

前言.....	- 12 -
版本介绍.....	- 14 -
实验 01 内核开发基础.....	- 16 -
1.1 本章导读.....	- 16 -
1.1.1 工具.....	- 16 -
1.1.2 预备课程.....	- 17 -
1.1.3 视频资源.....	- 17 -
1.2 学习目标.....	- 17 -
1.3 Linux 体系结构.....	- 17 -
1.4 Linux 内核结构.....	- 19 -
1.5 Linux 内核源码目录结构.....	- 21 -
1.6 学习方法介绍.....	- 25 -
实验 02 HelloDriverModule.....	- 28 -
2.1 本章导读.....	- 28 -
2.1.1 工具.....	- 28 -
2.1.2 预备课程.....	- 29 -
2.1.3 视频资源.....	- 29 -
2.2 学习目标.....	- 29 -
2.3 Vim 显示中文字符.....	- 29 -
2.4 Linux 内核最小模块代码分析.....	- 32 -
2.4 Linux 内核模块结构.....	- 36 -
2.5 模块加载函数.....	- 38 -

2.6 模块卸载函数.....	- 38 -
2.7 模块编译的流程.....	- 39 -
2.8 脚本文件 Makefile.....	- 40 -
2.9 实验操作.....	- 42 -
2.9.1 内核目录的确认.....	- 42 -
2.9.2 内核以及文件系统的烧写.....	- 44 -
2.9.3 烧写镜像.....	- 44 -
2.9.4 编译驱动模块.....	- 48 -
2.9.5 加载驱动.....	- 52 -
2.9.6 卸载驱动.....	- 53 -
实验 03 Menuconfig_Kconfig.....	- 56 -
3.1 本章导读.....	- 56 -
3.1.1 工具.....	- 56 -
3.2 学习目标.....	- 57 -
3.3 Linux 内核配置系统.....	- 57 -
3.4 Menuconfig 的操作.....	- 58 -
3.4.1 Menuconfig 发展历史简介.....	- 58 -
3.4.2 Menuconfig 操作方法.....	- 60 -
3.5 .config 文件和 menuconfig 的关系.....	- 70 -
3.6 Kconfig 和 menuconfig.....	- 75 -

3.7 图解 Kconfig 和 menuconfig 的关系.....	- 80 -
3.8 其它配置文件.....	- 81 -
实验 04_Makefile 编译.....	- 83 -
4.1 本章导读.....	- 83 -
4.1.1 工具.....	- 83 -
4.1.2 预备课程.....	- 83 -
4.1.3 视频资源.....	- 84 -
4.2 学习目标.....	- 84 -
4.3 编译器路径的设置.....	- 84 -
4.4 Make 内核编译命令执行过程简介.....	- 90 -
4.5 Makefile 文件.....	- 94 -
4.5.1 宏定义 LEDS_CTL 的使用.....	- 94 -
4.5.2 Makefile 脚本语法简介.....	- 96 -
4.6 Makefile 测试.....	- 99 -
4.6.1 将 LEDS 驱动不编译进内核.....	- 99 -
4.6.2 将 LEDS 驱动编译进内核.....	- 105 -
4.7 编译流程图解.....	- 107 -
实验 05 总线_设备_驱动注册流程详解.....	- 108 -
5.1 本章导读.....	- 109 -
5.1.1 工具.....	- 109 -

5.1.2 预备课程.....	- 109 -
5.1.3 视频资源.....	- 109 -
5.2 学习目标.....	- 110 -
5.3 总线、设备、驱动框架图分析.....	- 110 -
5.3.1 总线和平台总线.....	- 111 -
5.3.2 Linux 设备.....	- 113 -
5.3.3 Linux 驱动.....	- 115 -
5.3.4 Linux 驱动和设备的注册过程.....	- 115 -
5.3.5 设备节点简介.....	- 117 -
实验 06 设备注册.....	- 118 -
6.1 本章导读.....	- 119 -
6.1.1 工具.....	- 119 -
6.1.2 预备课程.....	- 119 -
6.1.3 视频资源.....	- 120 -
6.2 学习目标.....	- 120 -
6.3 在虚拟总线上注册设备.....	- 120 -
6.4 添加设备到平台总线.....	- 122 -
实验 07 驱动注册.....	- 128 -
7.1 本章导读.....	- 129 -
7.1.1 工具.....	- 129 -

7.1.2 预备课程.....	- 129 -
7.1.3 视频和代码资源.....	- 130 -
7.2 学习目标.....	- 130 -
7.3 platform_driver_register 和 platform_driver_unregister 函数.....	- 130 -
7.4 platform_driver 结构体.....	- 131 -
7.5 实验操作.....	- 135 -
实验 08_生成设备节点.....	- 140 -
8.1 本章导读.....	- 141 -
8.1.1 工具.....	- 141 -
8.1.2 预备课程.....	- 141 -
8.1.3 视频资源.....	- 142 -
8.2 学习目标.....	- 142 -
8.3 为什么引入杂项设备.....	- 142 -
8.4 杂项设备注册函数以及结构体.....	- 146 -
8.5 file_operations 结构体.....	- 147 -
8.6 实验操作.....	- 149 -
实验 09 编写简单应用调用驱动.....	- 156 -
9.1 本章导读.....	- 157 -
9.1.1 工具.....	- 157 -
9.1.2 预备课程.....	- 157 -

