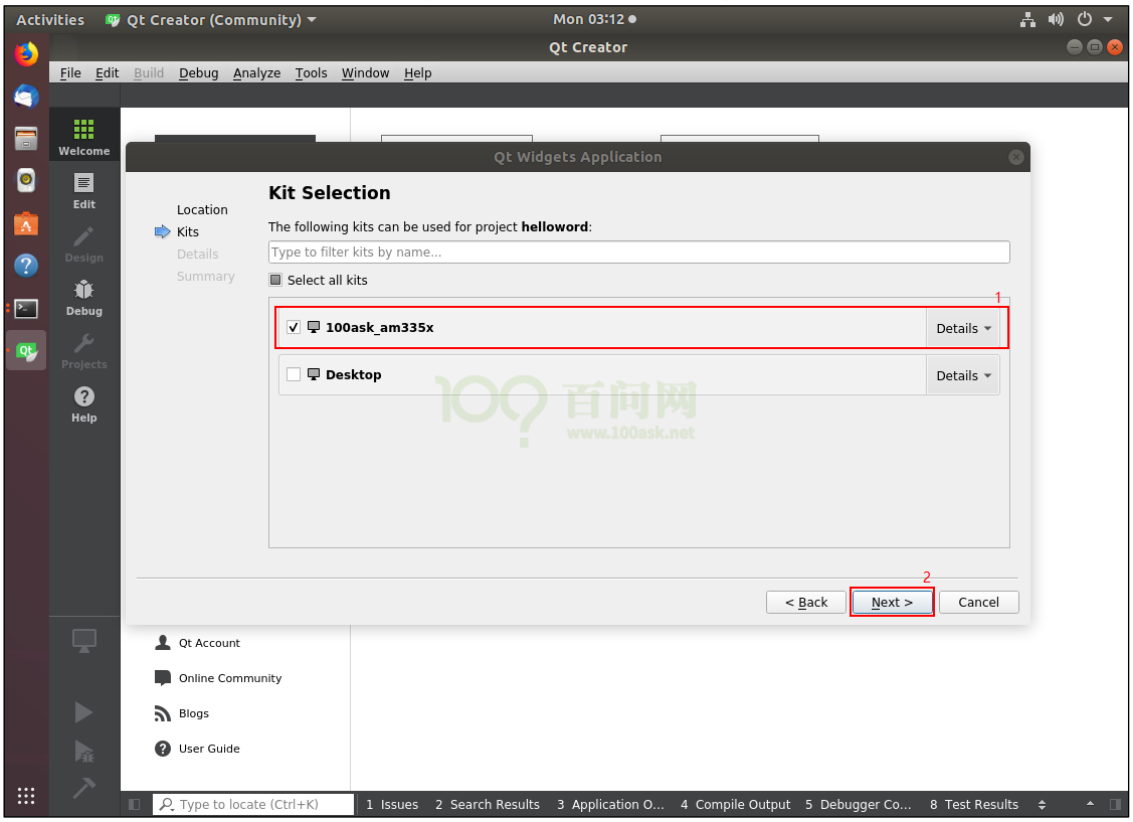


第3步 输入项目名字、设置保存位置：

在弹出的 Qt Widgets Application 对话框中，设置项目名称为 helloword,Create in 一栏填写项目的存储路，点击 Next。如下：

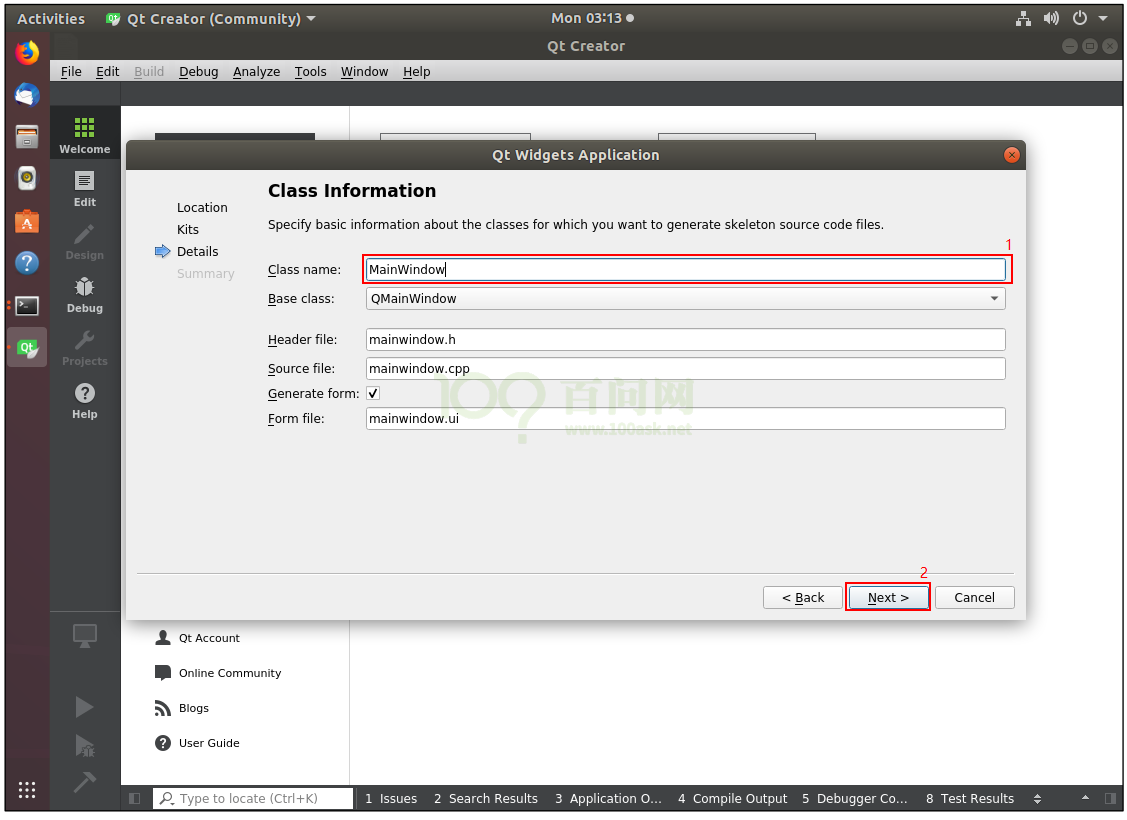


第4步 选择之前添加好的 Kits(100ask_imx6ull)，继续点击 Next，如下：



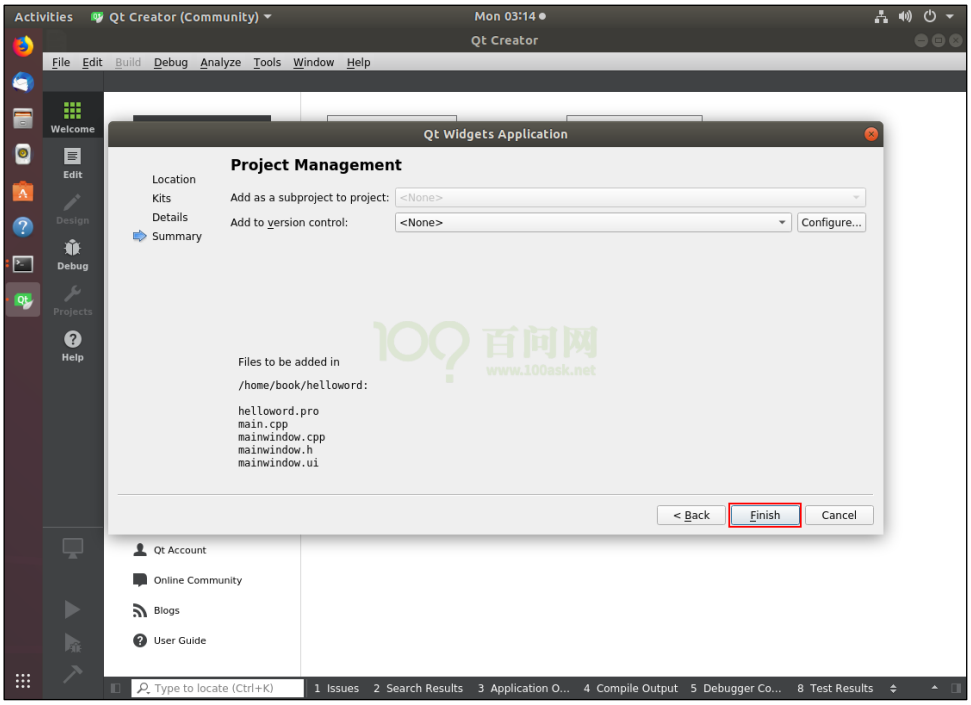
第5步 选择基类：

当前的的应用继承自哪种 Widget,默认选择 QMainWindow,然后点击 **Next** 进入下一步。



第6步 完成项目创建：

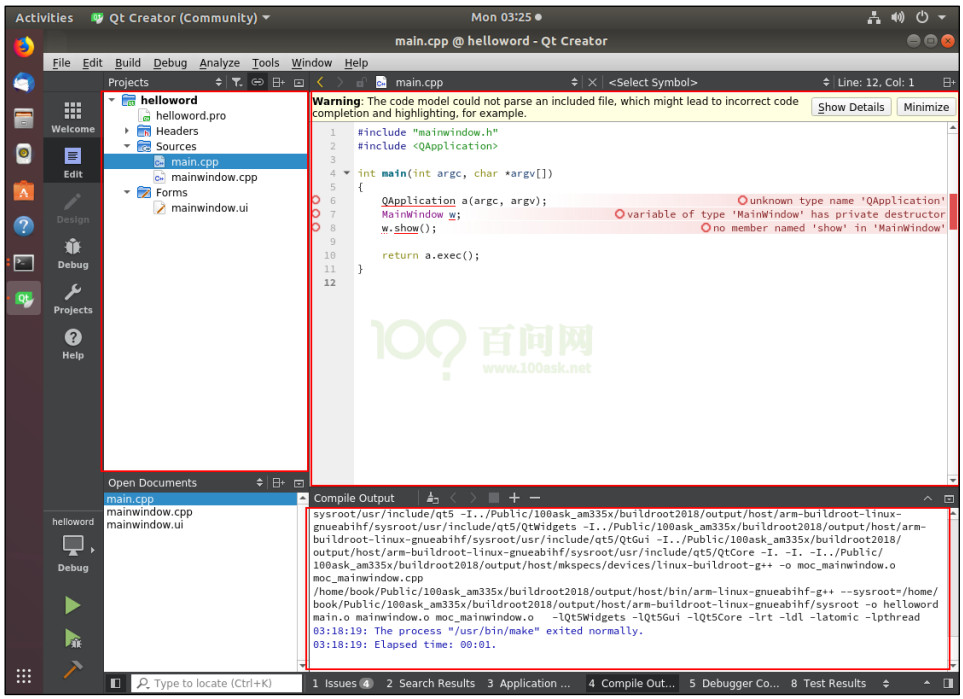
以上信息填写完后，点击 Finish，完成 Helloworld 项目的创建。



1.4 编译 Qt 程序

以下是创建好的 helloworld 项目截图，左侧是 QtCreator 创建好的项目目录结构，右侧是代码编辑区，

右下是编译时的输出信息：

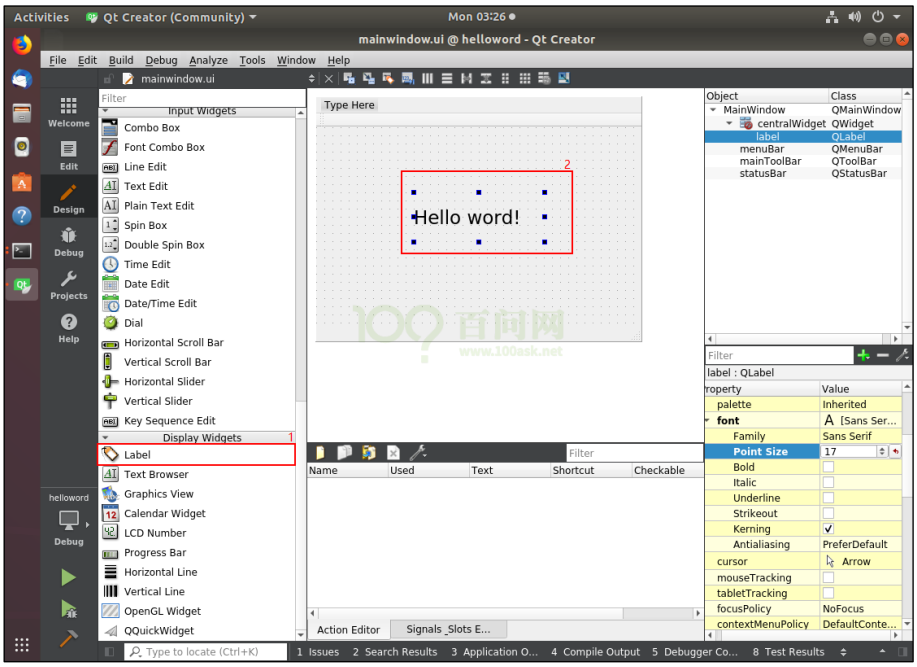


第1步 修改界面：

在上面的图 5-3-1 界面中，双击左侧的 Forms 里的 mainwindow.ui 文件，打开 Design 视图。

然后如下图所示，从左侧 Display Widgets 栏目下，拖动 Label 到中间的区域。

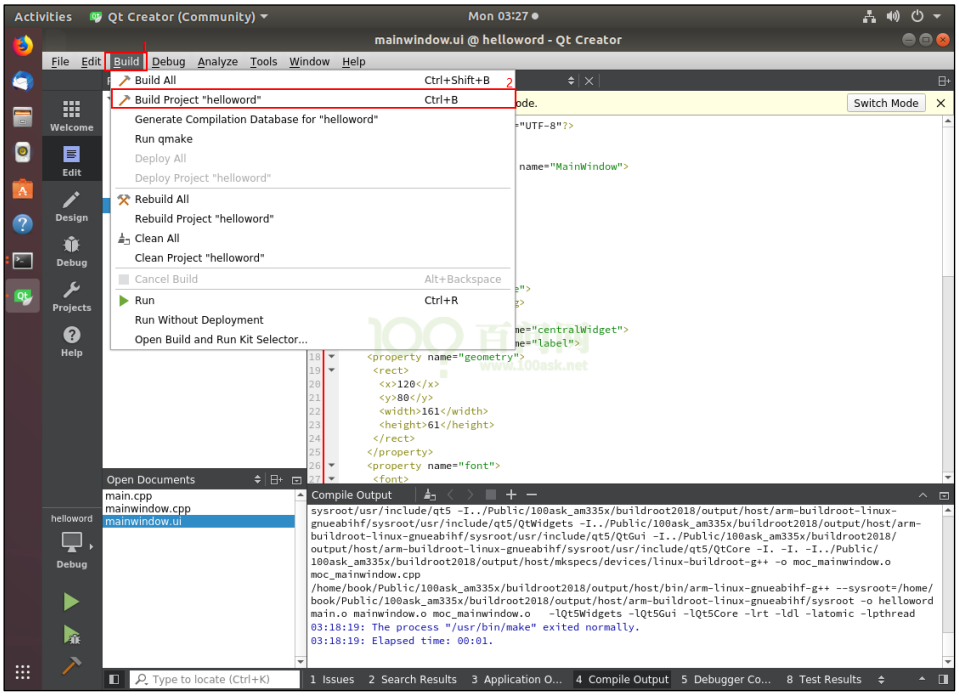
最后双击中间区域的 Label，修改内容为“Hello world! ”。



第2步 构建：

点击菜单栏 Build -> Build Project hellowrld，开始编译、构建项目。

在构建过程中，会在左下侧是“Compile Output”窗口打印构建信息。如果有错误，请根据提示出错信息修改，然后重新构建。



第3步 查看构建结果：

helloworld 项目构建成功后，编译好的二进制文件存放在以下目录中：
/home/book/build-helloword-100ask_imx6ull-Debug

可以使用 **file** 命令查看该 APP 是否被编译为 ARM 架构：

```
book@100ask:~/build-helloword-100ask_imx6ull-Debug$ file helloword
helloword: ELF 32-bit LSB executable, ARM, EABI5 version 1 (GNU/Linux), dynamically l
inked, interpreter /lib/ld-, for GNU/Linux 2.6.32, BuildID[sha1]=bb5f7737070472574fc
35af534b3e68e68bb02b6, with debug_info, not stripped
```

1.5 在开发板运行 Qt 程序

使用 ssh 远程登陆开发板，将生成的 QT 程序的可执行文件 helloworld 拷贝到开发板/usr/bin 目录下，然后在开发板上运行如下命令：

```
[root@100ask_imx6ull:~]#  helloworld
```

如果一切正常，将会在 LCD 屏幕上看到“Hello world!”的 Qt 窗口，如下图所示：

注意：由于我们对 QT 不专业，无法提供 QT 程序的技术支持。

第2章 开发板板载系统应用、库、工具、使用

注意：此章内容讲解开发板系统内应用/库工具等的使用介绍，仅在需要时参考学习 不需要逐一验证测试。

2.1 图形库和图形应用

2.1.1 Qt GUI

运行环境

- 支持的 INIT 服务：systemV
- 支持的开发板系统：Buildroot 版本系统

1 开发板 Qt GUI 简述

QT 是一种跨平台 C++图形用户界面应用程序开发框架。它既可以开发 GUI 程序，也可用于开发非 GUI 程序，比如控制台工具和服务器。Qt 是面向对象的框架，使用特殊的代码生成扩展以及一些宏，Qt 很容易扩展，并且允许真正地组件编程。

开发板会在出厂的时候烧写带有 Qt 运行时库的系统，并且提供了一个丰富的 HMI 演示系统，具体内容可以查看《MEasy HMI2.0 开发手册》。

在嵌入式 Linux 系统上，可以使用多个平台插件：EGLFS，LinuxFB，DirectFB 或 Wayland。但是，这些插件的可用性取决于实际硬件平台的特性以及 Qt 的配置方式。EGLFS 是许多主板上的默认插件。如果不合适，请使用 QT_QPA_PLATFORM 环境变量来请求另一个插件。另外，对于快速测试，请使用-platform 具有相同语法的命令行参数。例如在用户控制台下，可以使用如下命令进行配置。

```
export QT_QPA_PLATFORM=linuxfb
/usr/bin/mxapp2 --plugin tslib:/dev/input/event1
```