9.1.3 视频资源.....	- 157 -
9.2 学习目标.....	- 158 -
9.3 实验操作.....	- 158 -
实验 10-11 原理图的使用.....	- 162 -
11.1 本章导读.....	- 163 -
11.1.1 工具.....	- 163 -
11.1.2 预备课程.....	- 163 -
11.1.3 视频资源.....	- 163 -
11.2 学习目标.....	- 163 -
11.3 原理图 PDF 的操作简介.....	- 164 -
11.4 原理图——模块.....	- 166 -
11.5 原理图——元件标号.....	- 169 -
11.6 原理图——网络标号.....	- 171 -
实验 12-13 物理地址虚拟地址以及 GPIO 初始化.....	- 174 -
12.1 本章导读.....	- 175 -
12.1.1 工具.....	- 175 -
12.1.2 预备课程.....	- 175 -
12.1.3 视频资源.....	- 175 -
12.2 学习目标.....	- 176 -
12.3 单片机处理器和现代处理器.....	- 176 -

12.4 MMU 内存管理单元.....	- 177 -
12.5 物理地址虚拟地址以及 GPIO 的初始化.....	- 178 -
实验 14 LEDS 驱动一.....	- 178 -
14.1 本章导读.....	- 179 -
14.1.1 工具.....	- 179 -
14.1.2 预备课程.....	- 179 -
14.1.3 视频资源.....	- 180 -
14.2 学习目标.....	- 180 -
14.3 Led 硬件原理简单介绍.....	- 180 -
14.4 Led 管脚的调用、赋值以及配置.....	- 181 -
14.5 编写简单应用调用 LED 管脚，并测试.....	- 188 -
实验 15 LEDS 驱动二.....	- 193 -
15.1 本章导读.....	- 194 -
15.1.1 工具.....	- 194 -
15.1.2 预备课程.....	- 194 -
15.1.3 视频资源.....	- 195 -
15.2 学习目标.....	- 195 -
15.3 操作过程.....	- 195 -
实验 16 驱动模块传参数.....	- 205 -
16.1 本章导读.....	- 206 -

16.1.1 工具.....	- 206 -
16.1.2 预备课程.....	- 206 -
16.1.3 视频资源.....	- 206 -
16.2 学习目标.....	- 207 -
16.3 实验操作.....	- 207 -
实验 17 静态申请字符类设备号.....	- 215 -
17.1 本章导读.....	- 216 -
17.1.1 工具.....	- 216 -
17.1.2 预备课程.....	- 217 -
17.1.3 视频资源.....	- 217 -
17.2 学习目标.....	- 217 -
17.3 字符设备基本知识.....	- 217 -
17.4 实验操作.....	- 220 -
实验 18 动态申请字符类设备号.....	- 228 -
18.1 本章导读.....	- 229 -
18.1.1 工具.....	- 229 -
18.1.2 预备课程.....	- 229 -
18.1.3 视频资源.....	- 229 -
18.2 学习目标.....	- 230 -
18.3 实验操作.....	- 230 -

实验 19 注册字符类设备.....	- 235 -
19.1 本章导读.....	- 236 -
19.1.1 工具.....	- 236 -
19.1.2 预备课程.....	- 236 -
19.1.3 视频资源.....	- 237 -
19.2 学习目标.....	- 237 -
19.3 分配内存空间.....	- 237 -
19.4 注册字符类设备的函数.....	- 239 -
19.5 实验操作.....	- 240 -
实验 20 生成字符类设备节点.....	- 246 -
20.1 本章导读.....	- 247 -
20.1.1 工具.....	- 247 -
20.1.2 预备课程.....	- 247 -
20.1.3 视频资源.....	- 247 -
20.2 学习目标.....	- 248 -
20.3 创建设备类.....	- 248 -
20.4 创建字符设备节点.....	- 251 -
20.5 实验操作.....	- 252 -
实验 21 字符驱动.....	- 259 -
21.1 本章导读.....	- 260 -

21.1.1 工具.....	- 260 -
21.1.2 预备课程.....	- 260 -
21.1.3 视频资源.....	- 260 -
21.2 学习目标.....	- 261 -
21.3 实验操作.....	- 261 -
实验 22 字符类 GPIOS.....	- 266 -
22.1 本章导读.....	- 267 -
22.1.1 工具.....	- 267 -
22.1.2 预备课程.....	- 267 -
22.1.3 视频资源.....	- 267 -
22.2 学习目标.....	- 268 -
22.3 实验操作.....	- 268 -
实验 23 proc 文件系统.....	- 277 -
23.1 本章导读.....	- 278 -
23.1.1 工具.....	- 278 -
23.1.2 预备课程.....	- 278 -
23.1.3 视频资源.....	- 279 -
23.2 学习目标.....	- 279 -
23.3 实验操作.....	- 279 -
23.4 proc 参数介绍.....	- 282 -

实验 24 中断的基础知识.....	- 308 -
24.1 本章导读.....	- 309 -
24.1.1 工具.....	- 309 -
24.1.2 预备课程.....	- 309 -
24.1.3 视频资源.....	- 309 -
24.2 学习目标.....	- 310 -
24.3 中断的基础知识.....	- 310 -
实验 25 中断之独立按键.....	- 312 -
25.1 本章导读.....	- 313 -
25.1.1 工具.....	- 313 -
25.1.2 预备课程.....	- 313 -
25.1.3 视频资源.....	- 314 -
25.2 学习目标.....	- 314 -
25.3 中断的硬件知识和外部中断 datasheet 阅读.....	- 314 -
25.4 中断相关函数简介.....	- 318 -
25.5 实验操作.....	- 321 -
联系方式.....	- 333 -