或者使用-platform linuxfb 指令平台插件，如下：

```
export QT_QPA_PLATFORM=linuxfb
/usr/bin/mxapp2 --plugin tslib:/dev/input/event1 -platform linuxfb
```

对于包含 GPU 的现代嵌入式 Linux 设备，推荐使用 EGLFS 插件，使用 GPU 进行渲染。如果嵌入式 Linux 设备不包含 GPU 或者不支持 OpenGL ES 测试，则只能指定 linuxfb 平台插件通过 CPU 进行软件的渲染了。

注意：从 Qt 5.0 开始，Qt 不再具有自己的窗口系统（QWS）实现，因此不再支持 Qt 早期版本上使用的 -qws 参数。

2 运行 linuxfb platform qtgui

使用系统默认的 qt 脚本来启动 gui 应用，开机自启动脚本存放在 `/etc/init.d/` 目录下文件名为 `S99myirhmi2` 如下命令执行 `/etc/init.d/S99myirhmi2 start` 表示启动 qt gui 程序。

```
[root@100ask:~]# /etc/init.d/S99myirhmi2 start
```

```
[root@100ask:~]# /etc/init.d/S99myirhmi2 start
[root@100ask:~]# QStandardPaths: XDG_RUNTIME_DIR points to non-existing path '/tmp/runtime-root', please create it with 0700 permissions.
qt.qpa.input: xkbcommon not available, not performing key mapping
qml: index=0
qml: currentIndex=0
qml: index=0
net_status 0
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
```

如需关闭 qt gui 可以执行/etc/init.d/S99myirhmi2 stop 命令，来关闭 qt gui。

如果您不想使用我们的开机自启动脚本，可以使用 mv 命令把/etc/init.d/S99myirhmi2 文件移动到任意目录，则开机不会自动加载此脚本。

2.1.2 显示测试 fb-test-app

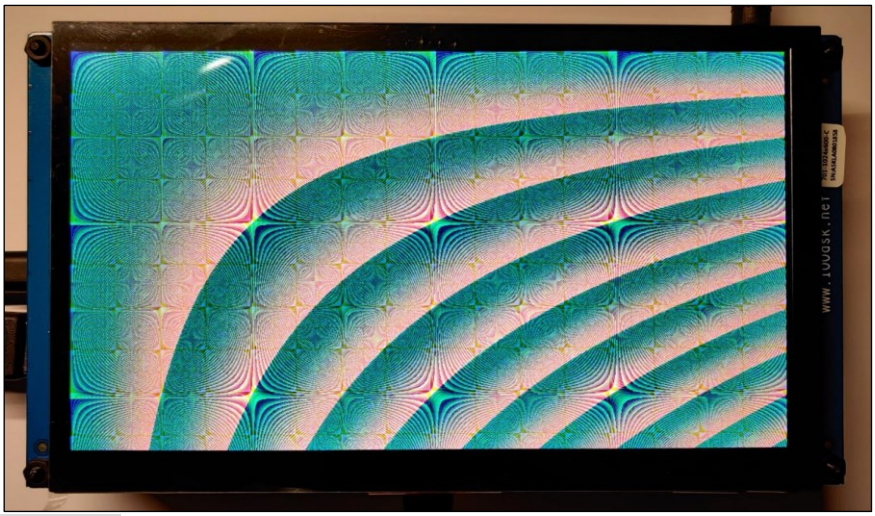
fb-test-app 是一套用来测试显示设备色彩性能的一套应用程序，使用此程序是之前需要先关闭默认启动的 qt gui 程序，开发板终端上执行/etc/init.d/S99myirhmi2 stop 命令来关闭 qt gui 之后继续执行如下测试操作。

① 使用 fb-test 来测试当前显示设备输出的 RGB 颜色，并获取到一些基本参数。

```
[root@100ask:~]# fb-test
fb-test 1.1.0 (rosetta)
fb res 1024x600 virtual 1024x600, line_len 4096, bpp 32
```

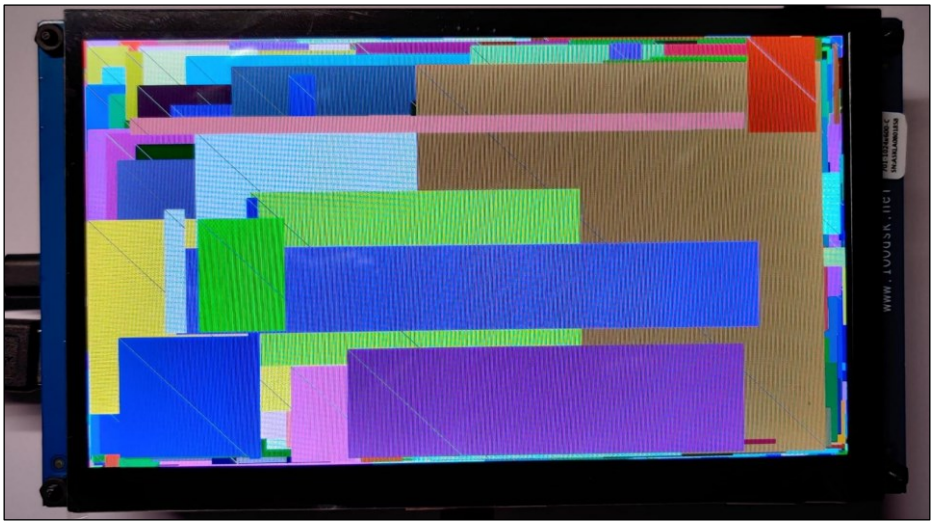
② 使用 fb-test-perf 对 lcd 显示进行性能分析， fb-test-perf <fbnum> <logfile> 使用其中<fbnum>表示 显示设备的设备节点<logfile>表示保存 log 文件的文件名，如下示例我们对 /dev/fb0 设备节点进行性能分析并把 log 文件保存在 log.txt 内。

```
[root@100ask:~]# fb-test-perf /dev/fb0 log.txt
perf 1.1.0 (rosetta)
sequential_horiz_singlepixel_read: 30720000 pix, 4871000 us, 6306713 pix/s
sequential_horiz_singlepixel_write: 735436800 pix, 4695710 us, 156618871 pix/s
sequential_vert_singlepixel_read: 25190400 pix, 4958052 us, 5080705 pix/s
sequential_vert_singlepixel_write: 158515200 pix, 4982069 us, 31817142 pix/s
sequential_line_read: 63283200 pix, 4976007 us, 12717666 pix/s
sequential_line_write: 1588838400 pix, 4869849 us, 326260300 pix/s
nonsequential_singlepixel_write: 74342400 pix, 5885829 us, 12630744 pix/s
nonsequential_singlepixel_read: 9830400 pix, 3357356 us, 2928018 pix/s
```



③ 使用 `fb-test-rect` 来对此显示屏做渲染测试。

```
[root@100ask:~]# fb-test-rect
rect 1.1.0 (rosetta)
```



2.1.3 开机动画 psplash

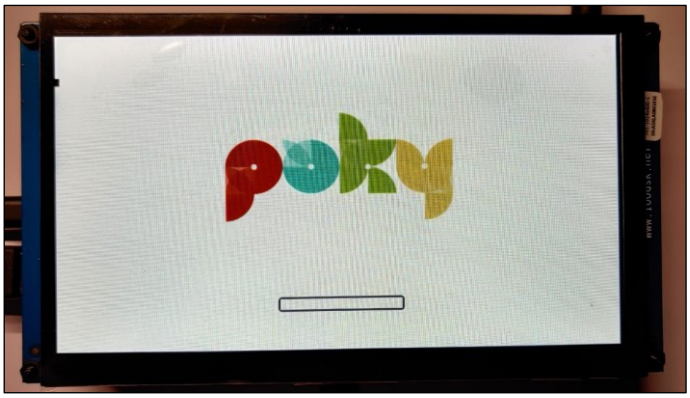
PSplash 是一个用户空间图形开机启动屏幕的应用程序，主要用于支持16bpp或32bpp帧缓冲区的嵌入式Linux设备。它有很少的依赖项(只是libc)，支持基本的图像和文本以及句柄回传。它的显示可以通过基本的源更改来配置。

还包括一个“终端”命令实用程序，用于将信息发送到 psplash，例如显示启动进度信息，psplash 包含 psplash-write psplash 两个应用程序，其中psplash 用来启动显示，psplash-write 用来写入进度提示信息退出程序等。

① 终端执行 psplash -n& 命令来启动 psplash 并将其在后台运行。

```
[root@100ask:~]# psplash -n &
```

```
[root@100ask:~]# psplash -n &
[1] 342
```

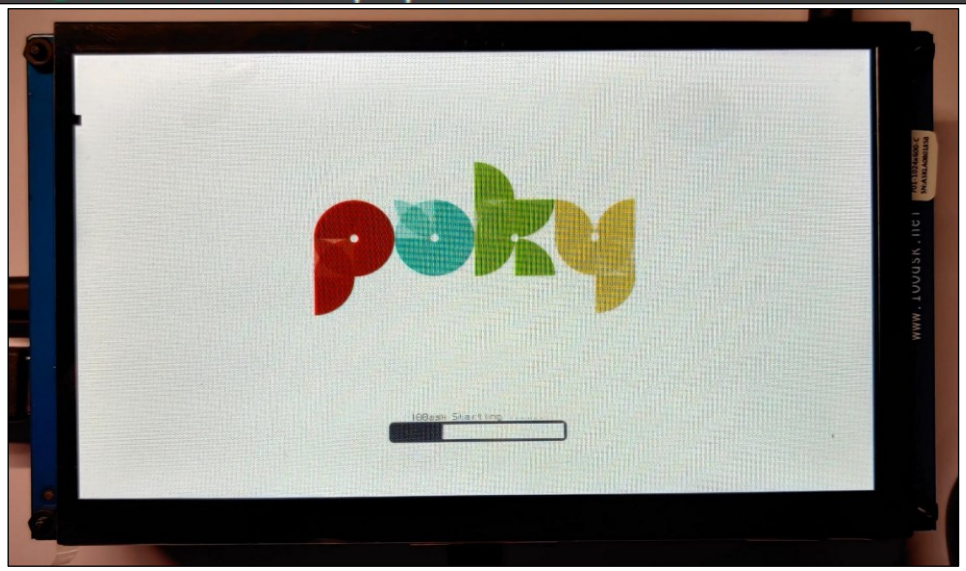


② 使用 psplash-write “ MSG <文本消息> “在系统进度条上方显示文本消息。

```
[root@100ask:~]# psplash-write "MSG 100ask Starting ......"
```

③ 使用 psplash-write “ PROGRESS <进度值> “在进度条哪里显示当前的运行进度，如下所示，显示当前启动进度为 30%。

```
[root@100ask:~]# psplash-write "PROGRESS 30"
```



④ 使用 psplash-write “ QUIT “来关闭 psplash 如下所示终端执行 psplash-write " QUIT " 命令即可关闭 psplash 程序，可以继续执行其它 GUI 应用，如 qt.

```
[root@100ask:~]# psplash-write " QUIT "
```

```
[root@100ask:~]# psplash-write "QUIT"
[1]+  Done                    psplash -n
```

2.2 压缩与解压缩应用

本节主要测试系统的解压缩工具。压缩是可以把多个文件压缩成一个压缩包可以把多个文件压缩成一个压缩包，方便进行文件的传输。而解压可以把经过压缩的压缩文件还原成原始大小方便使用。本节将在文件系统以 tar、gzip、gunzip 等工具为例进行说明。

2.2.1 Tar 压缩解压缩工具

1 tar 工具

现在我们在 Linux 中使用的 tar 工具，它不仅可以对文件打包，还可以对其进行压缩，查看，添加以及解压等一系列操作。这里是将打包操作。

语法格式

输入以下命令查看 tar 语法格式：

```
[root@100ask:~]# tar --help
Usage: tar [OPTION...] [FILE]...
GNU 'tar' saves many files together into a single tape or disk archive, and can
restore individual files from the archive.

Examples:
  tar -cf archive.tar foo bar  # Create archive.tar from files foo and bar.
  tar -tvf archive.tar         # List all files in archive.tar verbosely.
  tar -xf archive.tar          # Extract all files from archive.tar.
```

详细参数说明如下：

- c ：建立一个压缩文件的参数指令(create 的意思)；
- x ：解开一个压缩文件的参数指令！
- t ：查看 tarfile 里面的文件！特别注意，在参数的下达中， c/x/t 仅能存在一个！不可同一时候存在！由于不可能同一时候压缩与解压缩。
- z ：是否同一时候具有 gzip 的属性？亦即是否须要用 gzip 压缩？
- j ：是否同一时候具有 bzip2 的属性？亦即是否须要用 bzip2 压缩？
- -v ：压缩的过程中显示文件！这个经常使用，但不建议用在背景运行过程！
- -f ：使用档名，请留意，在 f 之后要马上接档名，不要再加参数！比如使用『tar -zcvfP tfile sfile』就是错误的写法，要写成『 tar -zcvPf tfile sfile』才对。
- -p ：使用原文件的原来属性（属性不会根据使用者而变）

- -P ：能够使用绝对路径来压缩！
- -N ：比后面接的日期(yyyy/mm/dd)还要新的才会被打包进新建的文件里！
- --exclude FILE：在压缩的过程中，不要将 FILE 打包！

2 使用 tar 压缩

新建 test.txt 文件，并输入以下命令将文件打包成.gz 格式：

```
[root@100ask:~]# tar -czf test.tar.gz test.txt
[root@100ask:~]# ls
test.txt
```

所以上面加 z 参数，则以 .tar.gz 或 .tgz 来代表 gzip 压缩过的 tar file 。

3 使用 tar 解压

把打包成 tar.gz 格式文件解压

```
[root@100ask:~]# tar -xvf test.tar.gz
test.txt
[root@100ask:~]# ls
test.c
```

2.2.2 压缩工具

gzip 是在 Linux 系统中经常使用的一个对文件进行压缩和解压缩的命令，既方便又好用。

1 语法格式

在开发板终端输入以下命令查看 gzip 语法：

```
[root@100ask:~]# gzip --help
BusyBox v1.31.1 () multi-call binary.
Usage: gzip [-cfkdt] [FILE]...
```

2 gzip 把文件压缩

```
[root@100ask:~]# gzip test.txt
[root@100ask:~]# ls
test.c
```

3 用 gunzip 工具解压

用 gunzip 把文件解压，示例如下：

```
[root@100ask:~]# gunzip test.txt.gz
[root@100ask:~]# ls
test.c
```

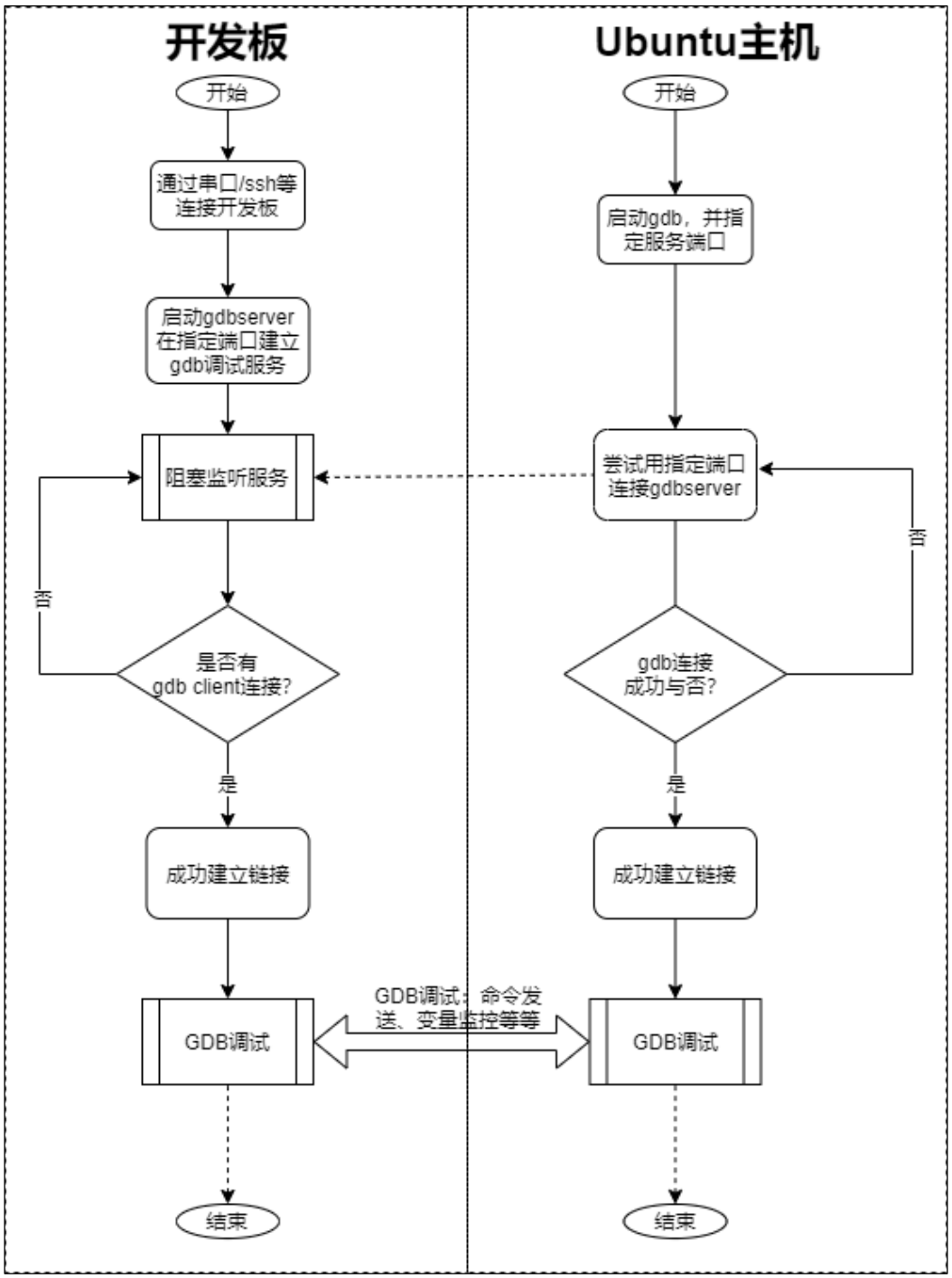
2.3 调试、性能分析、基准测试应用

2.3.1 远程调试 gdbserver

在板子上需满足有 gcc 编译器、gdb 调试器等，但一般对于嵌入式系统来说，在资源有限情况下，gcc 编译器及环境一般是在安装 X86 Linux 主机上。那么