## 前言

Linux 内核中有几百个驱动，知识点非常多而杂，所以初学者需要一个指导教程。

《iTOP-4412 驱动实验手册》是以“Linux 驱动的入门和提升教程”这一基本思想编撰而成。无论是以前学习过单片机还是进行过上位机编程的用户，都可以从这本实验手册入手学习 Linux 驱动。

学习《iTOP-4412 驱动实验手册》之前需要先学习推出的“01-烧写、编译以及基础知识视频”和“02-嵌入式 Linux 视频”。

“01-烧写、编译以及基础知识视频”对应的文字版是《iTOP-4412 开发板之精英版使用手册》；“02-嵌入式 Linux 视频”对应的文字版是《iTOP-4412 开发板之实验手册》。

对于想学习驱动的同学来说，需要尽快的掌握基础知识，没有这些基础知识，后面的学习将无法进行。

学习了前面介绍的基础知识之后，就可以开始使用《iTOP-4412 驱动实验手册》和配套视频来学习驱动的知识。

Linux 操作系统相当于“一个球”，要做的事情就是在这个球上添加驱动来实现具体的功能，不用去管这个球是从哪里开始旋转，转到什么地方了。更简单的理解就是，Linux 只是一个工具，学会使用就可以了，就像学习汽车驾驶，没有教练会从发动机原理开始介绍，只会给你发“方向盘右转一圈”“方向盘左转一圈”“拉手刹”“换挡”之类的指令。

当然学习 Linux 的最好的方法是阅读内核，但是在没有基础之前不要过多的去看内核的东西。

我们的目标其实很简单，就是学会写驱动。在嵌入式 Linux 驱动工程师的工作中，写驱动是必须掌握的技能，在学会了如何写驱动以及移植驱动，找到合适的工作之后，如果你还是对内核源码仍然感兴趣，而且还有富余的时间，可以看一看内核中“精妙”的代码，不过这对于工作并没有太多直接的帮助，可以纯粹的作为一个兴趣爱好来做。

目前实验手册已经介绍了字符驱动、杂项设备、中断、调试驱动的基本方法等等。

后面会根据实际情况渐进的增加一些驱动教程，特别是移植教程。其实学习完字符设备，就已经差不多了，需要程序员写的代码也只有字符设备的驱动，后面更多的是介绍移植的思路和方法。

迅为电子

2016.04.22

## 版本介绍

注意：《iTOP-4412 驱动实验手册》更新后，会直接上传到迅为 QQ 技术支持群的共享文件中。

当前版本为《iTOP-4412 驱动实验手册\_V1.2》

日期	改动
20170104	修改几处不严谨的地方

当前版本为《iTOP-4412 驱动实验手册\_V1.1》

更新说明 V1.1

日期	改动
2016.04.22	修改几处不严谨的地方

更新说明 V1.0

日期	改动
2015.09.08	添加内容
	前言
	版本介绍
	25 个驱动实验

	联系方法
--	------

# 实验 01 内核开发基础

## 1.1 本章导读

本实验将带您学习一遍 Linux 的框架和源码目录结构。

从任何地方拿到的 Linux 源码，都有几百 M 大小，包含上万个文件。

这么多的文件！那么问题来了，应该从什么地方入手呢？

哪些内容应该“深入研究”？

哪些内容应该“惊鸿一瞥”？

哪些内容应该“束之高阁”？

本期实验“内核开发基础”，带大家快速梳理一遍，把和学习无关的内容剔除掉。

### 1.1.1 工具

#### 1.1.1.1 硬件工具

PC 机一台

#### 1.1.1.2 软件工具

软件 Source Insight

Linux 源码 “iT0p4412\_Kernel\_3.0\_xxx”（在光盘目录“/Android 源码”文件夹下,xxx 表示日期）

### 1.1.2 预备课程

视频教程 “01-烧写、编译以及基础知识视频” → “实验 12-使用 Source Insight 加载和阅读内核源码”  
使用手册 “3.5 Source Insight 的安装和使用”

### 1.1.3 视频资源

本节配套视频为 “视频 01\_内核开发基础”