此时，编译环境与程序运行环境就分开了，如果此时要进行 GDB 调试时，就需要借助 ARM Linux 上的 GDBServer 实现了。即通过网络建立 server 与 client，进而实现 gdb 调试。

GDB Client 及 GDBServer 就构成了 ARM Linux 的调试环境，其框架如下图所示：



1 准备程序

我们使用 C 语言程序来演示一下，其代码内容如下：

```
/*********************************测试程序*********************************/
```

```
#include <stdio.h>
//实现 a+b
 int add(int a, int b)
 {
     return a + b;
 }
//实现 a-b
 int sub(int a, int b)
 {
     return a - b;
 }

 int main()
 {
     printf(" 2 + 3 = %d\n", add(2, 3));
     printf(" 2 - 3 = %d\n", sub(2, 3));
     return 1;
 }
```

接下来，使用命令将其编译，-g 表示可以调试，命令如下：

```
book@100ask:~$ arm-buildroot-linux-gnueabihf-gcc test.c -g -o test
```

2 启动 gdb

根据前面小节的准备，现需将编译生成的<test>程序拷贝到开发板上，拷贝的方法有很多种，100ASK 提供的虚拟机已经配置了 NFS、SSH 等多种服务，这里以 ssh 为例，将<test>程序拷贝开发板上，已知开发板 IP 为 192.168.0.8，使用命令：

```
book@100ask:~$ scp test root@192.168.0.8:/root
```

将<test>程序拷贝到开发板/root 目录下，效果如下所示：

```
:~$ scp test root@192.168.0.8:/root
The authenticity of host '192.168.0.8 (192.168.0.8)' can't be established.
ECDSA key fingerprint is SHA256:VSD9GX1xJkGP8ZGlTpgaoGyvTIEWAC+hZ4FzL7L6QbQ.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.8' (ECDSA) to the list of known hosts.
test
```

切换到开发板终端，并通过 gdbserver 在指定端口<12345>建立服务，同时阻塞进行监听，如下所示：

```
[root@100ask:~]# gdbserver :12345 test
```

```
[root@100ask:~]# ls
test
[root@100ask:~]# gdbserver :12345 test
Process /root/test created; pid = 622
Listening on port 12345
```

3 主机连接 gdbserver

根据前面第 1、2 小节的准备，并将编译生成的<test>程序拷贝到开发板/mnt 目录下，并成功在开发板的 12345 端口建立了监听服务，此时，需启动主机的 gdb 程序以进行连接，命令如下所示：

```
book@100ask:~$ arm-buildroot-linux-gnueabihf-gdb  test
```

此时，当出现以下提示时：

```
Reading symbols from test...done.
```

输入命令：

```
target remote 192.168.0.8:12345
```

表示连接至远程 GDB 服务 192.168.0.8，端口 12345；回车之后，可以看到主机已经成功连接上了，如下图所示：

```
book@virtual-machine:~$ arm-buildroot-linux-gnueabihf-gdb  test
GNU gdb (GDB) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=arm-buildroot-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...done.
(gdb) target remote 192.168.0.8:12345
Remote debugging using 192.168.0.8:12345
Reading /lib/ld-linux-armhf.so.3 from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading symbols from target:/lib/ld-linux-armhf.so.3...(no debugging symbols found)...done.
0x76fced00 in _start () from target:/lib/ld-linux-armhf.so.3
```

4 开始调试

经过第 3 小节的操作，我们已经使得主机与开发板通过网络实现远程 GDB 连接，此时，可以在主机中输入 gdb 调试指令以实现远程调试，以一下的几个命令进行演示，如下所示分别 b 设置断点 info breakpoint 查看设置断点 c 执行程序：

```
(gdb) target remote 192.168.0.8:12345
Remote debugging using 192.168.0.8:12345
Reading /lib/ld-linux-armhf.so.3 from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading symbols from target:/lib/ld-linux-armhf.so.3...(no debugging symbols found)...done.
0x76fced00 in _start () from target:/lib/ld-linux-armhf.so.3
(gdb) b 5
Breakpoint 1 at 0x10424: file test.c, line 6.
(gdb) b 16
Breakpoint 2 at 0x10478: file test.c, line 16.
(gdb) b 17
Breakpoint 3 at 0x10498: file test.c, line 17.
(gdb) info breakpoint
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010424 in add at test.c:6
2       breakpoint     keep y   0x00010478 in main at test.c:16
3       breakpoint     keep y   0x00010498 in main at test.c:17
(gdb) c
Continuing.
Reading /lib/libc.so.6 from remote target...

Breakpoint 2, main () at test.c:16
16          printf(" 2 + 3 = %d\n", add(2, 3));
(gdb) c
Continuing.

Breakpoint 1, add (a=2, b=3) at test.c:6
6           return a + b;
(gdb) c
Continuing.

Breakpoint 3, main () at test.c:17
17          printf(" 2 - 3 = %d\n", sub(2, 3));
(gdb) c
Continuing.
[Inferior 1 (process 680) exited with code 01]
(gdb)
```

5 附录　GDB 常用命令参考

表 2-1 GDB 常用命令

命令	简写形式	说明
backtrace	bt、where	显示 backtrace
break		设置断点
continue	c、cont	继续运行

delete	d	删除断点
finish		运行到函数结束
info breakpoints		显示断点信息
next	n	执行下一行
print	p	显示表达式
run	r	运行程序
step	s	一次执行一行，包括函数内部
x		显示内存内容
until	u	执行到指定行
directory	dir	插入目录
disable	dis	禁用断点
down	do	在当前调用的栈帧中选择要显示的栈帧
edit	e	编辑文件或函数
frame	f	选择要显示的栈帧
forward-search	fo	向前搜索
generate-core-file	gcore	生成内核转储
help	h	显示帮助一览
info	i	显示信息
list	l	显示函数或行
nexti	ni	执行下一行（以汇编代码为单位）
print-object	po	显示目标信息
sharedlibrary	share	加载共享库的符号
setpi	si	执行下一行

2.3.2 查看系统调用 strace

Linux 中的 strace 命令可以帮助您查看程序系统调用和信号。

1 使用 strace 查看 ls 命令的系统调用过程。

```
[root@100ask:/]# strace ls
execve("/bin/ls", ["ls"], 0x7ea3fdc0 /* 16 vars */) = 0
brk(NULL)                               = 0x951000
uname({sysname="Linux", nodename="100ask", ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76f4700
0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENOENT (No
 such file or directory)
openat(AT_FDCWD, "/lib/tls/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOE
XEC) = -1 ENOENT (No such file or directory)
stat64("/lib/tls/v7l/neon/vfp", 0x7e9a41f8) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory)
stat64("/lib/tls/v7l/neon", 0x7e9a41f8) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
= -1 ENOENT (No such file or directory)
stat64("/lib/tls/v7l/vfp", 0x7e9a41f8)  = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory)
stat64("/lib/tls/v7l", 0x7e9a41f8)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
```

```
 = -1 ENOENT (No such file or directory)
stat64("/lib/tls/neon/vfp", 0x7e9a41f8) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory)
stat64("/lib/tls/neon", 0x7e9a41f8)     = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory)
stat64("/lib/tls/vfp", 0x7e9a41f8)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/tls/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory)
stat64("/lib/tls", 0x7e9a41f8)          = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory)
stat64("/lib/v7l/neon/vfp", 0x7e9a41f8) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory)
stat64("/lib/v7l/neon", 0x7e9a41f8)     = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory)
stat64("/lib/v7l/vfp", 0x7e9a41f8)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory)
stat64("/lib/v7l", 0x7e9a41f8)          = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory)
stat64("/lib/neon/vfp", 0x7e9a41f8)     = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 EN
OENT (No such file or directory)
stat64("/lib/neon", 0x7e9a41f8)         = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory)
stat64("/lib/vfp", 0x7e9a41f8)          = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\260#\0\0004\0\0\0"..., 512)
= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=67220, ...}) = 0
mmap2(NULL, 141216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76ef600
0
mprotect(0x76f06000, 61440, PROT_NONE)  = 0
mmap2(0x76f15000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0xf000) = 0x76f15000
mmap2(0x76f17000, 6048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0x76f17000
close(3)                                = 0
openat(AT_FDCWD, "/lib/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\234w\1\0004\0\0\0"..., 512)
= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1241860, ...}) = 0
mmap2(NULL, 1311420, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76db50
00
mprotect(0x76ee0000, 65536, PROT_NONE)  = 0
mmap2(0x76ef0000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x12b000) = 0x76ef0000
mmap2(0x76ef3000, 8892, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0x76ef3000
close(3)                                = 0
```

```
set_tls(0x76f48060)                     = 0
mprotect(0x76ef0000, 8192, PROT_READ)   = 0
mprotect(0x76f15000, 4096, PROT_READ)   = 0
mprotect(0xd4000, 4096, PROT_READ)      = 0
mprotect(0x76f49000, 4096, PROT_READ)   = 0
getuid32()                              = 0
gettimeofday({tv_sec=6434, tv_usec=363916}, NULL) = 0
ioctl(0, TIOCGWINSZ, {ws_row=84, ws_col=137, ws_xpixel=0, ws_ypixel=0}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x951000
brk(0x972000)                           = 0x972000
stat64(".", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_CLOEXEC|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
getdents64(3, /* 24 entries */, 32768)  = 632
lstat64("./mnt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./media", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./boot", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./run", {st_mode=S_IFDIR|0755, st_size=340, ...}) = 0
lstat64("./bin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./lib32", {st_mode=S_IFLNK|0777, st_size=3, ...}) = 0
lstat64("./root", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
lstat64("./sys", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
lstat64("./linuxrc", {st_mode=S_IFLNK|0777, st_size=11, ...}) = 0
lstat64("./etc", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=260, ...}) = 0
lstat64("./lost+found", {st_mode=S_IFDIR|0700, st_size=16384, ...}) = 0
lstat64("./usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./proc", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
lstat64("./var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./sbin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./opt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat64("./dev", {st_mode=S_IFDIR|0755, st_size=3440, ...}) = 0
getdents64(3, /* 0 entries */, 32768)   = 0
close(3)                                = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0xcf, 0x10), ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
write(1, "\33[1;34mbin\33[m        \33[1;34mdev"..., 212bin        dev         lib
     linuxrc     media       opt         root        sbin        tmp         var
) = 212
write(1, "\33[1;34mboot\33[m       \33[1;34metc"..., 190boot       etc         lib32
      lost+found  mnt         proc        run         sys         usr
) = 190
exit_group(0)                           = ?
+++ exited with 0 +++
```

2 使用**-i** 命令打印调用过程的栈指针地址。

```
[root@100ask:/]# strace -i ls
[76ee0b1c] execve("/bin/ls", ["ls"], 0x7eee0db4 /* 16 vars */) = 0
[76f3d788] brk(NULL)                     = 0x14c2000
[76f3ed2c] uname({sysname="Linux", nodename="100ask", ...}) = 0
[76f3ec44] mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
 = 0x76f53000
[76f3e80c] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
```

```
[76f3ea10] openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGE
FILE|O_CLOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/v7l/neon/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or d
irectory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE
|O_CLOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/v7l/neon", 0x7ebb41f8) = -1 ENOENT (No such file or direc
tory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|
O_CLOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/v7l/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or direct
ory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CL
OEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/v7l", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE
|O_CLOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/neon/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or direc
tory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_C
LOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/neon", 0x7ebb41f8) = -1 ENOENT (No such file or director
y)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CL
OEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/tls/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXE
C) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/tls", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE
|O_CLOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/v7l/neon/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or direc
tory)
[76f3ea10] openat(AT_FDCWD, "/lib/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_C
LOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/v7l/neon", 0x7ebb41f8) = -1 ENOENT (No such file or director
y)
[76f3ea10] openat(AT_FDCWD, "/lib/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CL
OEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/v7l/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXE
C) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/v7l", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_C
LOEXEC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/neon/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or director
y)
[76f3ea10] openat(AT_FDCWD, "/lib/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEX
EC) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/neon", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXE
C) = -1 ENOENT (No such file or directory)
[76f3e640] stat64("/lib/vfp", 0x7ebb41f8) = -1 ENOENT (No such file or directory)
[76f3ea10] openat(AT_FDCWD, "/lib/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) =
```

```
 3
[76f3eaf0] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\260#\0\0004\0\0\0
"..., 512) = 512
[76f3e684] fstat64(3, {st_mode=S_IFREG|0755, st_size=67220, ...}) = 0
[76f3ec44] mmap2(NULL, 141216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
 = 0x76f02000
[76f3ecec] mprotect(0x76f12000, 61440, PROT_NONE) = 0
[76f3ec44] mmap2(0x76f21000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_D
ENYWRITE, 3, 0xf000) = 0x76f21000
[76f3ec44] mmap2(0x76f23000, 6048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_A
NONYMOUS, -1, 0) = 0x76f23000
[76f3e848] close(3)                     = 0
[76f3ea10] openat(AT_FDCWD, "/lib/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
[76f3eaf0] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\234w\1\0004\0\0\0
"..., 512) = 512
[76f3e684] fstat64(3, {st_mode=S_IFREG|0755, st_size=1241860, ...}) = 0
[76f3ec44] mmap2(NULL, 1311420, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x76dc1000
[76f3ecec] mprotect(0x76eec000, 65536, PROT_NONE) = 0
[76f3ec44] mmap2(0x76efc000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
DENYWRITE, 3, 0x12b000) = 0x76efc000
[76f3ec44] mmap2(0x76eff000, 8892, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_A
NONYMOUS, -1, 0) = 0x76eff000
[76f3e848] close(3)                     = 0
[76f25b88] set_tls(0x76f54060)          = 0
[76f3ecec] mprotect(0x76efc000, 8192, PROT_READ) = 0
[76f3ecec] mprotect(0x76f21000, 4096, PROT_READ) = 0
[76f3ecec] mprotect(0xd4000, 4096, PROT_READ) = 0
[76f3ecec] mprotect(0x76f55000, 4096, PROT_READ) = 0
[76e5e29c] getuid32()                   = 0
[76e4c8e0] gettimeofday({tv_sec=6476, tv_usec=495220}, NULL) = 0
[76e8a46c] ioctl(0, TIOCGWINSZ, {ws_row=84, ws_col=137, ws_xpixel=0, ws_ypixel=0}) =
0
[76e89ab4] ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
[76e89ab4] ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
[76e8a2f4] brk(NULL)                    = 0x14c2000
[76e8a2f4] brk(0x14e3000)               = 0x14e3000
[76e80cb4] stat64(".", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e893dc] openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_CLOEXEC|O_DIRECTO
RY) = 3
[76e80cfc] fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e58c30] getdents64(3, /* 24 entries */, 32768) = 632
[76e80d44] lstat64("./mnt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./media", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./boot", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./run", {st_mode=S_IFDIR|0755, st_size=340, ...}) = 0
[76e80d44] lstat64("./bin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./lib32", {st_mode=S_IFLNK|0777, st_size=3, ...}) = 0
[76e80d44] lstat64("./root", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
[76e80d44] lstat64("./sys", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
[76e80d44] lstat64("./linuxrc", {st_mode=S_IFLNK|0777, st_size=11, ...}) = 0
[76e80d44] lstat64("./etc", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=260, ...}) = 0
[76e80d44] lstat64("./lost+found", {st_mode=S_IFDIR|0700, st_size=16384, ...}) = 0
[76e80d44] lstat64("./usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
```

```
[76e80d44] lstat64("./proc", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
[76e80d44] lstat64("./var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./sbin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./opt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
[76e80d44] lstat64("./dev", {st_mode=S_IFDIR|0755, st_size=3440, ...}) = 0
[76e58c30] getdents64(3, /* 0 entries */, 32768) = 0
[76e89254] close(3)                     = 0
[76e80cfc] fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0xcf, 0x10), ...}) = 0
[76e89ab4] ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
[76e819b8] write(1, "\33[1;34mbin\33[m        \33[1;34mdev"..., 212bin        dev
     lib        linuxrc     media       opt         root        sbin        tmp
 var
) = 212
[76e819b8] write(1, "\33[1;34mboot\33[m       \33[1;34metc"..., 190boot       etc
     lib32      lost+found  mnt         proc        run         sys         usr
) = 190
[76e5dac4] exit_group(0)                = ?
[????????] +++ exited with 0 +++
```

3 使用 **-t** 命令使每一行都以系统时间开始显示。

```
[root@100ask:/]# strace -t ls
01:48:35 execve("/bin/ls", ["ls"], 0x7e8f6db4 /* 16 vars */) = 0
01:48:35 brk(NULL)                      = 0x16de000
01:48:35 uname({sysname="Linux", nodename="100ask", ...}) = 0
01:48:35 mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x76f75000
01:48:35 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 E
NOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/tls/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFI
LE|O_CLOEXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/v7l/neon/vfp", 0x7eac31f8) = -1 ENOENT (No such file or dir
ectory)
01:48:35 openat(AT_FDCWD, "/lib/tls/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O
_CLOEXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/v7l/neon", 0x7eac31f8) = -1 ENOENT (No such file or directo
ry)
01:48:35 openat(AT_FDCWD, "/lib/tls/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_
CLOEXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/v7l/vfp", 0x7eac31f8) = -1 ENOENT (No such file or director
y)
01:48:35 openat(AT_FDCWD, "/lib/tls/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOE
XEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/v7l", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/tls/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O
_CLOEXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/neon/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directo
ry)
01:48:35 openat(AT_FDCWD, "/lib/tls/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLO
EXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/neon", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/tls/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOE
XEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/tls/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/tls/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory)
```

```
01:48:35 stat64("/lib/tls", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O
_CLOEXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/v7l/neon/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directo
ry)
01:48:35 openat(AT_FDCWD, "/lib/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLO
EXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/v7l/neon", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOE
XEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/v7l/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/v7l", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLO
EXEC) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/neon/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXE
C) = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/neon", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory)
01:48:35 stat64("/lib/vfp", 0x7eac31f8) = -1 ENOENT (No such file or directory)
01:48:35 openat(AT_FDCWD, "/lib/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
01:48:35 read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\260#\0\0004\0\0\0
"..., 512) = 512
01:48:35 fstat64(3, {st_mode=S_IFREG|0755, st_size=67220, ...}) = 0
01:48:35 mmap2(NULL, 141216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x76f24000
01:48:35 mprotect(0x76f34000, 61440, PROT_NONE) = 0
01:48:35 mmap2(0x76f43000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DEN
YWRITE, 3, 0xf000) = 0x76f43000
01:48:35 mmap2(0x76f45000, 6048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANO
NYMOUS, -1, 0) = 0x76f45000
01:48:35 close(3)                       = 0
01:48:35 openat(AT_FDCWD, "/lib/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
01:48:35 read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\234w\1\0004\0\0\0
"..., 512) = 512
01:48:35 fstat64(3, {st_mode=S_IFREG|0755, st_size=1241860, ...}) = 0
01:48:35 mmap2(NULL, 1311420, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
 0x76de3000
01:48:35 mprotect(0x76f0e000, 65536, PROT_NONE) = 0
01:48:35 mmap2(0x76f1e000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DE
NYWRITE, 3, 0x12b000) = 0x76f1e000
01:48:35 mmap2(0x76f21000, 8892, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANO
NYMOUS, -1, 0) = 0x76f21000
01:48:35 close(3)                       = 0
01:48:35 set_tls(0x76f76060)             = 0
01:48:35 mprotect(0x76f1e000, 8192, PROT_READ) = 0
01:48:35 mprotect(0x76f43000, 4096, PROT_READ) = 0
01:48:35 mprotect(0xd4000, 4096, PROT_READ) = 0
01:48:35 mprotect(0x76f77000, 4096, PROT_READ) = 0
01:48:35 getuid32()                     = 0
01:48:35 gettimeofday({tv_sec=6515, tv_usec=490797}, NULL) = 0
01:48:35 ioctl(0, TIOCGWINSZ, {ws_row=84, ws_col=137, ws_xpixel=0, ws_ypixel=0}) = 0
01:48:35 ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
```

```
01:48:35 ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
01:48:35 brk(NULL)                      = 0x16de000
01:48:35 brk(0x16ff000)                 = 0x16ff000
01:48:35 stat64(".", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_CLOEXEC|O_DIRECTOR
Y) = 3
01:48:35 fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 getdents64(3, /* 24 entries */, 32768) = 632
01:48:35 lstat64("./mnt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./media", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./boot", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./run", {st_mode=S_IFDIR|0755, st_size=340, ...}) = 0
01:48:35 lstat64("./bin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./lib32", {st_mode=S_IFLNK|0777, st_size=3, ...}) = 0
01:48:35 lstat64("./root", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
01:48:35 lstat64("./sys", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
01:48:35 lstat64("./linuxrc", {st_mode=S_IFLNK|0777, st_size=11, ...}) = 0
01:48:35 lstat64("./etc", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=260, ...}) = 0
01:48:35 lstat64("./lost+found", {st_mode=S_IFDIR|0700, st_size=16384, ...}) = 0
01:48:35 lstat64("./usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./proc", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
01:48:35 lstat64("./var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./sbin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./opt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
01:48:35 lstat64("./dev", {st_mode=S_IFDIR|0755, st_size=3440, ...}) = 0
01:48:35 getdents64(3, /* 0 entries */, 32768) = 0
01:48:35 close(3)                       = 0
01:48:35 fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0xcf, 0x10), ...}) = 0
01:48:35 ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
01:48:35 write(1, "\33[1;34mbin\33[m        \33[1;34mdev"..., 212bin        dev
    lib        linuxrc     media       opt         root        sbin        tmp         v
ar
) = 212
01:48:35 write(1, "\33[1;34mboot\33[m       \33[1;34metc"..., 190boot       etc
    lib32       lost+found  mnt         proc        run         sys         usr
) = 190
01:48:35 exit_group(0)                  = ?
01:48:35 +++ exited with 0 +++
```

4 使用**-T** 参数来显示 **ls** 命令系统调用花费的时间，时间显示在每行最后的**<>**内。

```
[root@100ask:/]# strace -T ls
execve("/bin/ls", ["ls"], 0x7ee5edb4 /* 16 vars */) = 0 <0.003837>
brk(NULL)                               = 0x1e82000 <0.000196>
uname({sysname="Linux", nodename="100ask", ...}) = 0 <0.000207>
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76f1800
0 <0.000257>
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory) <0.00
0326>
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENOENT (No
 such file or directory) <0.000321>
openat(AT_FDCWD, "/lib/tls/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOE
XEC) = -1 ENOENT (No such file or directory) <0.000289>
```

```
stat64("/lib/tls/v7l/neon/vfp", 0x7e8e31f8) = -1 ENOENT (No such file or directory) <
0.000257>
openat(AT_FDCWD, "/lib/tls/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory) <0.000263>
stat64("/lib/tls/v7l/neon", 0x7e8e31f8) = -1 ENOENT (No such file or directory) <0.00
0313>
openat(AT_FDCWD, "/lib/tls/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
= -1 ENOENT (No such file or directory) <0.000260>
stat64("/lib/tls/v7l/vfp", 0x7e8e31f8)  = -1 ENOENT (No such file or directory) <0.00
0220>
openat(AT_FDCWD, "/lib/tls/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory) <0.000324>
stat64("/lib/tls/v7l", 0x7e8e31f8)      = -1 ENOENT (No such file or directory) <0.00
0216>
openat(AT_FDCWD, "/lib/tls/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory) <0.000256>
stat64("/lib/tls/neon/vfp", 0x7e8e31f8) = -1 ENOENT (No such file or directory) <0.00
0217>
openat(AT_FDCWD, "/lib/tls/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory) <0.000258>
stat64("/lib/tls/neon", 0x7e8e31f8)     = -1 ENOENT (No such file or directory) <0.00
0212>
openat(AT_FDCWD, "/lib/tls/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory) <0.000249>
stat64("/lib/tls/vfp", 0x7e8e31f8)      = -1 ENOENT (No such file or directory) <0.00
0122>
openat(AT_FDCWD, "/lib/tls/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory) <0.000135>
stat64("/lib/tls", 0x7e8e31f8)          = -1 ENOENT (No such file or directory) <0.000
099>
openat(AT_FDCWD, "/lib/v7l/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC)
 = -1 ENOENT (No such file or directory) <0.000118>
stat64("/lib/v7l/neon/vfp", 0x7e8e31f8) = -1 ENOENT (No such file or directory) <0.00
0091>
openat(AT_FDCWD, "/lib/v7l/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory) <0.000115>
stat64("/lib/v7l/neon", 0x7e8e31f8)     = -1 ENOENT (No such file or directory) <0.00
0088>
openat(AT_FDCWD, "/lib/v7l/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1
 ENOENT (No such file or directory) <0.000103>
stat64("/lib/v7l/vfp", 0x7e8e31f8)      = -1 ENOENT (No such file or directory) <0.00
0094>
openat(AT_FDCWD, "/lib/v7l/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory) <0.000104>
stat64("/lib/v7l", 0x7e8e31f8)          = -1 ENOENT (No such file or directory) <0.000
086>
openat(AT_FDCWD, "/lib/neon/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -
1 ENOENT (No such file or directory) <0.000106>
stat64("/lib/neon/vfp", 0x7e8e31f8)     = -1 ENOENT (No such file or directory) <0.00
0133>
openat(AT_FDCWD, "/lib/neon/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 EN
OENT (No such file or directory) <0.000104>
stat64("/lib/neon", 0x7e8e31f8)         = -1 ENOENT (No such file or directory) <0.000
093>
openat(AT_FDCWD, "/lib/vfp/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENO
ENT (No such file or directory) <0.000101>
```

```
stat64("/lib/vfp", 0x7e8e31f8)          = -1 ENOENT (No such file or directory) <0.000
083>
openat(AT_FDCWD, "/lib/libresolv.so.2", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3 <0.00022
2>
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\260#\0\0004\0\0\0"..., 512)
= 512 <0.000129>
fstat64(3, {st_mode=S_IFREG|0755, st_size=67220, ...}) = 0 <0.000071>
mmap2(NULL, 141216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76ec700
0 <0.000232>
mprotect(0x76ed7000, 61440, PROT_NONE)  = 0 <0.000117>
mmap2(0x76ee6000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0xf000) = 0x76ee6000 <0.000182>
mmap2(0x76ee8000, 6048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0x76ee8000 <0.000403>
close(3)                                = 0 <0.000193>
openat(AT_FDCWD, "/lib/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3 <0.000550>
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\234w\1\0004\0\0\0"..., 512)
= 512 <0.000292>
fstat64(3, {st_mode=S_IFREG|0755, st_size=1241860, ...}) = 0 <0.000188>
mmap2(NULL, 1311420, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76d860
00 <0.000249>
mprotect(0x76eb1000, 65536, PROT_NONE)  = 0 <0.000264>
mmap2(0x76ec1000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x12b000) = 0x76ec1000 <0.000369>
mmap2(0x76ec4000, 8892, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0x76ec4000 <0.000328>
close(3)                               = 0 <0.000187>
set_tls(0x76f19060)                     = 0 <0.000190>
mprotect(0x76ec1000, 8192, PROT_READ)   = 0 <0.000296>
mprotect(0x76ee6000, 4096, PROT_READ)   = 0 <0.000255>
mprotect(0xd4000, 4096, PROT_READ)      = 0 <0.000236>
mprotect(0x76f1a000, 4096, PROT_READ)   = 0 <0.000243>
getuid32()                              = 0 <0.000428>
gettimeofday({tv_sec=6556, tv_usec=889804}, NULL) = 0 <0.000211>
ioctl(0, TIOCGWINSZ, {ws_row=84, ws_col=137, ws_xpixel=0, ws_ypixel=0}) = 0 <0.00021
6>
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0 <0.000308>
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0 <0.000223>
brk(NULL)                               = 0x1e82000 <0.000171>
brk(0x1ea3000)                          = 0x1ea3000 <0.000212>
stat64(".", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000267>
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_CLOEXEC|O_DIRECTORY) = 3 <0.
000285>
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000186>
getdents64(3, /* 24 entries */, 32768)  = 632 <0.000504>
lstat64("./mnt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000283>
lstat64("./media", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000249>
lstat64("./boot", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000251>
lstat64("./lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000247>
lstat64("./run", {st_mode=S_IFDIR|0755, st_size=340, ...}) = 0 <0.000248>
lstat64("./bin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000240>
lstat64("./lib32", {st_mode=S_IFLNK|0777, st_size=3, ...}) = 0 <0.000246>
lstat64("./root", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0 <0.000232>
lstat64("./sys", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0 <0.000248>
lstat64("./linuxrc", {st_mode=S_IFLNK|0777, st_size=11, ...}) = 0 <0.000260>
lstat64("./etc", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000248>
```

```
lstat64("./tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=260, ...}) = 0 <0.000247>
lstat64("./lost+found", {st_mode=S_IFDIR|0700, st_size=16384, ...}) = 0 <0.000244>
lstat64("./usr", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000237>
lstat64("./proc", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0 <0.000244>
lstat64("./var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000238>
lstat64("./sbin", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000295>
lstat64("./opt", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0 <0.000228>
lstat64("./dev", {st_mode=S_IFDIR|0755, st_size=3440, ...}) = 0 <0.000232>
getdents64(3, /* 0 entries */, 32768)   = 0 <0.000259>
close(3)                                = 0 <0.000255>
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0xcf, 0x10), ...}) = 0 <0.000184>
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0 <0.000267>
write(1, "\33[1;34mbin\33[m        \33[1;34mdev"..., 212bin        dev        lib
      linuxrc     media       opt         root        sbin        tmp         var
) = 212 <0.000272>
write(1, "\33[1;34mboot\33[m       \33[1;34metc"..., 190boot       etc        lib32
      lost+found  mnt         proc        run         sys         usr
) = 190 <0.000252>
exit_group(0)                           = ?
+++ exited with 0 +++
```

5 使用**-c** 参数可以使 **strace** 打印摘要而不是通常的输出。

```
[root@100ask:/]# strace -c ls
bin         dev         lib         linuxrc     media       opt         root        sbin
      tmp         var
boot        etc         lib32       lost+found  mnt         proc        run         sys
      usr
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         2           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         3           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         4           ioctl
  0.00    0.000000           0         1           gettimeofday
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         6           mprotect
  0.00    0.000000           0         7           mmap2
  0.00    0.000000           0        16        15 stat64
  0.00    0.000000           0        19           lstat64
  0.00    0.000000           0         4           fstat64
  0.00    0.000000           0         1           getuid32
  0.00    0.000000           0         2           getdents64
  0.00    0.000000           0        19        16 openat
  0.00    0.000000           0         1           set_tls
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    93        32 total
```

6 使用**-o** 文件名参数可以使 **strace** 打印信息输出到某个文件内。

```
[root@100ask:/]# strace  -o log.txt -t ls
[root@100ask:/]# strace  -o log.txt -t ls
bin         dev         lib         linuxrc     lost+found  mnt         proc        run         sys         usr
boot        etc         lib32       log.txt     media       opt         root        sbin        tmp         var
[root@100ask:/]#
```

2.4 开发工具应用

2.4.1 Grep

Linux grep 命令用于查找文件里符合条件的字符串。
grep 指令用于查找内容包含指定的范本样式的文件，如果发现某文件的内容符合所指定的范本样式，预设 grep 指令会把含有范本样式的那一列显示出来。若不指定任何文件名称，或是所给予的文件名为 -，则 grep 指令会从标准输入设备读取数据。

1 语法

```
grep [-abcEFGhHilLnqrsvVwxy][-A<显示列数>][-B<显示列数>][-C<显示列数>][-d<进行动作>][-e<范本样式>][-f<范本文件>][--help][范本样式][文件或目录...]
```

参数：

- -a 或 --text ：不要忽略二进制的数据。
- -A<显示行数> 或 --after-context=<显示行数> ：除了显示符合范本样式的那一列之外，并显示该行之后的内容。
- -b 或 --byte-offset ：在显示符合样式的那一行之前，标示出该行第一个字符的编号。
- -B<显示行数> 或 --before-context=<显示行数> ：除了显示符合样式的那一行之外，并显示该行之前的内容。
- -c 或 --count ：计算符合样式的列数。
- -C<显示行数> 或 --context=<显示行数>或-<显示行数> ：除了显示符合样式的那一行之外，并显示该行之前后的内容。
- -d <动作> 或 --directories=<动作> ：当指定要查找的是目录而非文件时，必须使用这项参数，否则 grep 指令将回报信息并停止动作。
- -e<范本样式> 或 --regexp=<范本样式> ：指定字符串做为查找文件内容的样式。
- -E 或 --extended-regexp ：将样式为延伸的正则表达式来使用。
- -f<规则文件> 或 --file=<规则文件> ：指定规则文件，其内容含有一个或多个规则样式，让 grep 查找符合规则条件的文件内容，格式为每行一个规则样式。
- -F 或 --fixed-regexp ：将样式视为固定字符串的列表。
- -G 或 --basic-regexp ：将样式视为普通的表示法来使用。
- -h 或 --no-filename ：在显示符合样式的那一行之前，不标示该行所属的文件名称。
- -H 或 --with-filename ：在显示符合样式的那一行之前，表示该行所属的文件名称。
- -i 或 --ignore-case ：忽略字符大小写的差别。

- -l 或 --file-with-matches ： 列出文件内容符合指定的样式的文件名称。
- -L 或 --files-without-match ： 列出文件内容不符合指定的样式的文件名称。
- -n 或 --line-number ： 在显示符合样式的那一行之前，标示出该行的列数编号。
- -o 或 --only-matching ： 只显示匹配 PATTERN 部分。
- -q 或 --quiet 或--silent ： 不显示任何信息。
- -r 或 --recursive ： 此参数的效果和指定"-d recurse"参数相同。
- -s 或 --no-messages ： 不显示错误信息。
- -v 或 --invert-match ： 显示不包含匹配文本的所有行。
- -V 或 --version ： 显示版本信息。
- -w 或 --word-regexp ： 只显示全字符合的列。
- -x --line-regexp ： 只显示全列符合的列。
- -y ： 此参数的效果和指定"-i"参数相同。

2 示例

- 查找 log 文件内某个时间的所有记录

```
grep -n '2020-10-24 00:01:11' *.log
```

grep "被查找的字符串" 文件名

例子：在当前目录里第一级文件夹中寻找包含指定字符串的 .in 文件

```
grep "thermcontact" /.in
```

- 从文件内容查找与正则表达式匹配的行：

```
grep -e "正则表达式" 文件名
```

- 查找时不区分大小写：

```
grep -i "被查找的字符串" 文件名
```

- 查找匹配的行数：

```
grep -c "被查找的字符串" 文件名
```

- 从文件内容查找不匹配指定字符串的行：

```
grep -v "被查找的字符串" 文件名
```

- 从根目录开始查找所有扩展名为 .log 的文本文件，并找出包含 "ERROR" 的行：

```
find / -type f -name "*.log" | xargs grep "ERROR"
```

- 从当前目录开始查找所有扩展名为 .in 的文本文件，并找出包含 "thermcontact" 的行：

```
find . -name "*.in" | xargs grep "thermcontact"
```

参考阅读

https://www.gnu.org/software/grep/manual/grep.html
https://www.runoob.com/linux/linux-comm-grep.html
https://www.runoob.com/linux/linux-comm-grep.html

2.4.2 Gawk

awk 是一种编程语言，用于在 linux/unix 下对文本和数据进行处理。数据可以来自标准输入、一个或多个文件，或其它命令的输出（即管道）。它支持用户自定义函数和 动态正则表达式等先进功能，是 linux/unix 下的一个强大编程工具。它在命令行中使用，但更多是作为脚本来使用。

awk 的处理文本和数据的方式是这 样的，它逐行扫描文件，从第一行到最后一行，寻找匹配的特定模式的行，并在这些行上进行你想要的操作。如果没有指定处理动作，则把匹配的行显示到标准输出 (屏幕),即默认处理动作是 print；如果没有指定模式，则所有被操作所指定的行都被处理，即默认指定模式是全部。awk 分别代表其作者姓氏的第一个字母。因为它的作者是三个人，分别是 Alfred Aho、Brian Kernighan、Peter Weinberger。gawk 是 awk 的 GNU 版本，它提供了 Bell 实验室和 GNU 的一些扩展。

像 shell 一样，awk 也有好几种，常见的如 awk、nawk、mawk、gawk，其中

- gawk:是 GNU Project 的 awk 解释器的开放源代码实现。
- 尽管早期的 GAWK 发行版是旧的 AWK 的替代程序，但不断地对其进行了更新，以包含 NAWK 的特性;