## 1.2 学习目标

本章需要学习以下内容：

理解 Linux 体系结构

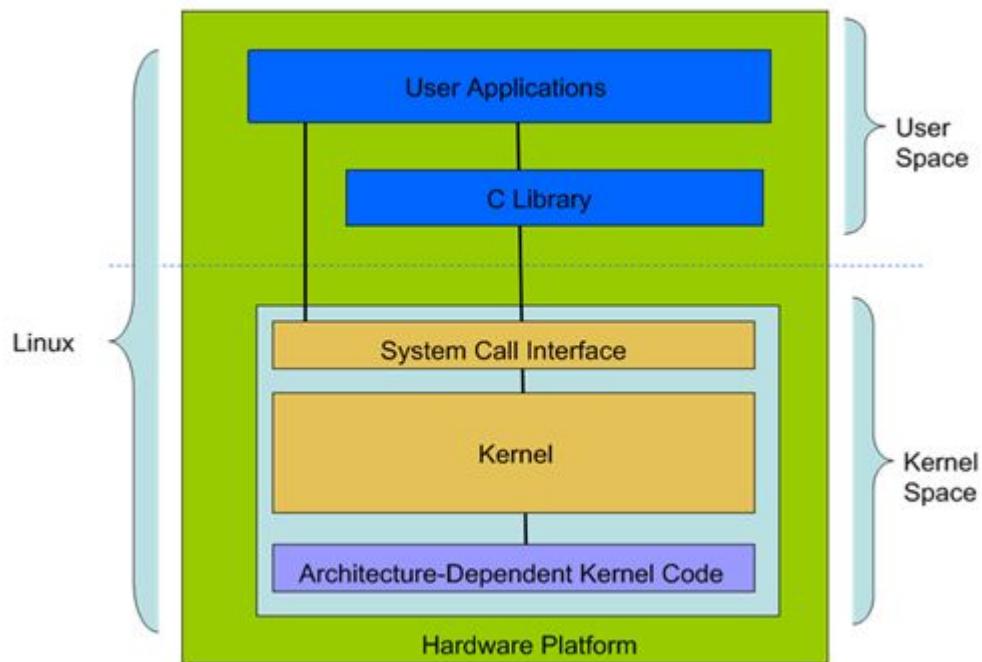
了解 Linux 内核结构

了解 Linux 内核源码目录结构

了解学习 Linux 的大方向→驱动

## 1.3 Linux 体系结构

如下图所示，Linux 体系结构，从大的方面可以分为用户空间（User Space）和内核空间（Kernel Space）。



用户空间中包含了 C 库 ,用户的的应用程序。在某些体系结构图中还包含了 shell ,当然 shell 脚本也是 Linux 体系中不可缺少的一部分。

内核空间包括硬件平台、平台依赖代码、内核、系统调用接口。

在任何一个现代操作系统中，都是分层的。为什么需要分层呢？

从程序员的角度分析，将 linux 底层和应用分开，将 linux 底层和应用分开，做应用的做应用，做底层的做底层，各干各的。经济学的基本原理是，分工产生效率。

从安全性的角度分析，是为了保护内核。现代 CPU 通常都实现了不同的工作模式。

以 ARM 为例：ARM 实现了 7 种工作模式，不同模式下 CPU 可以执行的指令或者访问的寄存器不同：  
(1) 用户模式 usr  
(2) 系统模式 sys  
(3) 管理模式 svc  
(4) 快速中断 fiq  
(5) 外部中断 irq  
(6) 数据访问终止 abt  
(7) 未定义指令异常。如果任何一个上层应用都可以调用都可以调用寄

存器，那样肯定是无法稳定执行的。而且因为出现了这个问题，出现了一个新的学科“现代操作系统”，如果大家感兴趣可以看一下“现代操作系统”相关文章或者书籍。

以 X86 为例：X86 实现了 4 个不同级别的权限，Ring0—Ring3；Ring0 下可以执行特权指令，可以访问 IO 设备；Ring3 则有很多的限制。

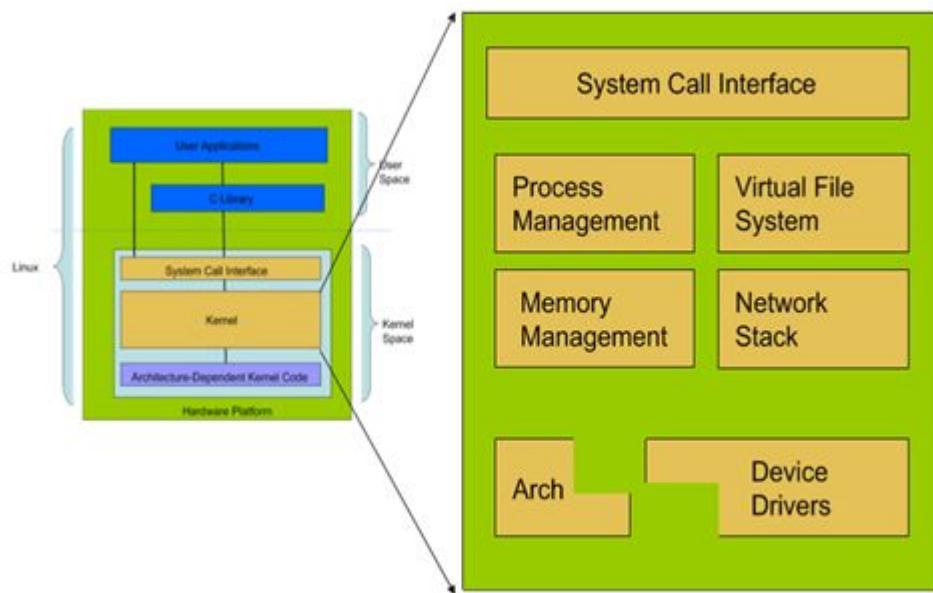
如果分析一下 Android 的，这方面做的更加“丧心病狂”，Android 所有的 APK 应用程序，都是在 Java 虚拟机上面运行，应用程序更加远离底层。