目前，大家都比较倾向于使用 awk 和 gawk,本文中要介绍的 awk 是以 GUN 的 gawk 为例。

gawk 命令可用于：

- 逐行扫描文件。
- 将每个输入行拆分为字段。
- 比较输入行列/方向与样式。
- 在匹配的行上执行操作。
- 转换数据文件。
- 生成格式化的报告。
- 格式化输出行。
- 算术和字符串运算。
- 条件和循环。

1 语法格式

```
gawk [POSIX / GNU style options] -f progfile [--] file ...
gawk [POSIX / GNU style options] [--] 'program' file ...
```

一些重要的参数

- -f progfile, –file=progfile: 从文件 program-file 而不是从第一个命令行参数读取 AWK 程序源。可以使用多个-f（或–file）选项。
- -F fs, –field-separator=fs: 它将 FS 用作输入字段分隔符（FS 为预定义变量的值）。

- -v var=val, -assign=var=val: 在程序开始执行之前，将值 val 分配给变量 var。

2 示例

- -F：它将 FS 用作输入字段分隔符（FS 预定义变量的值）。

```
gawk -F: '{print $1}' /etc/passwd
```

```
book@100ask:~$ gawk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
```

- -f：从文件 program-file 而不是从第一个命令行参数读取 AWK 程序源。可以使用多个-f（或–file）选项。

```
gawk -F: -f test.txt /etc/passwd
```

参考资料

https://www.gnu.org/software/gawk/manual/gawk.html
https://www.geeksforgeeks.org/gawk-command-in-linux-with-examples/

2.4.3 Sed

sed 是流编辑器。它可以对文件执行很多功能，例如搜索，查找和替换，插入或删除。流编辑器用于对输入流（文件或来自管道的输入）执行基本的文本转换。尽管在某种程度上类似于允许脚本化编辑的编辑器（例如 ed），但 sed 仅对输入进行一次传递即可工作，因此效率更高。但这是 sed 在管道中过滤文本的能力，这使其与其他类型的编辑器特别有区别。

1 语法

```
sed [-hnV][-e<script>][-f<script 文件>][文本文件]
```

参数说明：

- -e<script>或--expression=<script> 以选项中指定的 script 来处理输入的文本文件。
- -f<script 文件>或--file=<script 文件> 以选项中指定的 script 文件来处理输入的文本文件。
- -h 或--help 显示帮助。
- -n 或--quiet 或--silent 仅显示 script 处理后的结果。
- -V 或--version 显示版本信息。

动作说明：

  - a ：新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～
  - c ：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！

- d ：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；
- i ：插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；
- p ：打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～
- s ：取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 1,20s/old/new/g 就是啦！

2 示例

通常这样使用 sed 命令：

```
sed SCRIPT INPUTFILE...
```

例如，在文件 input.txt 中替换所有出现的“hello”为“word” ，并重定向输出到 output.txt 文件内。

```
sed 's/hello/world/' input.txt > output.txt
```

- 如果未指定 INPUTFILE，或者如果 INPUTFILE 为 - ，sed 过滤标准输入的内容。以下命令是等效的：

```
sed 's/hello/world/' input.txt > output.txt
sed 's/hello/world/' < input.txt > output.txt
cat input.txt | sed 's/hello/world/' - > output.txt
```

- sed 将输出写入标准输出。用-可以就地编辑文件，而不是打印到标准输出。另请参见 W 和 s///w 命令，用于将输出写入其他文件。以下命令修改 file.txt 并且不产生任何输出：

```
sed -i 's/hello/world/' file.txt
```

- 默认情况下,sed打印所有已处理的输入(诸如命令 修改/删除的输入除外)。用 -n 禁止输出，使用 p 显示特定的行。以下命令仅打印输入文件的第 45 行：

```
sed -n '45p' file.txt
```

- sed 将多个输入文件视为一个长流。以下示例显示第一个文件的第一行（one.txt）和最后一个文件的最后一行（three.txt）。

```
sed -n  '1p ; $p' one.txt two.txt three.txt
```

不管是-e 或 -f 选项，sed 使用第一个非选项参数作为脚本，然后使用以下非选项参数作为输入文件。

如果使用 -e 命令或者 -f 选项用于指定脚本，并非所有的非选项参数均作为输入文件。

选项 -e 和 -d 可以组合，并且可以多次出现（在这种情况下，最终有效的脚本将是所有单个脚本的串联）。

以下示例是等效的：

```
sed 's/hello/world/' input.txt > output.txt

sed -e 's/hello/world/' input.txt > output.txt
sed --expression='s/hello/world/' input.txt > output.txt

echo 's/hello/world/' > myscript.sed
sed -f myscript.sed input.txt > output.txt
sed --file=myscript.sed input.txt > output.txt
```

扩展参考

GNU 官方 SED 命令手册：

https://www.gnu.org/software/sed/manual/html_node/sed-commands-list.html
https://www.gnu.org/software/sed/manual/sed.html

2.4.4 GNU Binutils 工具集

GNU Binutils 是二进制工具的集合。主要的是：

- ld -GNU 链接器。
- as -GNU 汇编器。

但它们还包括：

- addr2line-将地址转换为文件名和行号。
- ar-用于创建，修改和提取档案的实用程序。
- c ++ filt-过滤以解编码编码的 C ++符号。
- dlltool-创建用于构建和使用 DLL 的文件。
- gold-一个新的，更快的，仅 ELF 的链接器，仍处于 beta 测试中。
- gprof-显示分析信息。
- nlmconv-将目标代码转换为 NLM。
- nm-列出目标文件中的符号。
- objcopy-复制并转换目标文件。
- objdump-显示目标文件中的信息。
- ranlib-生成指向档案内容的索引。
- readelf-显示来自任何 ELF 格式对象文件的信息。
- size-列出的对象或归档文件的部分的尺寸。
- strings-列出文件中的可打印字符串。
- strip-丢弃的符号。
- windmc -Windows 兼容的消息编译器。
- windres -Windows 资源文件的编译器。

这些程序大多数使用 BFD（二进制文件描述符库）进行低级操作。他们中的许多人还使用操作码库来汇编和反汇编机器指令。

2.5 文件系统和闪存类应用

2.5.1 fdisk 分区工具使用

运行环境

- 支持的 INIT 服务： systemV systemD
- 支持的开发板系统：Buildroot Yocto busybox

1 fdisk 工具使用界面简介

```
[ root@100ask :~]# fdisk /dev/mmcblk1

The number of cylinders for this disk is set to 119296.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command Action
a       toggle a bootable flag
b       edit bsd disklabel
c       toggle the dos compatibility flag
d       delete a partition   <==删除一个分区
l       list known partition types
n       add a new partition  <==创建一个新的分区
o       create a new empty DOS partition table
p       print the partition table   <==显示所有的分区
q       quit without saving changes <==不保存配置并退出
s       create a new empty Sun disklabel
t       change a partition's system id
u       change display/entry units
v       verify the partition table
w       write table to disk and exit  <==保存分区表并退出程序
x       extra functionality (experts only)

Command (m for help):
```

2 查看当前系统内所有分区

使用 fdisk -l 列出系统下的所有磁盘设备分区信息，每个磁盘设备的提示信息意义为：

- Device：装置档名，依据不同的磁盘界面/分区位置而变。
- StartCHS,EndCHS：指的是 MBR 分区的开始和结束地址。
- Boot：是否为开机启动区块？通常 Windows 系统的 C 需要这块。
- Start， End：这个分区在哪个磁柱号码之间，可以决定分区的大小；
- Sectors：这里指的是此分区占用的扇区个数一共有多少个。
- Id，Type：分别代表文件系统代号，磁盘类型。

此时我们需要得知，不同的设备分区来自哪个磁盘设备，如下图所示,/dev/mmcblk0 为我插入的 8GB SD 卡设备，此时可以从下图中得知 此 sd 卡有两个分区信息，容量大小为 7560MB。

```
[root@ :~]# fdisk -l
Disk /dev/mmcblk1: 3728 MB, 3909091328 bytes, 7634944 sectors
119296 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk1p1    320,0,1     895,3,16         20480     122879     102400 50.0M  c Win95 FAT32 (LBA)
/dev/mmcblk1p2    896,0,1     511,3,16        122880    7634943    7512064 3668M 83 Linux
Disk /dev/mmcblk1boot1: 2 MB, 2097152 bytes, 4096 sectors
64 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Disk /dev/mmcblk1boot1 doesn't contain a valid partition table
Disk /dev/mmcblk1boot0: 2 MB, 2097152 bytes, 4096 sectors
64 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Disk /dev/mmcblk1boot0 doesn't contain a valid partition table
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
241920 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)
/dev/mmcblk0p2    1,86,22     523,128,53       21504    8410111    8388608 4096M 83 Linux
```

3 新增一个分区

注意：如下示例演示的 SD 卡设备节点为/dev/mmcblk0 你的可能会是/dev/mmcblk1 或者其它，请根据卡插入后终端提示来修改为相应的。

fdisk /dev/mmcblk0 ：先进入 fdisk 画面；

- p ：先看一下分区的信息，这里显示只有一个分区。
- n ：这个时候让你选择 primary partition(主分区) 还是 extended(扩展分区)，我们这里输入 p 选择主分区。
- 2 ：此时让你输入创建到第几个分区，这里直接输入 2， 输入成功后再次打印显示分区信息，显示已经有两个分区。
- w ：按 w 可将分区信息存储到分区表中，并离开 fdisk ；当然啰， 如果你反悔了，直接按下 q 就可以取消刚刚的删除动作,此时，我们需要格式化并挂载新的分区。

```
[root@j :~]# fdisk  /dev/mmcblk0

The number of cylinders for this disk is set to 8474.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
8474 cylinders, 87 heads, 21 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)

Command (m for help): n
Partition type
   p   primary partition (1-4)
   e   extended
```

从上图可知，我们的第二个分区设备为 /dev/mmcblk0p2 ，分区类型为 Linux ，此时我们可以用如下命令对其进行格式化，并挂载。

```
// 格式化此分区设备为 ext3 格式，如果提示是否强制继续，输入 y 即可
[root@100ask:~]# mkfs.ext3 /dev/mmcblk0p2
```

格式化完成后，需要将其挂载到相应的目录，才可对其进行操作，此时我们挂载的目录为 /mnt

```
[root@100ask:~]# mount  -t ext3 /dev/mmcblk0p2 /mnt
```

此时可以使用 df -Th 命令查看系统所有的挂载信息，来确认是否挂载成功以及分区的详细信息。

```
[root@imx6ull:~]# mkfs.ext3 /dev/mmcblk0p2
mke2fs 1.44.5 (15-Dec-2018)
/dev/mmcblk0p2 contains a ext4 file system
        last mounted on / on Thu Jan  1 00:00:04 1970
Proceed anyway? (y,N) y
Discarding device blocks: done
Creating filesystem with 1932672 4k blocks and 483328 inodes
Filesystem UUID: 2a4264b2-fb53-491a-af9e-fea7ad42c430
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

[root@imx6ull:~]# mount -t ext3 /dev/mmcblk0p2 /mnt
[ 1763.782460] EXT4-fs (mmcblk0p2): mounting ext3 file system using the ext4 subsystem
[ 1763.824605] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
[root@imx6ull:~]# df -h
Filesystem                Size      Used Available Use% Mounted on
/dev/root                 3.5G    341.1M      3.0G  10% /
devtmpfs                 85.3M         0     85.3M   0% /dev
tmpfs                   245.8M         0    245.8M   0% /dev/shm
tmpfs                   245.8M    680.0K    245.1M   0% /tmp
tmpfs                   245.8M    180.0K    245.6M   0% /run
/dev/mmcblk0p2            7.2G     16.6M      6.8G   0% /mnt
[root@imx6ull:~]#
```

4 删除一个分区

`fdisk /dev/mmcblk0` ： 先进入 fdisk 操作界面；

- p ： 先看一下分区的信息，这里显示只有一个分区。
- d ： 这时候让你选择删除那个分区，我们有两个分区就选择删除第 2 个分区好了，删除后，再次输入 p 来查看当前磁盘设备有几个分区。
- w ： 按 w 可将分区信息存储到分区表中，并离开 fdisk ； 当然啰， 如果你反悔了，直接按下 q 就可以取消刚刚的删除动作。

```
[root@imx6ull:~]# fdisk  /dev/mmcblk0

The number of cylinders for this disk is set to 241920.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
241920 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)
/dev/mmcblk0p2    1,86,22     523,128,53       21504    8410111    8388608 4096M 83 Linux

Command (m for help): d
Partition number (1-4): 2

Command (m for help): p
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
241920 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
[  766.790225]  mmcblk0: p1
[root@imx6ull:~]# [  766.832909]  mmcblk0: p1
```

5 修改某个分区的分区类型

`fdisk /dev/mmcblk0` ： 先进入 fdisk 操作界面；

- p ： 先看一下分区的信息，这里显示只有一个分区。
- n ： 这个时候让你选择 primary partition(主分区) 还是 extended(扩展分区)，我们这里输入 p 选择主分区。
- t ： 输入 t 表示要修改分区类型，然后输入需要修改的分区，这里选择第二个分区，
- L ： 列出所有的分区类型，此时输入我们将要修改成的分区类型，这里是要修改成 FAT32(LBA)分区类型，所以输入 c 。
- p ： 再次显示所有的分区类型，看是否已经更改。
- w ： 按 w 可将分区信息存储到分区表中，并离开 fdisk ； 当然啰， 如果你反悔了，直接按下 q 就可以取消刚刚的删除动作,此时，我们需要格式化并挂载新的分区。

```
[root@imx6ull:~]# fdisk  /dev/mmcblk0

The number of cylinders for this disk is set to 8474.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
8474 cylinders, 87 heads, 21 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)
/dev/mmcblk0p2    11,67,1     1023,86,21       21504   15482879   15461376 7549M 83 Linux

Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): L

 0 Empty                 1b Hidden Win95 FAT32     9f BSD/OS
 1 FAT12                 1c Hidden W95 FAT32 (LBA) a0 Thinkpad hibernation
 4 FAT16 <32M            1e Hidden W95 FAT16 (LBA) a5 FreeBSD
 5 Extended              3c Part.Magic recovery    a6 OpenBSD
 6 FAT16                 41 PPC PReP Boot          a8 Darwin UFS
 7 HPFS/NTFS             42 SFS                    a9 NetBSD
 a OS/2 Boot Manager     63 GNU HURD or SysV       ab Darwin boot
 b Win95 FAT32           80 Old Minix              b7 BSDI fs
 c Win95 FAT32 (LBA)     81 Minix / old Linux      b8 BSDI swap
 e Win95 FAT16 (LBA)     82 Linux swap             be Solaris boot
 f Win95 Ext'd (LBA)     83 Linux                  eb BeOS fs
11 Hidden FAT12          84 OS/2 hidden C: drive   ee EFI GPT
12 Compaq diagnostics    85 Linux extended         ef EFI (FAT-12/16/32)
14 Hidden FAT16 <32M     86 NTFS volume set        f0 Linux/PA-RISC boot
16 Hidden FAT16          87 NTFS volume set        f2 DOS secondary
17 Hidden HPFS/NTFS      8e Linux LVM              fd Linux raid autodetect
Hex code (type L to list codes): c
Changed system type of partition 2 to c (Win95 FAT32 (LBA))

Command (m for help): p
Disk /dev/mmcblk0: 7560 MB, 7927234560 bytes, 15482880 sectors
8474 cylinders, 87 heads, 21 sectors/track
Units: sectors of 1 * 512 = 512 bytes

Device       Boot StartCHS    EndCHS        StartLBA     EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1    0,16,17     1,86,21           1024      21503      20480 10.0M  c Win95 FAT32 (LBA)
/dev/mmcblk0p2    11,67,1     1023,86,21       21504   15482879   15461376 7549M  c Win95 FAT32 (LBA)

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
[ 1831.620429]  mmcblk0: p1 p2
[root@imx6ull:~]# [ 1831.669915]  mmcblk0: p1 p2
```

从上图可知，我们的第二个分区设备为 /dev/mmcblk0p2 ，分区类型为重新设置为 FAT32,此时我们可以用如下命令对其进行格式化，并挂载。

```
// 格式化此分区设备为 fat 格式，如果提示是否强制继续，输入 y 即可
[root@100ask:~]# mkfs.fat  /dev/mmcblk0p2
```

格式化完成后，需要将其挂载到相应的目录，才可对其进行操作，此时我们挂载的目录为 /mnt

```
[root@100ask:~]# mount  -t vfat /dev/mmcblk0p2 /mnt
```

此时可以使用 df -Th 命令查看系统所有的挂载信息，来确认是否挂载成功以及分区的详细信息。

```
[root@imx6ull:~]# [ 1831.669915]  mmcblk0: p1 p2

[root@imx6ull:~]# mkfs.fat /dev/mmcblk0p2
mkfs.fat 4.1 (2017-01-24)
[root@imx6ull:~]# mount -t vfat /dev/mmcblk0p2 /mnt
[root@imx6ull:~]# df -Th
Filesystem           Type            Size      Used Available Use% Mounted on
/dev/root            ext3            3.5G    341.1M      3.0G  10% /
devtmpfs             devtmpfs       85.3M         0     85.3M   0% /dev
tmpfs                tmpfs         245.8M         0    245.8M   0% /dev/shm
tmpfs                tmpfs         245.8M    688.0K    245.1M   0% /tmp
tmpfs                tmpfs         245.8M    180.0K    245.6M   0% /run
/dev/mmcblk0p2       vfat            7.4G      4.0K      7.4G   0% /mnt
[root@imx6ull:~]#
```

2.5.2 使用 resize2fs 命令调整 rootfs 分区大小

运行环境

- 支持的 INIT 服务：systemV systemD
- 支持的开发板系统：Buildroot Yocto busybox

在开发板执行如下命令来扩充 rootfs 分区容量大小，调整完成后执行 sync 同步缓存数据。

```
[root@100ask:~]# resize2fs  /dev/mmcblk2p3
[root@100ask:~]# sync
```

同步之后可以使用 df -h 命令来查看当前系统分区空间大小信息。

```
[root@100ask:~]# df -h
[root@100ask:~]# df -h
Filesystem               Size      Used Available Use% Mounted on
devtmpfs                 146.6M         0    146.6M   0% /dev
/dev/disk/by-partuuid/491f6117-415d-4f53-88c9-6e0de54deac6
                           3.6G    659.9M      2.7G  19% /
tmpfs                    212.8M         0    212.8M   0% /dev/shm
tmpfs                    212.8M    752.0K    212.1M   0% /run
tmpfs                    212.8M         0    212.8M   0% /sys/fs/cgroup
tmpfs                    212.8M         0    212.8M   0% /tmp
/dev/mmcblk2p2            14.5M     11.9M      1.5M  89% /boot
[root@100ask:~]#
```