另外，用户空间和内核空间是程序执行的两种不同状态，可以通过“系统调用”和“硬件中断”来完成用户空间到内核空间的转移。

## 1.4 Linux 内核结构

这一节，分析一下内核结构。

如下图所示，是 Linux 内核结构图。

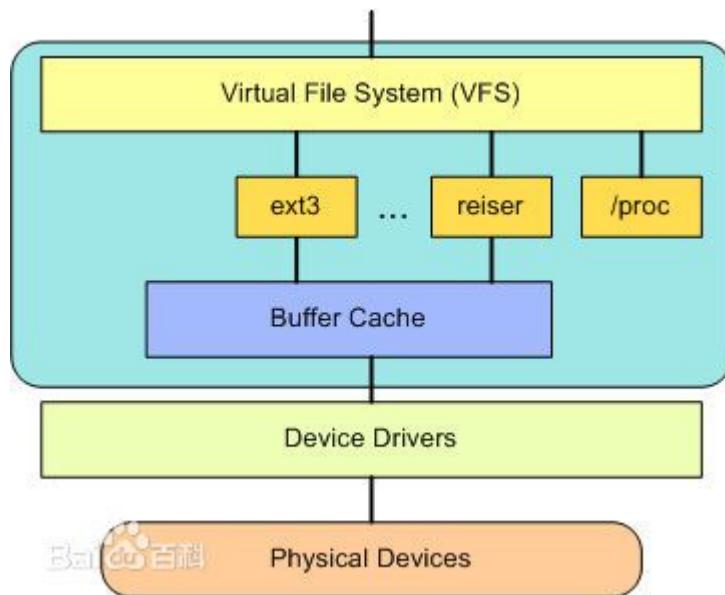


SCI 层 ( System Call Interface ) , 这一层是给应用用户空间提供一套标准的系统调用函数来访问 Linux。前面分析 Linux 体系结构的时候 , 介绍过任何一类现代操作系统都不会允许上层应用直接访问底层 , 在 Linux 中 , 内核提供了一套标准接口 , 上层应用就可以通过这一套标准接口来访问底层。

PM ( Procees Management ) , 这一部分包括具体创建进程 ( fork、 exec ) , 停止进程 ( kill、 exit ) , 并控制他们之间的通信 ( signal 等 ) 。还包括进程调度 , 控制活动进程如何共享 CPU。这一部分是 Linux 已经做好的 , 在写驱动的时候 , 只需要调用对应的函数即可实现这些功能 , 例如创建进程、进程通信等等。

MM ( Memory Management ) , 内存管理的主要作用是控制多个进程安全的共享内存区域。

VFS ( Virtual File Systems ) , 虚拟文件系统 , 隐藏各种文件系统的具体细节 , 为文件操作提供统一的接口。在 Linux 中 “一切皆文件” , 这些文件就是通过 VFS 来实现的。Linux 提供了一个大的通用模型 , 使这个模型包含了所有文件系统功能的集合。如下图所示 , 是一个虚拟文件系统的结构图。



Device Drivers 设备驱动 , 这一部分就是需要学习和掌握的。Linux 内核中有大量的代码在设备驱动程序部分 , 用于控制特定的硬件设备。

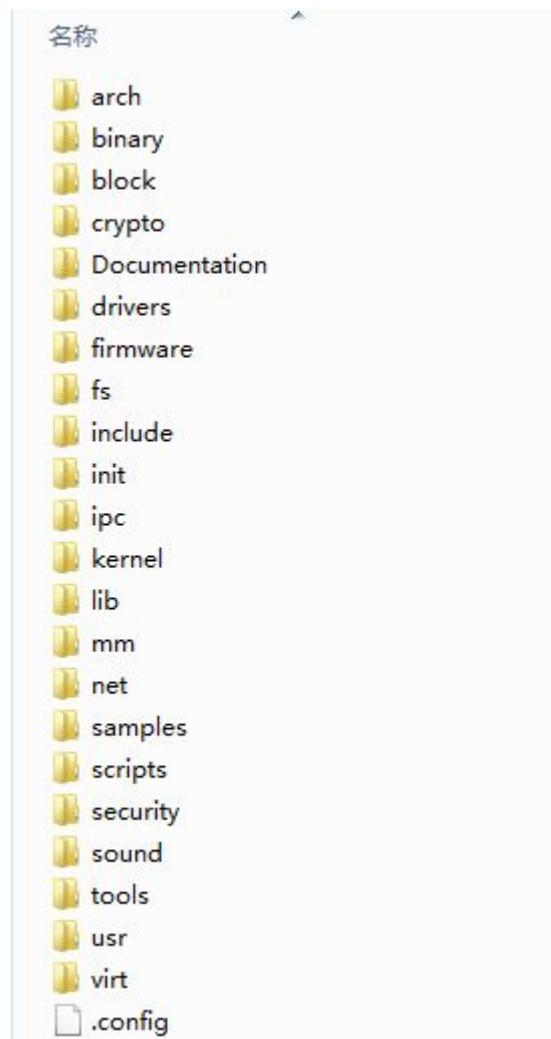
Linux 驱动一般分为网络设备、块设备、字符设备、杂项设备 , 需要编写的只有字符设备 , 杂项设备是不容易归类的一种驱动 , 杂项设备和字符设备有很多重合的地方。