调整后的分区大小

2.6 硬件处理类工具

2.6.1 设备调试工具 i2c-tools

i2c-tools 是一组 I²C 程序，无需编写任何代码即可轻松调试 I²C 设备，i2c-tools 有如下几个测试命令 i2cdetect，i2cdump，i2cget，i2cset。

1 用 i2cdetect 检测有几组 i2c 总线在系统上，输入：i2cdetect -l

```
[root@100ask:~]# i2cdetect -l
[root@100ask:~]# i2cdetect -l
i2c-1   i2c             21a4000.i2c                             I2C adapter
i2c-0   i2c             21a0000.i2c                             I2C adapter
[root@100ask:~]#
```

由上图可知，系统中存在两组总线分别 i2c-1 和 i2c-0。

2 用 i2cdetect 检测挂载在 i2c 总线上器件，输入 i2cdetect -r -y 0（检测 i2c-0 的挂载情况）。

```
[root@100ask:~]# i2cdetect -r -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 1e --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
[root@100ask:~]#
```

3 用 i2cdump 查看挂载在 i2c 0 总线上器件地址为 0x1e 的所有寄存器值，其中 0x1e 为 ap3216 传感器模块的 id。

```
[root@100ask:~]# i2cdump  -f -y 0 0x1e
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 00 00 00 02 02 01 ee ee 00 00 00 00 00 00 00 00    ...?????........
10: 00 01 00 03 ee ee ee ee 40 00 00 ff ff ee ee ee    .?.??????@....??
20: 05 13 01 00 00 ee ee ee 00 00 00 80 00 80 ee ee    ???..???...?.???
30: 00 00 ee 05 ee 00 ee 1b 00 00 00 00 00 00 00 00    ..???.??........
40: 82 84 81 8d 81 83 86 81 ee ee ee ee ee ee ee ee    ????????????????
50: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
60: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
70: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
80: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
90: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
a0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
b0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
c0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
d0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
e0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
f0: ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee    ????????????????
```

4 使用 i2cget -f -y 1 0x1e 0x0 （读取 i2c-1 上 0x20 器件的 0x0 存器值）。

```
[root@100ask:~]# i2cget -f -y 0 0x1e 0x00
0x00
[root@100ask:~]#
```

5 使用 i2cset -f -y 1 0x1e 0x0 3 (设置 i2c-0 上 0x1e 器件的 0x00 寄存器值为 0x3)

```
[root@100ask:~]# i2cset -f -y 0  0x1e 0x00 3
[root@100ask:~]# i2cget -f -y 0 0x1e 0x00
0x03
[root@100ask:~]#
```

2.6.2 Irda 设备调试 irda-utils

2.6.3 内存测试 mtester

您可以使用压力测试的 memtester 命令查找内存子系统故障。 memtester 命令是有效的用户空间测试工具，用于对内存子系统进行压力测试。 在 Linux 下查找间歇性和不确定性故障非常有效。

可以按以下方式运行 memtester：

```
[root@100ask:~]# memtester MEMORY ITERATIONS
```

其中 MEMORY：要分配和测试的内存量（以兆字节为单位）、ITERATIONS：要迭代的循环数。

如果一切正常，默认值为无限循环退出代码为 0。否则会使如下退出值

- x01 表示分配或锁定内存错误，或调用错误
- x02 表示卡住地址测试期间出错

- x04 其他测试之一中的错误

```
[root@100ask:~]# memtester  5 1
memtester version 4.3.0 (32-bit)
Copyright (C) 2001-2012 Charles Cazabon.
Licensed under the GNU General Public License version 2 (only).

pagesize is 4096
pagesizemask is 0xfffff000
want 5MB (5242880 bytes)
got  5MB (5242880 bytes), trying mlock ...locked.
Loop 1/1:
  Stuck Address       : ok
  Random Value        : ok
  Compare XOR         : ok
  Compare SUB         : ok
  Compare MUL         : ok
  Compare DIV         : ok
  Compare OR          : ok
  Compare AND         : ok
  Sequential Increment: ok
  Solid Bits          : ok
  Block Sequential    : ok
  Checkerboard        : ok
  Bit Spread          : ok
  Bit Flip            : ok
  Walking Ones        : ok
  Walking Zeroes      : ok
  8-bit Writes        : ok
  16-bit Writes       : ok

Done.
```

2.7 开发语言和脚本语言

2.7.1 python 环境测试

运行环境

- 支持的 INIT 服务: systemV systemD
- 支持的开发板系统: Buildroot Yocto qt 版本系统

开发板终端上输入 python 即可进入 python 编译器界面，我们在里面执行 print ("hello word")让其输出 hello word 来验证 python 环境是否正常。

```
[root@100ask:~]# python
```

```
[root@100ask:~]# python
Python 3.8.5 (default, Aug 26 2020, 09:52:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("hello word")
hello word
>>>
[root@100ask:~]#
```

如上图所示成功打印 hello word 字符，表示 python 基本环境正常，可以使用 ctrl + d 组合键来退出 python 环境。

2.7.2 Shell

2.8 网络通信类应用

2.8.1 开机自动设置 IP

运行环境

- 支持的 INIT 服务: systemD
- 支持的开发板系统: Buildroot Yocto

设置 eth0 自动获取 IP 地址,如下操作在开发板/etc/network/目录下对 interfaces 文件进行修改。

```
[root@100ask:~]# vi /etc/network/interfaces
```

修改并为如下内容，保存并退出，执行/etc/init.d/S40network restart 重启网络服务。

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
```

设置 eth0 为静态 IP 地址， 开发板/etc/network/目录下对 interfaces 文件进行修改 。

```
[root@100ask:~]# vi /etc/network/interfaces
```

修改并为如下内容，保存并退出，执行/etc/init.d/S40network restart 重启网络服务。

```
auto lo
  iface lo inet loopback
auto eth0
  iface eth0 inet static
  address 192.168.5.11
  netmask 255.255.255.0
  gateway 192.168.5.1
```

参考下图执行/etc/init.d/S40network restart 命令。

```
[root@100ask:~]# /etc/init.d/S40network reset
Usage: /etc/init.d/S40network {start|stop|restart}
[root@100ask:~]# /etc/init.d/S40network restart
Stopping network: ifdown: interface eth0 not configured
OK
Starting network: OK
[root@100ask:~]#
[root@100ask:~]#
[root@100ask:~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:01:3F:2D:3E:4D
          inet addr:192.168.5.11  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:860 errors:0 dropped:0 overruns:0 frame:0
          TX packets:121 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:58771 (57.3 KiB)  TX bytes:21990 (21.4 KiB)
```

2.8.2 PING

PING 主要用来测试网络的连通性，也可以测试网络延迟以及丢包率。

1 接线与信息输出

通过 CAT6 网线将设备连接到交换机或路由器，控制台会显示内核输出的连接信息，如下：

```
[root@100ask:~]# [   11.495330] fec 20b4000.ethernet eth0: Link is Up - 100Mbps/Full
- flow control off
[   11.503167] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

2 测试外网网址

```
[root@100ask:~]# ping www.baidu.com -I eth0
PING www.baidu.com (112.80.248.75) 56(84) bytes of data.
64 bytes from 112.80.248.75 (112.80.248.75): icmp_seq=1 ttl=56 time=28.3 ms
64 bytes from 112.80.248.75 (112.80.248.75): icmp_seq=2 ttl=56 time=26.4 ms
64 bytes from 112.80.248.75 (112.80.248.75): icmp_seq=3 ttl=56 time=34.5 ms
64 bytes from 112.80.248.75 (112.80.248.75): icmp_seq=4 ttl=56 time=106 ms
```

注意： ping 公网需要确保 DNS 正常工作。

上面结果显示 www.baidu.com 经过域名解析之后的 IP 地址为

112.80.248.75，icmp_seq 代表 icmp 包的编号，如果编号连续说明没有丢包；time 代表响应的延迟时间，当然这个时间越短越好。除了对以太网进行测试，ping 命令也可以用于测试 Wi-Fi。

2.8.3 ssh 登陆工具使用

运行环境

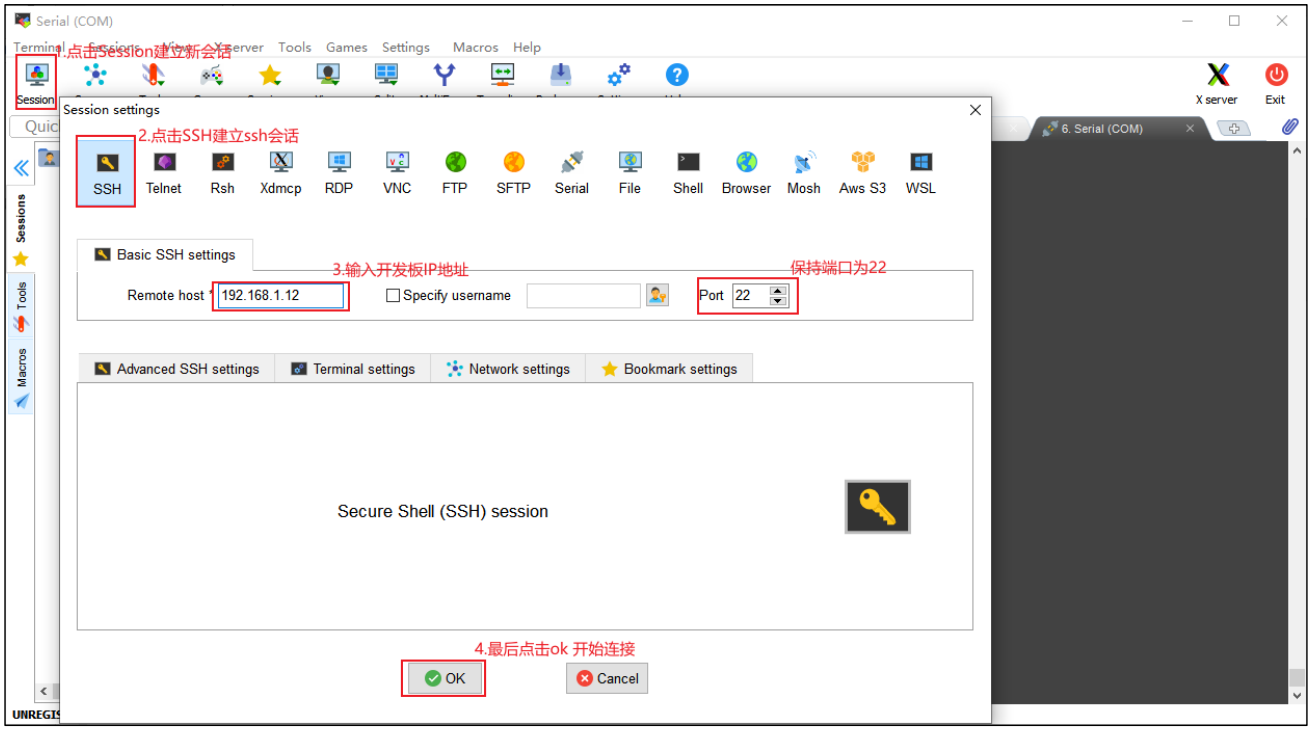
- 支持的 INIT 服务：systemV systemD
- 支持的开发板系统：Buildroot Yocto busybox

开发板系统默认自带 ssh 服务，可以通过网络方式远程登录开发板，用以传输文件或者开发。

首先需要将开发板联网并可以 ping 通 widnows 或者 ubuntu 主机，保证局域网内网络连通。

如下图所示，我的开发板 IP 地址为 192.168.1.12 ，则使用 mobaxterm 来建立一个 ssh 会话登录到开发板。

```
[root@100ask:~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.12  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fdfd:3597:aa7a:0:201:1fff:fe2d:3e4d  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::201:1fff:fe2d:3e4d  prefixlen 64  scopeid 0x20<link>
        ether 00:01:1f:2d:3e:4d  txqueuelen 1000  (Ethernet)
        RX packets 28  bytes 2360 (2.3 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2176 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 51

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2  bytes 196 (196.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 196 (196.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```



如果点击 ok 后输入 root 用户名提示需要输入密码表示 ssh 服务不支持

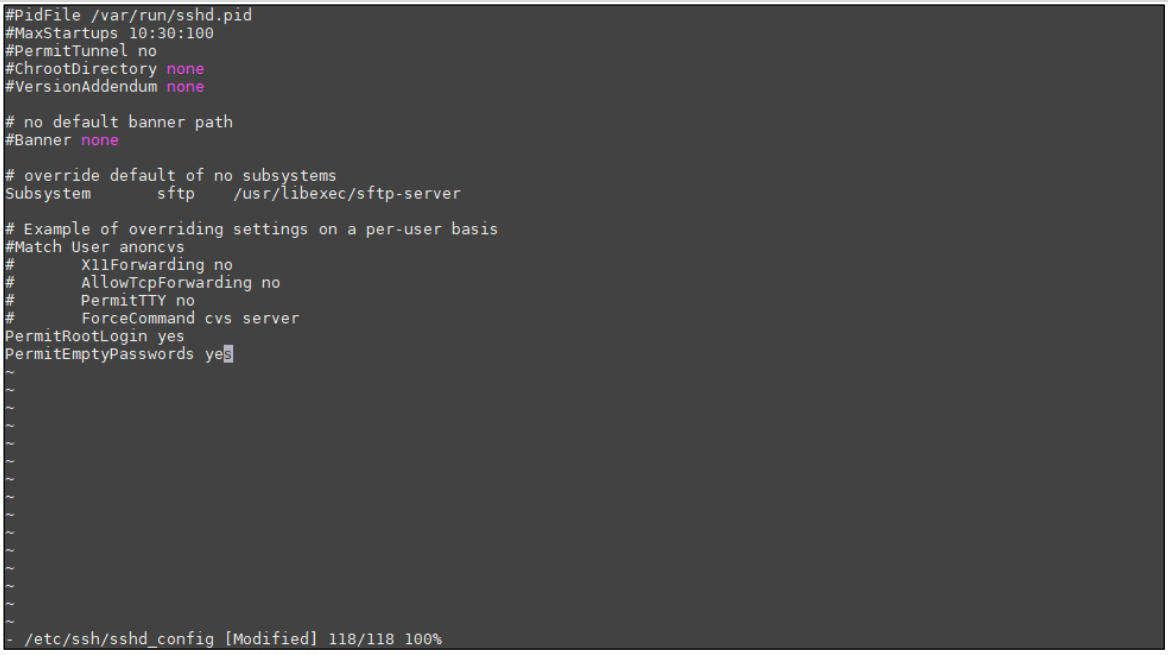
root 用户空密码登录，则需要参考参考如下配置来修改 ssh 配置文件增加对 root 用户空密码的支持。

在开发板终端上编辑 ssh 配置文件

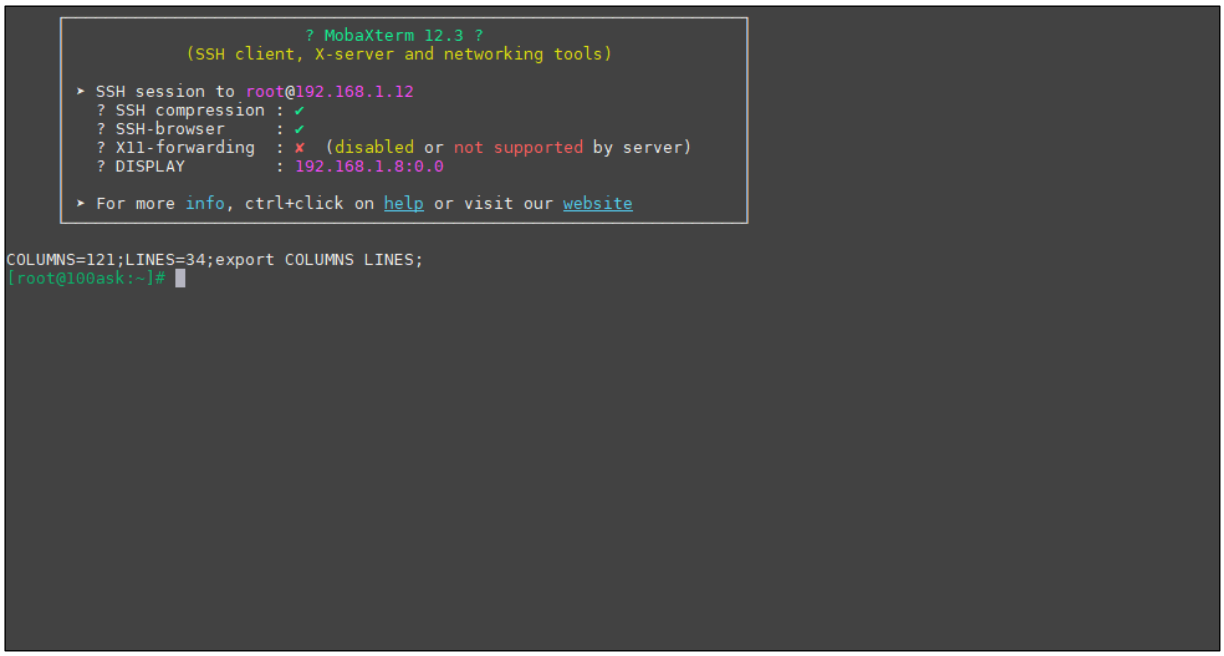
```
[root@100ask:~]# vi  /etc/ssh/sshd_config
```

在/etc/ssh/sshd_config 最后中加入或者修改，如下图示例，添加完成后，输入 :wq 保存并退出，之后执行 reboot 命令重启开发板系统。

```
PermitRootLogin yes             #允许 root 登录
PermitEmptyPasswords yes        #允许空密码登录
```



重启系统后，首先确保开发板已经获取到 IP 地址，之后重新打开 ssh 会话，输入 root 用户名按下回车确定即可登录开发板系统。



2.8.4 hci 工具使用说明

运行环境

- 支持的 INIT 服务：systemV systemD
- 支持的开发板系统：Buildroot Yocto busybox

hci 工具集是用来设置并查看蓝牙芯片特性的一套工具集，如下所示演示如何启用蓝牙设备，并扫描周边普通蓝牙设备和低功耗蓝牙设备，最后通过 sdptool 命令来查看扫描到设备的详细参数。

如下所示，使用 hciconfig -a 命令查看板载目前都有蓝牙设备，以及获取所有蓝牙设备的详细信息。

```
[root@100ask:~]# hciconfig -a
hci0:   Type: Primary  Bus: UART
        BD Address: B0:02:47:36:98:E9  ACL MTU: 1021:7  SCO MTU: 64:1
        DOWN
        RX bytes:2162 acl:0 sco:0 events:205 errors:0
        TX bytes:39610 acl:0 sco:0 commands:205 errors:0
        Features: 0xbf 0xfe 0xcf 0xfe 0xdb 0xff 0x7b 0x87
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH SNIFF
        Link mode: SLAVE ACCEPT

[root@100ask:~]#
```

由上图我们可以知道有

```
[root@100ask:~]# hciconfig hci0 up
[root@100ask:~]# hciconfig hci0
[root@100ask:~]# hciconfig hci0 up
[root@100ask:~]# hciconfig hci0
hci0:   Type: Primary  Bus: UART
        BD Address: B0:02:47:36:98:E9  ACL MTU: 1021:7  SCO MTU: 64:1
        UP RUNNING
        RX bytes:2927 acl:0 sco:0 events:251 errors:0
        TX bytes:40401 acl:0 sco:0 commands:251 errors:0

[root@100ask:~]#
```

```
[root@100ask:~]# hcitool scan
[root@100ask:~]# hcitool scan
Scanning ...
        00:1A:7D:DA:71:13       DESKTOP-BNFRK7N
        94:65:2D:9D:86:F1       OnePlus 5
[root@100ask:~]#
```

```
[root@100ask:~]# hcitool lescan
[root@100ask:~]# hcitool lescan
LE Scan ...
C9:02:C9:23:53:6F (unknown)
C9:02:C9:23:53:6F Mi Smart Band 4
C0:15:83:33:18:1E (unknown)
2D:EA:30:96:79:7F (unknown)
5A:A4:5F:07:A0:52 (unknown)
5A:A4:5F:07:A0:52 (unknown)
2D:DD:29:10:D3:67 (unknown)
36:DE:3F:BF:4C:CF (unknown)
02:A8:22:D6:C6:C0 (unknown)
C0:15:83:33:18:1E BTP-P33-181E
```

```
[root@100ask:~]# hcitool leinfo C9:02:C9:23:53:6F
[root@100ask:~]# hcitool leinfo C9:02:C9:23:53:6F
Requesting information ...
        Handle: 64 (0x0040)
        LMP Version: 5.0 (0x9) LMP Subversion: 0x10f
        Manufacturer: Dialog Semiconductor B.V. (210)
        Features: 0xff 0x00 0x00 0x00 0x00 0x00 0x00 0x00
[root@100ask:~]#
```

```
[root@100ask:~]# sdptool  browse
Inquiring ...
Browsing 00:1A:7D:DA:71:13 ...
Service Name: Service Discovery
Service Description: Publishes services to remote devices
Service Provider: Microsoft
Service RecHandle: 0x0
Service Class ID List:
  "SDP Server" (0x1000)
Protocol Descriptor List:
  "L2CAP" (0x0100)
    PSM: 1
  "SDP" (0x0001)
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100

Browsing 00:1A:7D:DA:71:13 ...
Service Search failed: Invalid argument
Service Name: Device ID Service Record
Service Description: Device ID Service Record
Service RecHandle: 0x10000
Service Class ID List:
  "PnP Information" (0x1200)
Protocol Descriptor List:
  "L2CAP" (0x0100)
    PSM: 1
  "SDP" (0x0001)
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
```

2.8.5 iw 工具使用说明

运行环境

- 支持的 INIT 服务：systemV systemD
- 支持的开发板系统：Buildroot Yocto busybox

使用 iw 命令获取周围所有 wifi 网络的详细信息。

```
[root@100ask:~]# iw wlan0 scan
[  542.309595] [dhd-wlan0] wl_run_escan : LEGACY_SCAN sync ID: 0, bssidx: 0
BSS cc:08:fb:62:66:cd(on wlan0)
        TSF: 543411018 usec (0d, 00:09:03)
        freq: 2412
        beacon interval: 100 TUs
        capability: ESS Privacy ShortPreamble (0x0031)
        signal: -49.00 dBm
        last seen: 0 ms ago
        SSID: hyd
        Supported rates: 1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0
        DS Parameter set: channel 1
        ERP: Use_Protection
        Extended supported rates: 24.0 36.0 48.0 54.0
        HT capabilities:
                Capabilities: 0x11ef
                        RX LDPC
                        HT20/HT40
                        SM Power Save disabled
                        RX HT20 SGI
                        RX HT40 SGI
                        TX STBC
                        RX STBC 1-stream
                        Max AMSDU length: 3839 bytes
                        DSSS/CCK HT40
                Maximum RX AMPDU length 65535 bytes (exponent: 0x003)
                Minimum RX AMPDU time spacing: 8 usec (0x06)
                HT TX/RX MCS rate indexes supported: 0-15
        HT operation:
```

2.9 系统工具

进程也是操作系统中的一个重要概念，它是一个程序的一次执行过程，程序是进程的一种静态描述，系统中运行的每一个程序都是在它的进程中运行的。

Linux 系统中所有的进程是相互联系的，除了初始化进程外，所有进程都有一个父进程。新的进程不是被创建，而是被复制，或是从以前的进程复制而来。Linux 中所有的进程都是由一个进程号为 1 的 init 进程衍生而来的。Linux 系统包括 3 种不同类型的进程，每种进程都有自己的特点和属性：

① 交互进程：由一个 Shell 启动的进程，既可以在前台运行，又可以在后台运行。

② 批处理进程：这种进程和终端没有联系，是一个进程序列。这种进程被提交到等待队列顺序执行的进程。

③ 监控进程(守护进程)：守护进程总是活跃的，一般是在后台运行，守护进程一般是由系统在开始时通过脚本自动激活启动或 root 启动

④ 对于 linux 系统来说，进程的管理是重要的一环，对于进程的管理通常是通过进程管理工具实现的，Linux 系统中比较常用的进程管理命令有以下几种：ps top vmstat kill 。

2.9.1 显示当前进程工具

显示当前系统进程的运行情况，一般语法如下：

```
[root@100ask:~]# ps --help
BusyBox v1.31.1 (2020-10-13 11:02:58 UTC) multi-call binary.

Usage: ps [-o COL1,COL2=HEADER]

Show list of processes

       -o COL1,COL2=HEADER     Select columns for display
```

部分参数组合说明：

- -u：以用户为中心组织进程状态信息显示；
- -a：与终端无关的进程；
- -x：与终端有关的进程； （线程，就是轻量级进程；）

通常上面命令组合使用：aux。

- --e：显示所有进程；相当于 ax；
- -f：显示完整格式程序信息；

通常以上命令组合使用：ef

- -H：以进程层级显示进程数的
- -F：显示更多的程序信息

通常组合使用命令：eHF

显示所有进程的信息情况示例如下：

```
[root@100ask:~]#  ps aux
PID   USER     COMMAND
    1 root     init [3]
    2 root     [kthreadd]
    3 root     [ksoftirqd/0]
    5 root     [kworker/0:0H]
    6 root     [kworker/u2:0]
```

```
    7 root     [rcu_preempt]
    8 root     [rcu_sched]
    9 root     [rcu_bh]
   10 root     [migration/0]
   11 root     [lru-add-drain]
   12 root     [cpuhp/0]
   13 root     [kdevtmpfs]
   14 root     [oom_reaper]
   15 root     [writeback]
   16 root     [kcompactd0]
   17 root     [crypto]
   18 root     [bioset]
   19 root     [kblockd]
   21 root     [ata_sff]
   22 root     [cfg80211]
   23 root     [watchdogd]
   24 root     [rpciod]
   25 root     [xprtiod]
   26 root     [kswapd0]
   27 root     [vmstat]
   28 root     [nfsiod]
   74 root     [bioset]
```

其中 PID 为当前进程的 ID 号 USER 表示表示程序所属用户 COMMAND 表示运行的应用程序

2.9.2 top 显示 linux 进程

top 命令将相当多的系统整体性能信息放在一个屏幕上。显示内容还能以交互的方式进行改变。动态的持续监控进程的运行状态，top 语法一般如下：

```
[root@100ask:~]# top  --help
BusyBox v1.31.1 (2020-10-13 11:02:58 UTC) multi-call binary.

Usage: top [-b] [-n COUNT] [-d SECONDS]

Provide a view of process activity in real time.
Read the status of all processes from /proc each SECONDS
and display a screenful of them.
Keys:
       N/M/P/T: sort by pid/mem/cpu/time
       R: reverse sort
       Q,^C: exit
Options:
       -b      Batch mode
       -n N    Exit after N iterations
       -d SEC  Delay between updates
```

动态查看系统进程示例如下：

```
Mem: 165428K used, 335520K free, 820K shrd, 17492K buff, 37752K cached
CPU:   0% usr   1% sys   0% nic  98% idle   0% io   0% irq   0% sirq
Load average: 0.06 0.06 0.02 1/117 814
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
  814   317 root     R     2708   1%   1% top
  113     2 root     SW       0   0%   0% [mmcqd/0]
  266     1 root     S    51280  10%   0% /usr/sbin/NetworkManager
  278     1 mosquitt S     5412   1%   0% /usr/sbin/mosquitto -c /etc/mosquitto/mosqui
```

```
tto.conf
    1     0 root     S     1748   0%   0% init [3]
  803     2 root     SW       0   0%   0% [kworker/0:2]
  284     1 root     S<     102m  21%   0% /usr/bin/pulseaudio --system --daemonize --d
isallow-module-loading --disallow-exit --exit-idle-
  315     1 root     S    43916   9%   0% /usr/bin/swupdate -f /etc/swupdate/swupdate.
cfg -L -e rootfs,rootfs-2 -u
  258     1 root     S    40784   8%   0% /usr/sbin/ModemManager
  295   284 root     S    19192   4%   0% /usr/libexec/pulse/gsettings-helper
  319   315 root     S    17512   3%   0% /usr/bin/swupdate -f /etc/swupdate/swupdate.
cfg -L -e rootfs,rootfs-2 -u
  197     1 root     S    11480   2%   0% /sbin/udevd -d
  273     1 root     S     7900   2%   0% /usr/sbin/ntpd -g
  292     1 root     S     7388   1%   0% /usr/sbin/wpa_supplicant -u
  303     1 root     S     5880   1%   0% /usr/sbin/sshd
  342   317 root     S     4164   1%   0% psplash -n
  317     1 root     S     3388   1%   0% -bash
```

对其上面一些任务栏进行简单解释说明：

- VSZ：vittual memory size 虚拟内存集
- RSS：resident size，常驻内存集
- STAT：进程状态有以下几个
    - R：runing
    - S：interuptable sleeping 可中断睡眠
    - D：uninteruptable sleeping 不可中断睡眠
    - T：stopped
    - Z：zombie 僵尸进程
    - +:前台进程
    - N:低优先级进程
    - l:多线程进程

2.9.3 vmstat 虚拟内存的统计工具

这个命令可查看内存空间的使用状态，能获取整个系统的性能粗略信息，vmstat 运行于两种模式：采样模式和平均模式。如果不指定参数，则 vmstat 统计运行于平均模式下，vmstat 显示从系统启动以来所有统计数据的均值。常用语法以及参数如下：

```
[root@100ask:~]# vmstat -h
Usage:
 vmstat [options] [delay [count]]
Options:
 -a, --active           active/inactive memory
 -f, --forks            number of forks since boot
 -m, --slabs            slabinfo
 -n, --one-header       do not redisplay header
 -s, --stats            event counter statistics
 -d, --disk             disk statistics
 -D, --disk-sum         summarize disk statistics
 -p, --partition <dev>  partition specific statistics
 -S, --unit <char>      define display unit
```

```
 -w, --wide            wide output
 -t, --timestamp       show timestamp

 -h, --help     display this help and exit
 -V, --version  output version information and exit
```

vmstat 运行于平均模式，显示从系统启动以来所有统计数据的均值：

```
[root@100ask:~]# vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b  swpd  free buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0  0 200640  13556 131852  0  0  39    11  148  227  6  3 90  0  0
```

对其上面一些任务栏进行简单解释说明

- r：当前可运行的进程数。这些进程没有等待 I/O,而是已经准备号运行。理想状态，可运行进程数与可用 cpu 数量相等
- b：等待 I/O 完成的被阻塞进程数
- forks：创建新进程的次数
- in：系统发生中断的次数
- cs：系统发生上下文切换的次数
- us：用户进程消耗的总 CPU 时间百分比
- sy：系统代码消耗的总 CPU 时间的百分比，其中包括消耗在 system、irq 和 softirq 状态的时间
- wa：等待 I/O 消耗的总 CPU 的百分比
- id：系统空闲消耗的总 CPU 时间的百分比

统计系统各种数据详细信息如下：

```
[root@100ask:~]# vmstat -s
      435992 K total memory
       90008 K used memory
       82380 K active memory
       88924 K inactive memory
      200528 K free memory
       13556 K buffer memory
      131900 K swap cache
           0 K total swap
           0 K used swap
           0 K free swap
       20273 non-nice user cpu ticks
        4412 nice user cpu ticks
       19691 system cpu ticks
      630164 idle cpu ticks
         241 IO-wait cpu ticks
           0 IRQ cpu ticks
          22 softirq cpu ticks
           0 stolen cpu ticks
      127150 pages paged in
       36195 pages paged out
           0 pages swapped in
           0 pages swapped out
      969112 interrupts
     1492562 CPU context switches
  1581090644 boot time
       10649 forks
```

对其上面一些任务栏进行简单解释说明：

- total memory：系统总内存
- used memory：已使用内存
- CPU ticks：显示的是自系统启动的 CPU 时间，这里的“tick”是一个时间单位
- forks：大体上表示的是从系统启动开始已经创建的新进程的数量
- vmstat 提供了关于 linux 系统性能的众多信息。在调查系统问题时，它是核心工具之一。

2.9.4 kill 进程终止工具

发送指定的信号到相应进程。不指定型号将发送 SIGTERM（15）终止指定进程。如果任无法终止该程序可用“-KILL” 参数，其发送的信号为 SIGKILL(9) ，将强制结束进程，使用 ps 命令或者 jobs 命令可以查看进程号。root 用户将影响用户的进程，非 root 用户只能影响自己的进程。kill 命令一般语法如下：

```
kill [ -s signal | -p ] [ -a ] pid ...
kill -l [ signal ]
```

部分参数组合说明：

- -s：指定发送的信号
- -p：模拟发送信号
- -l：指定信号的名称列表
- pid：要中止进程的 ID 号
- Signal：表示信号

① 首先使用 ps -ef 与管道命令确定要杀死进程的 PID。

```
[root@100ask:~]# ps -ef | grep mxapp2
  835 root     /usr/bin/mxapp2 --plugin tslib:/dev/input/event1
 3148 root     grep mxapp2
```

② 然后输入以下命令终止进程：

```
[root@100ask:~]# kill -9 835
```

③ killall 命令终止同一进程组内的所有进程，允许指定要终止的进程的名称而非 PID 进程号。

```
[root@100ask:~]# killall mxapp2
```

2.9.5 du 命令统计目录大小

运行环境

- 支持的 INIT 服务:systemV systemD
- 支持的开发板系统:Buildroot Yocto busybox

du 命令支持参数简介

```
[root@100ask:~]# du [-aHLdclsxhmk] 文件或目录名称
```

选项与参数：

- -a：同时显示每个文件的文件大小

- -d：N 将输出限制为深度<N 的目录
- -c：显示总计
- -s：每个参数仅显示总计
- -x：跳过不同文件系统上的目录
- -h：可读格式的大小（例如 1K 243M 2G）
- -m：大小（以兆字节为单位）
- -k：大小（以千字节为单位）（默认）

1 统计/etc 目录下所有文件所占用的容量。

先执行“cd /etc”进入/etc 目录，再执行 du 命令：

```
[root@imx6ull:/etc]# du
12      ./profile.d
12      ./openldap
8       ./udev/rules.d
16      ./udev
12      ./bash_completion.d
16      ./sysconfig
8       ./ofono
48      ./lighttpd/conf.d
72      ./lighttpd
4       ./network/if-down.d
8       ./network/if-pre-up.d
4       ./network/if-up.d
4       ./network/if-post-down.d
32      ./network
8       ./default
28      ./dbus-1/system.d
40      ./dbus-1
4       ./ssl/private
12      ./ssl/misc
4       ./ssl/certs
56      ./ssl
40      ./iproute2
80      ./init.d
16      ./samba
12      ./libnl
24      ./pulse
8       ./fonts/conf.d
16      ./fonts
8       ./ppp/peers
60      ./ppp/radius
76      ./ppp
4       ./sudoers.d
604     ./ssh
1388    .
[root@imx6ull:/etc]#
```

2 统计每个文件和目录所占用的容量大小，并以易读的方式展示出来

```
[root@imx6ull:/etc]# du -ah
4.0K    ./profile.d/umask.sh
4.0K    ./profile.d/bash_completion.sh
12.0K   ./profile.d
4.0K    ./shadow
4.0K    ./openldap/ldap.conf.default
4.0K    ./openldap/ldap.conf
12.0K   ./openldap
0       ./resolv.conf
4.0K    ./random-seed
4.0K    ./fstab
4.0K    ./memstat.conf
4.0K    ./udev/udev.conf
4.0K    ./udev/rules.d/70-persistent-net.rules
8.0K    ./udev/rules.d
16.0K   ./udev
4.0K    ./ntp.conf
4.0K    ./inputrc
4.0K    ./slsh.rc
4.0K    ./sudoers
4.0K    ./hostname
8.0K    ./bash_completion.d/perf
12.0K   ./bash_completion.d
12.0K   ./services
4.0K    ./timezone
4.0K    ./sysconfig/sysstat
8.0K    ./sysconfig/sysstat.ioconf
16.0K   ./sysconfig
```

3 统计根目录下每个目录所占用的容量

```
[root@imx6ull:~]# du -sm /*
2       /bin
8       /boot
0       /dev
2       /etc
17      /home
15      /lib
0       /lib32
1       /libexec
0       /linuxrc
1       /lost+found
1       /media
1       /mnt
1       /opt
0       /proc
1       /root
1       /run
1       /samba
3       /sbin
0       /sys
1       /tmp
289     /usr
2       /var
[root@imx6ull:~]#
```

4 示例四:统计 /etc 目录下层级 1 的所有目录所占用的大小。

```
[root@imx6ull:/etc]# du -d 1
12      ./profile.d
12      ./openldap
16      ./udev
12      ./bash_completion.d
16      ./sysconfig
8       ./ofono
72      ./lighttpd
32      ./network
8       ./default
40      ./dbus-1
56      ./ssl
40      ./iproute2
80      ./init.d
16      ./samba
12      ./libnl
24      ./pulse
16      ./fonts
76      ./ppp
4       ./sudoers.d
604     ./ssh
1388    .
[root@imx6ull:/etc]#
```

2.9.6 df：查看系统已用空间

运行环境

- 支持的 INIT 服务:systemV systemD
- 支持的开发板系统: Buildroot Yocto busybox

df 支持的命令参数简介

```
[root@100ask:~]# df [-PkmhT] [目录或档名]
```

选项与参数:

- -k：以 KBytes 的容量显示各档案系统;
- -m：以 MBytes 的容量显示各档案系统;
- -h：以人们较易阅读的 GBytes, MBytes, KBytes 等格式自行显示;
- -T：连同该 partition 的 filesystem 名称 (例如 ext3) 也列出;

```
[root@100ask:~]# df --help
BusyBox v1.31.1 (2020-07-30 11:20:21 UTC) multi-call binary.