网络协议栈 , Linux 内核中提供了丰富的网络协议实现。

## 1.5 Linux 内核源码目录结构

Linux 内核源码采用树形结构。功能相关的文件放到不同的子目录下面 , 使程序更具有可读行。

使用 Source Insight 打开源码，如下图所示，可以看到源码是树形结构。



下面来介绍每一个目录的作用。

arch 目录是平台目录。处理器原厂提供一套 Linux 内核的源码，那么在这个目录下都有一套针对具体处理器 CPU 的子目录。每个 CPU 的子目录，又进一步分解为 boot , mm , kernel 等子目录，分别控制系统引导，内存管理，系统调用，动态调频，主频率设置部分等。

在 arch 目录中有关键的平台文件。任何一款支持 Linux 的处理器，都有一部分内核代码是针对特定的处理器来提供的，具体的实现就是通过平台文件。

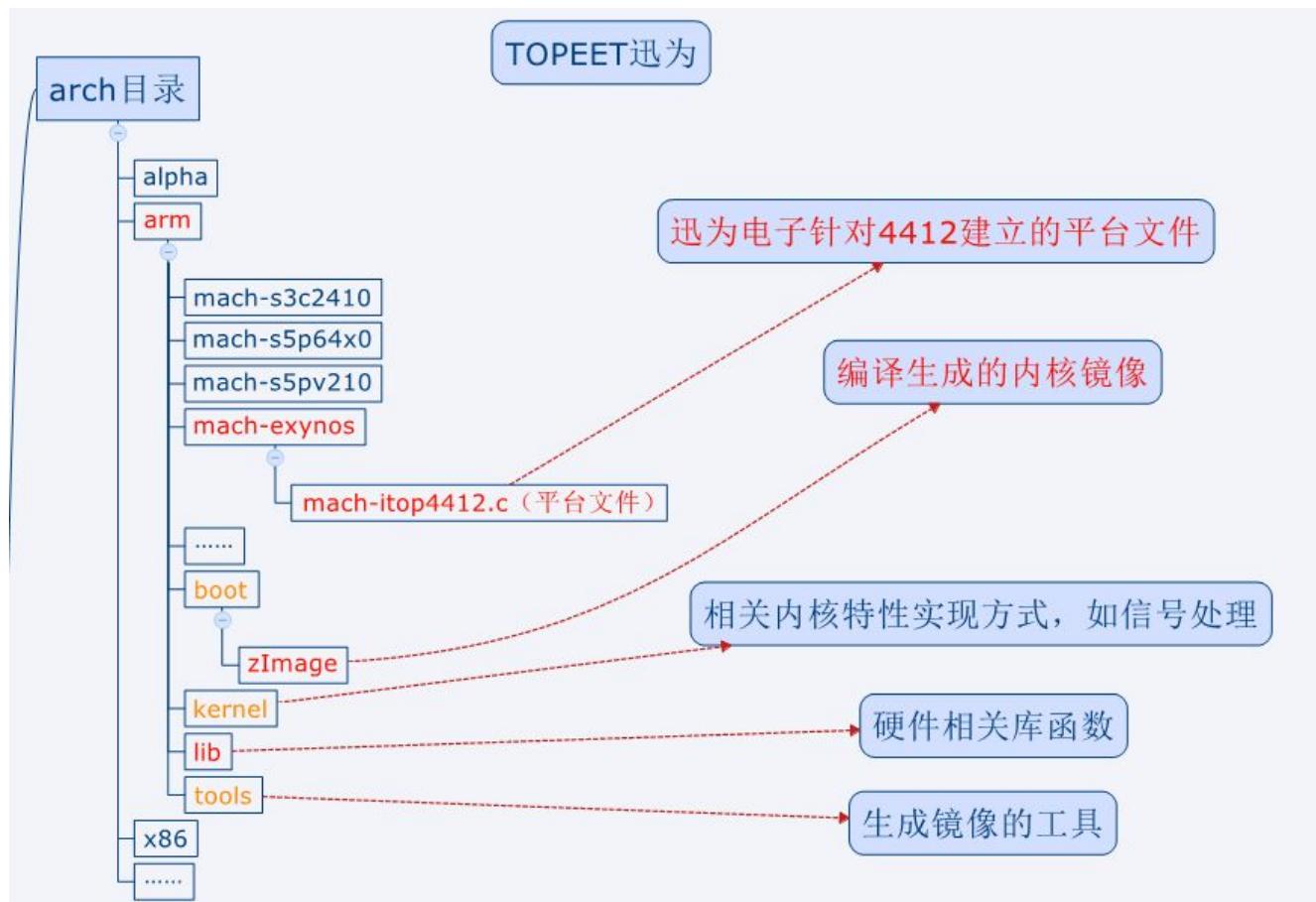
迅为 iTOP-4412 的平台文件，是 arch→arm→mach-exynos→mach-itop4412.c。

arch→arm→boot 目录，默认编译生成的内核镜像是在这个目录下。

在 arch→arm→kernel 目录中，有针对具体 CPU 处理器的代码，有相关内核特性实现方式，如信号处理等。这一部分当然是芯片厂商做好了，4412 的这部分就是三星已经做好的部分。