Usage: df [-PkmhT] [FILESYSTEM]...

Print filesystem usage statistics

        -P      POSIX output format
        -k      1024-byte blocks (default)
        -m      1M-byte blocks
        -h      Human readable (e.g. 1K 243M 2G)
        -T      Print filesystem type
[root@100ask:~]#
```

1 将系统内所有的 Filesystem 列出来！

```
[root@100ask:~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/root               999320    651352    279156  70% /
devtmpfs                152480         0    152480   0% /dev
tmpfs                   218528         0    218528   0% /dev/shm
tmpfs                   218528       744    217784   0% /run
tmpfs                   218528         0    218528   0% /sys/fs/cgroup
tmpfs                   218528         0    218528   0% /tmp
[root@100ask:~]#
```

2 将文件系统容量显示格式以易读的方式展示。

```
[root@imx6ull:~]# df -h
Filesystem               Size      Used Available Use% Mounted on
/dev/root                3.5G    341.1M      3.0G  10% /
devtmpfs                85.3M         0     85.3M   0% /dev
tmpfs                  245.8M         0    245.8M   0% /dev/shm
tmpfs                  245.8M    644.0K    245.2M   0% /tmp
tmpfs                  245.8M    164.0K    245.6M   0% /run
[root@imx6ull:~]#
```

3 将系统内的文件系统类型和容量大小以易读的方式展示出来。

```
[root@imx6ull:~]# df -hT
Filesystem           Type            Size      Used Available Use% Mounted on
/dev/root            ext4            1.9G    436.3M      1.4G  23% /
devtmpfs             devtmpfs       85.3M         0     85.3M   0% /dev
tmpfs                tmpfs         245.8M         0    245.8M   0% /dev/shm
tmpfs                tmpfs         245.8M       1.6M    244.2M   1% /tmp
tmpfs                tmpfs         245.8M    188.0K    245.6M   0% /run
/dev/mmcblk0p2       ext4          623.8M    590.4M         0 100% /mnt
[root@imx6ull:~]#
```

4 输出结果提示信息含义简介。

- Filesystem：代表该系统是在哪个设备的哪个分区，有些是虚拟文件系统比如 tmpfs。
- Type：　文件系统类型。
- 1k-blocks：说明底下的数字单位是 1KB ，可利用 -h 或 -m 来改变单位；
- Used：顾名思义，就是使用掉的磁盘空间。
- Available：也就是剩下的磁盘空间大小。
- Use%：就是磁盘的使用率，如果使用率高达 90% 以上时， 最好需要注意一下了，免得容量不足造成系统问题。
- Mounted on：就是磁盘挂载所在目录。

df 读取的资料整个文件系统的统计信息，在显示的结果中你需要特别留意的是那个根目录(/dev/root)的剩余容量。所有的资料都是由根目录衍生出来的，当根目录的剩余容量剩下 0 时，那你的 Linux 存储空间肯定不够了。

2.9.7 kmod 内核模块管理工具使用

运行环境

- 支持的 INIT 服务： systemV systemD

- 支持的开发板系统：Buildroot Yocto busybox

1 lsmod：列出已经安装了哪些模块

```
[root@100ask:~]# lsmod
```

```
[root@imx6ull:~]# lsmod
Module                  Size  Used by    Not tainted
inv_mpu6050_spi         2052  0
inv_mpu6050            10948  2 inv_mpu6050_spi
evbug                   2078  0
8723bu               1823356  0
[root@imx6ull:~]#
```

提示信息含义说明：

- Module ：表示模块的名称。
- Size ： 表示模块的大小
- Used： 使用者。

2 insmod：手工安装模块

后面讲到的 modprobe 命令，它是从/lib/modules 下的目录里自动安装某个模块。但是在实验过程中，我们经常需要手工安装其他目录下的模块，可以使用以下命令安装(需要指定模块文件即 ko 文件的位置)。

```
[root@100ask:~]# insmod /path/to/module/xxx.ko
[root@100ask:~]# insmod  -f  /path/to/module/xxx.ko    // 强制安装
```

开发板出厂时运行的是我们编译好的内核，当你做实验时需要先编译出自己的内核，然后编译出自己的驱动程序。如果你不想替换内核，那么你的驱动程序跟板上的内核并不完全匹配。这时就要用 insmod -f 命令强制安装驱动程序。下面是一个例子：

```
[root@imx6ull:~]# cd /
[root@imx6ull:/]# insmod gpio_key_drv.ko   1. 由于内核不匹配，安装驱动失败
[ 2306.475737] gpio_key_drv: loading out-of-tree module taints kernel.
[ 2306.483446] gpio_key_drv: disagrees about version of symbol device_create
[ 2306.492258] gpio_key_drv: Unknown symbol device_create (err -22)          错误信息
[ 2306.498561] gpio_key_drv: disagrees about version of symbol device_destroy
[ 2306.506343] gpio_key_drv: Unknown symbol device_destroy (err -22)
insmod: ERROR: could not insert module gpio_key_drv.ko: Invalid parameters
[root@imx6ull:/]# insmod -f gpio_key_drv.ko   2. 可以强制安装
[ 2311.452458] gpio_key_drv: module_layout: kernel tainted.
[ 2311.457854] Disabling lock debugging due to kernel taint
[root@imx6ull:/]#
```

3 rmmod：卸载掉某个已安装的模块

```
[root@100ask:~]# rmmod <模块名称>
```

从上图 lsmod 可知系统已经安装了哪些模块，这里我们以卸载 usb wifi 模块驱动为例，具体操作如下图所示。

```
[root@imx6ull:~]# rmmod 8723bu
[ 7647.713332] usbcore: deregistering interface driver rtl8723bu
[root@imx6ull:~]#
```

4 modinfo：查看某个模块的详细信息

```
[root@100ask:~]# modinfo <模块>
```

```
[root@imx6ull:~]# modinfo /lib/modules/4.9.88/kernel/drivers/input/evbug.ko
filename:       /lib/modules/4.9.88/kernel/drivers/input/evbug.ko
author:         Vojtech Pavlik <vojtech@ucw.cz>
description:    Input driver event debug module
license:        GPL
alias:          input:b*v*p*e*-e*k*r*a*m*l*s*f*w*
srcversion:     65B0A26AD78122C417312C2
depends:
intree:         Y
vermagic:       4.9.88 SMP preempt mod_unload modversions ARMv7 p2v8
[root@imx6ull:~]#
```

5 **modinfo 命令用于显示 kernel 模块的信息。**

用法：

```
[root@100ask:~]# modinfo [-adlpn0Fkbvh]  <模块文件>
```

- -a 或--author　显示模块开发人员。
- -d 或--description　显示模块的说明。
- -l 或—license　显示版本信息
- -p 或--parameters　显示模块所支持的参数。
- -0 或--null　用 \0 代替 \n
- -F 或--field=FIELD　仅打印提供的字段
- -k 或--set-version=VERSION　用 VERSION 代替 `uname -r`
- -b 或--basedir=DIR　使用 DIR 作为/lib/modules 的文件系统根目录
- -V 或--version　显示版本信息
- -h 或--help　显示帮助信息

可以在开发板执行 modinfo -h 命令查看帮助信息，如下图：

```
[root@imx6ull:~]# modinfo -h
Usage:
        modinfo [options] filename [args]
Options:
        -a, --author                Print only 'author'
        -d, --description           Print only 'description'
        -l, --license               Print only 'license'
        -p, --parameters            Print only 'parm'
        -n, --filename              Print only 'filename'
        -0, --null                  Use \0 instead of \n
        -F, --field=FIELD           Print only provided FIELD
        -k, --set-version=VERSION   Use VERSION instead of `uname -r`
        -b, --basedir=DIR           Use DIR as filesystem root for /lib/modules
        -V, --version               Show version
        -h, --help                  Show this help
```

下面是一个例子，用来显示 evbug 模块的信息：

```
[root@100ask:~]# modinfo  evbug
```

```
[root@imx6ull:/]# modinfo evbug
filename:       /lib/modules/4.9.88/kernel/drivers/input/evbug.ko
license:        GPL
description:    Input driver event debug module
author:         Vojtech Pavlik <vojtech@ucw.cz>
alias:          input:b*v*p*e*-e*k*r*a*m*l*s*f*w*
depends:
intree:         Y
vermagic:       4.9.88 SMP preempt mod_unload modversions ARMv7 p2v8
[root@imx6ull:/]#
```

6 **modprobe：自动安装模块**

modprobe 可载入指定的个别模块，或是载入一组相依的模块。modprobe 会

根据 depmod 所产生的相依关系，决定要载入哪些模块。若在载入过程中发生错误，则 modprobe 会卸载整组的模块。

insmod 与 modprobe 都是用于安装内核模块，差别是：modprobe 能够处理模块的依赖问题。比方你要加载 a 模块，但是 a 要求系统先载入 b 模块时，直接用 insmod 加载可能会出现错误讯息。modprobe 会自动加载 b，才加载 a，帮你处理这些依赖关系。

用法：

```
[root@100ask:~]# modprobe [options] [模块名]
```

开发板执行 modprobe -h 可以看到命令用法，如下图：

```
[root@imx6ull:~]# modprobe -h
Usage:
        modprobe [options] [-i] [-b] modulename
        modprobe [options] -a [-i] [-b] modulename [modulename...]
        modprobe [options] -r [-i] modulename
        modprobe [options] -r -a [-i] modulename [modulename...]
        modprobe [options] -c
        modprobe [options] --dump-modversions filename
Management Options:
        -a, --all                   Consider every non-argument to
                                    be a module name to be inserted
                                    or removed (-r)
        -r, --remove                Remove modules instead of inserting
            --remove-dependencies   Also remove modules depending on it
        -R, --resolve-alias         Only lookup and print alias and exit
            --first-time            Fail if module already inserted or removed
        -i, --ignore-install        Ignore install commands
        -i, --ignore-remove         Ignore remove commands
        -b, --use-blacklist         Apply blacklist to resolved alias.
        -f, --force                 Force module insertion or removal.
                                    implies --force-modversions and
                                    --force-vermagic
            --force-modversion      Ignore module's version
            --force-vermagic        Ignore module's version magic

Query Options:
        -D, --show-depends          Only print module dependencies and exit
        -c, --showconfig            Print out known configuration and exit
        -c, --show-config           Same as --showconfig
            --show-modversions      Dump module symbol version and exit
            --dump-modversions      Same as --show-modversions

General Options:
        -n, --dry-run               Do not execute operations, just print out
        -n, --show                  Same as --dry-run
        -C, --config=FILE           Use FILE instead of default search paths
        -d, --dirname=DIR           Use DIR as filesystem root for /lib/modules
        -S, --set-version=VERSION   Use VERSION instead of `uname -r`
        -s, --syslog                print to syslog, not stderr
        -q, --quiet                 disable messages
        -v, --verbose               enables more messages
        -V, --version               show version
        -h, --help                  show this help
```

常用的命令解释如下：

- -r 卸载模块
- -f 名制安装或卸载
- -r 删除模块（堆栈）或自动清洁
- -D 显示依赖

操作示例，modprobe 自动解析依赖并安装相应模块：

```
[root@imx6ull:~]# modprobe g_serial.ko
[root@imx6ull:~]# lsmod
Module                  Size  Used by    Not tainted
g_serial                3865  0
inv_mpu6050_spi         2052  0
inv_mpu6050            10948  2 inv_mpu6050_spi
evbug                   2078  0
[root@imx6ull:~]#
```

注意：使用 modproe 也会碰到 insmod 同样的内核版本不一致问题，可以使用 modprobe -f 强制安装。

2.10 库

目前我们提供开发板系统镜像配套的工具链含有如下库，库的路径在

```
/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-nueabihf_sdk-buildroot/arm-buildroot-linux-gnueabihf/sysroot/usr/include
```

2.10.1 opencv 库使用

使用 python 语言环境的 opencv 库来做相关开发示例，如下图所示，使用 python 语言去调用 opecv 模块库来显示图片。

```
import cv2
imgviewx=cv2.imread("/usr/share/myir/Capture/image2.png")
cv2.namedWindow("Linux")
cv2.imshow("Linux",imgviewx)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

新建文件保存后，添加可执行权限，并用 python 来运行此 python 文件。

```
[root@100ask:~]# chmod +x opencv.py
[root@100ask:~]# python opencv.py
```

```
[root@100ask:~]# python opencv.py
libpng warning: iCCP: known incorrect sRGB profile
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
```

注意事项与售后维修

1 注意事项

- 使用产品之前，请仔细阅读本手册，并妥善保管，以备将来参考；
- 请注意和遵循标注在产品上的所有警示和指引信息；
- 请使用配套电源适配器，以保证电压、电流的稳定；
- 请在凉爽、干燥、清洁的地方使用本产品；
- 请勿在冷热交替环境中使用本产品，避免结露损坏元器件；
- 请勿在湿气过重、温度过高或过低环境中使用本产品，使用时注意产品的通风；
- 请勿将任何液体泼溅在本产品上，禁止使用有机溶剂或腐蚀性液体清洗本产品；
- 请勿在多尘、脏乱的环境中使用本产品，如果长期不使用，请包装好本产品；
- 请勿在振动过大的环境中使用，任何跌落、敲打或剧烈晃动都可能损坏线路及元器件；
- 请勿在通电情况下，插拔核心板及外围模块(特别是串口模块)；
- 请勿自行维修、拆解本产品，如产品出现故障应及时联系本公司进行维修；
- 请勿自行修改或使用未经授权的配件，由此造成的损坏将不予保修；

2 售后维修

- 保修期限
  - 底板、核心板：三个月（非人为损坏）
  - 显示屏：七天（非人为损坏）
- 保修说明
  - 7 天内：产品（底板、核心板、屏幕）非人为损坏，本公司免费更换/维修，并承担来回运费；
  - 7 天至 3 个月内：底板、核心板非人为损坏，本公司免费维修，并承担来回运费（屏幕不提供维修）；
  - 3 个月至 1 年：底板、核心板非人为损坏或人为轻微损坏，只收更换元器件费用，免费维修，买家承担来回运费；
  - 起始时间以快递签收日为准；
- 联系方式

  官方网站：www.100ask.net
  淘宝网站：100ask.taobao.com
  地　址：广东省深圳市龙岗区布吉南湾街道平吉大道建昇大厦 B1505
  联系人：售后维修部
  电　话：0755-86200561
  邮　编：518114
  邮寄须知：保修期限内，寄回本产品请预先垫付邮费，公司不接收任何到付快递。

技术支持与开发定制

1 技术支持范围

1) 本公司提供的各类开发软件的安装，入门使用，环境搭建；
2) 本公司提供的所有裸机代码的烧写验证；
3) 本公司发布的操作系统的编译、烧写；
4) 本公司发布产品的工控板、模块的硬件原理；
5) 本公司发布的各种外设模块驱动及源码；
6) 本公司发布的配套手册在使用过程中遇到的问题；
7) 本公司产品的故障诊断及售后维修服务；

2 技术讨论范围

由于嵌入式系统知识范围广泛，涉及知识纷繁复杂，我们无法保证对各种问题都能一一解答，以下内容无法供技术支持，只能提供建议。

1) 本公司发布的教程之外的知识；
2) 非本公司发布的 U-Boot、Linux 内核的编译和移植；
3) 非本公司发布的工控板的各类驱动支持；
4) 非本公司发布的外设模块的硬件原理和驱动设计；

3 技术支持方式

1) 官方论坛发帖提问(推荐)：bbs.100ask.net
2) 官方淘宝通过阿里旺旺咨询：100ask.taobao.com
3) QQ 群咨询（QQ 群号咨询淘宝客服，需提供淘宝购买订单号验证加入）；

4 技术支持时间

星期一到星期五;上午 9:00—12:00;下午 14:00—17:30;

公司按照国家法定节假日安排休息，在此期间无法提供技术支持，请将问题发送在论坛对应板块发帖，我们将在工作日尽快给您回复。

资料获取与后续更新

1 资料的获取

- 百度网盘下载

百度网盘里面有本产品的所有配套资料，包括原理图、发布的 U-Boot、内核镜像和源码、所需的开发软件、工具等等。

进入 http://download.100ask.net/，找到对应的开发板即可。

- 视频配套教程

后续会为该工控板录制一套裸机、Linux 驱动、应用的配套付费教学视频，有需要的客户可以进入官方淘宝 100ask.taobao.com 选购。

2 后续更新

后续文档、视频等资料的更新，为了确保您的资料是最新状态，请密切关注我们的动态，我们将会通过微信公众号和 QQ 群公告推送，购买了本产品的客户请添加 QQ 群（QQ 群号咨询淘宝客服，需提供淘宝购买订单号验证加入）或关注微信公众号。

版权声明

百问科技©2022

深圳百问网科技有限公司版权所有，并保留对本手册及声明的一切权力。未得到本公司的书面许可，任何单位和个人不得以任何方式或形式对本手册内的任何部分进行复制、摘录、备份、修改、传播、翻译成其他语言、将其全部或部分用于商业用途。