在 arch→arm→lib 目录中，有一些和硬件相关库函数，后面学习驱动的时候会使用到。

在 arch→arm→tools 目录中，包含了生成镜像的工具。



如下图所示。

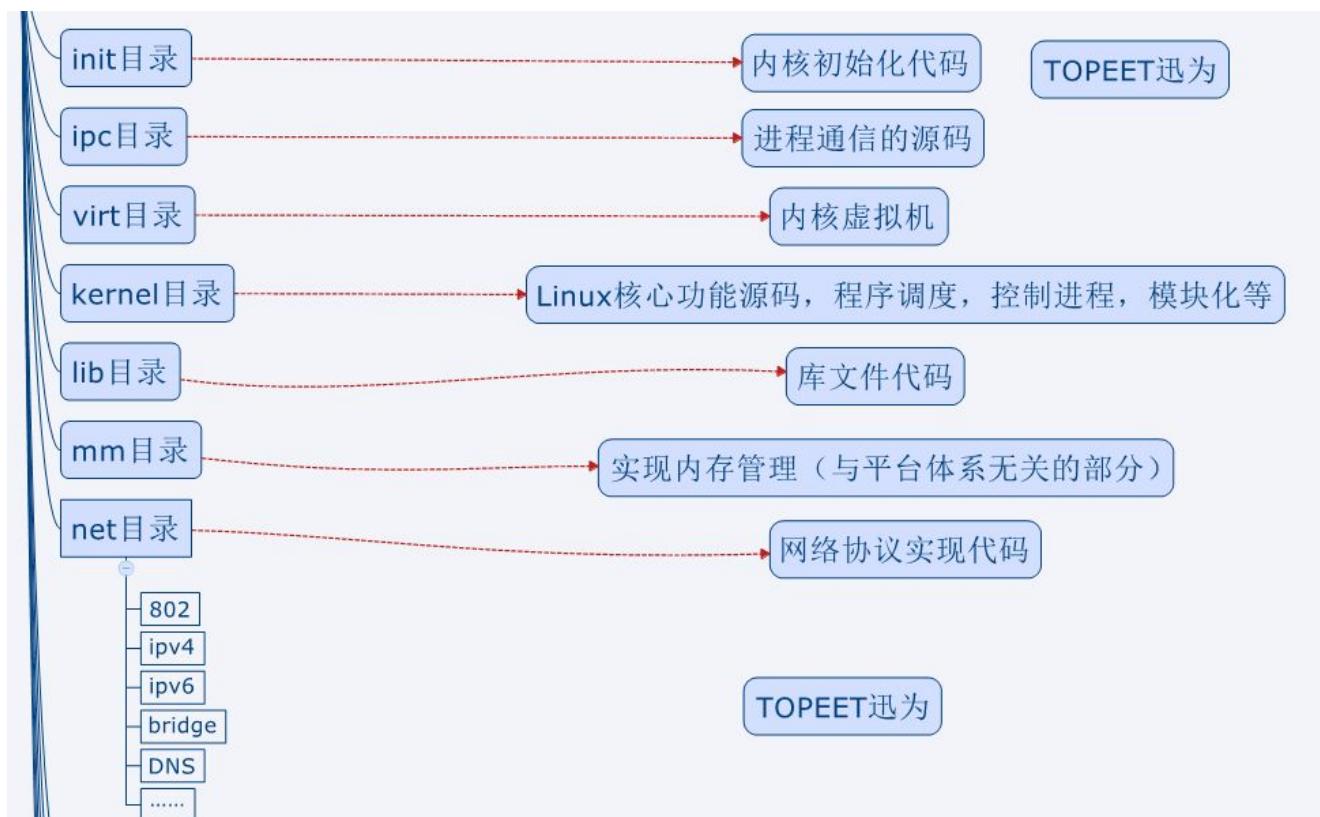
在 binary 目录中，有一些无源码的驱动以二进制放到该文件夹，例如一些测试版本或者不愿意公布源码，都可以将二进制文件放到这个目录中。

在 drivers 目录中，就是需要重点学习的部分，后面的实验都是围绕这一步进行的。

在 include 目录中，通用的 Linux 头文件都在该文件夹。

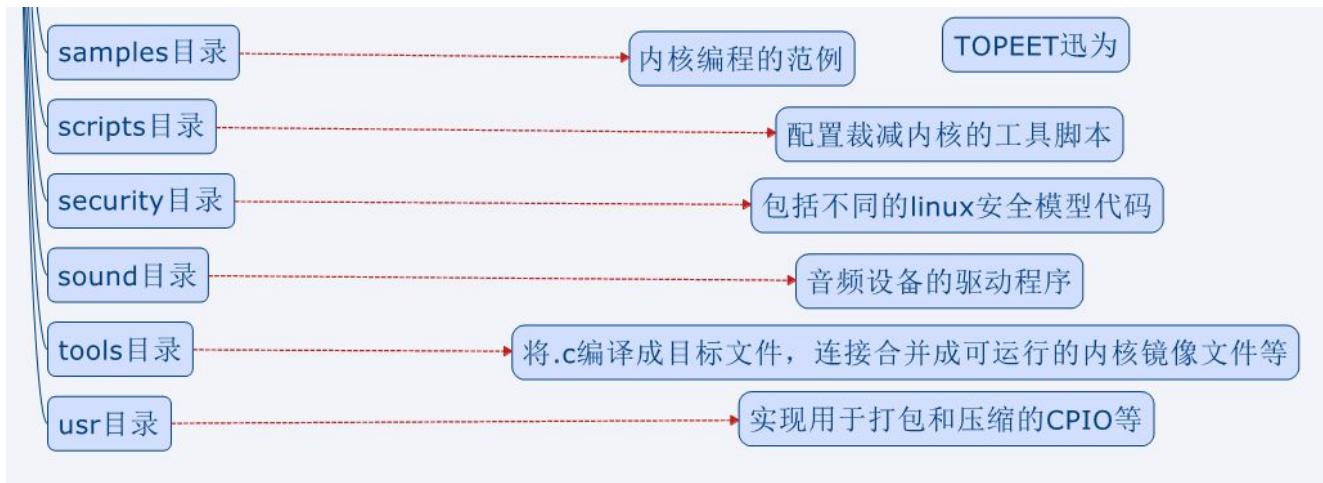


如下图所示，部分目录如下。下面的这些目录，几乎不需要去动其中任何一个文件。



如下图所示，有内核编程的范例，实现安全性的代码，声卡设备驱动等

还有内核裁减配置工具目录 tools，这一部分实现的功能是将.c 编译成目标文件，连接合并成可运行的内核镜像文件等。提供给大家的内核源码一百多 M，最后编译成的 zImage 只有不到 5M，这都是依靠这个工具来实现的，后面会有针对性的实验来教大家如何使用编译工具。



## 1.6 学习方法介绍

从事 Linux 相关工作的人员中，可以分为以下几类。

- 1 ) linux 运维工程师。Linux 服务器运用很广泛，需要一些工程师在后台维护，只需要掌握基本的电脑知识，然后熟悉服务器操作 shell 命令、网络知识就可以胜任。
- 2 ) linux 系统工程师。各种软件的安装、服务器的配置、常见故障排除等
- 3 ) Linux 应用工程师。实现应用程序，在 linux 系统中编写应用程序，需要很强的 C 语言功底和编程能力。

4 ) Linux 嵌入式驱动工程师。相比其他种类的 Linux 工程师 , Linux 嵌入式驱动工程师需要掌握各种各样的知识 , 包括 “基础的硬件知识” “测量工具万用表示波器等等” “Linux 嵌入式驱动” “Linux 应用的编写” ( 对于应用掌握的最低要求是能写驱动的测试程序 ) 。

在嵌入式范围内 , 学习 Linux 可以分为应用和驱动 , 这个教程的重点不在应用而在驱动 , 在需要用到应用程序测试的时候 , 也会介绍如何编写简单的测试代码。

这里先提醒学习者一下 , 广义的说法驱动是属于内核的 , 但是不要以为 “内核就是驱动” , 那样的就犯了 “白马非马” 的错误。大家不要觉得这问题说的很 “幼稚” , 作者在和初学者接触的过程中 , 大量的初学者上来直接问一些内核源码的问题。不可否认 , 阅读内核源码是提高能力的非常好的一种方式 , 但是对于初学者来说 , 如果你一套扎进内核 , 整天研究 “Linux 是怎么启动” , “linux 是怎么引导到内存” , “超级终端打印是怎么实现” 等等类似的问题 , 那么你将在很长一段时间内无法进步。

这里郑重提醒一下 , 在 Linux 内核中 , 驱动只是其中非常非常小的一部分 , 而且在这 “非常小的一部分” 中 , 只有极其小的一部分是需要自己写的 , 这就是字符类驱动。其它的驱动全部都是移植的 , 而且字符驱动中很大一部分也是移植的。

我们的目标不是 “星辰大海” , 而是 “溺水三千我只取一瓢” , 这 “一瓢” 虽然是很容易掌握 , 但是要到灵活应用却不是那么容易的。

这里大家要做好心理准备 , Linux 不是 “单片机” , 单片机号称可以 15 天学会 ( 其实不是号称 , 学习十五天真就能干点活了 , 本人刚参加嵌入式方面的工作前 , 就是搞了一周单片机 , 然后就找了家公司去实习了 ) 。

如果真想踏踏实实的学习 Linux , 以后想从事着方面的工作 , 最少要准备三个月到半年 , 很可能有些学习者学习到这里的时候 , 已经都过了三个月了 , 不过大家不要担心 , 只要坚持就可以学会的。

如果真要说一个简单的学习方法 , 那么就是一个问题一个问题的 “攻克” , 教程中的代码大家好好研究 , 教程中的内容搞得差不多了 , 然后在网上找一些相关文章来看 , 一步一个脚印 , 夯实了基础 , 后面的学习就会 “顺风顺水” , 找到 Linux 相关的工作也不是难事。

# 实验 02 HelloDriverModule

## 2.1 本章导读

本实验将带您走进 Linux 设备驱动的精彩世界。

Linux 设备驱动会以模块的形式出现，所以学会编写、编译、加载、卸载 Linux 内核模块是学习 Linux 驱动的先决条件。

本期实验 HelloDriverModule，以一个简单的 Linux 驱动为例，实现打印功能，让用户对 Linux 驱动模块有一个基本认识。

### 2.1.1 工具

#### 2.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 2.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）

4 ) 源码文件夹 “HelloDriverModule”

## 2.1.2 预备课程

1 ) Linux 无界面最小系统

2 ) Linux 内核的编译

3 ) Linux 常用命令

## 2.1.3 视频资源

本节配套视频为 “视频 02\_HelloDriverModule”

## 2.2 学习目标

本章需要学习以下内容：

Vim 编辑器显示中文字符

掌握 Linux 驱动的最简结构

学习以模块化形式编译驱动的方法

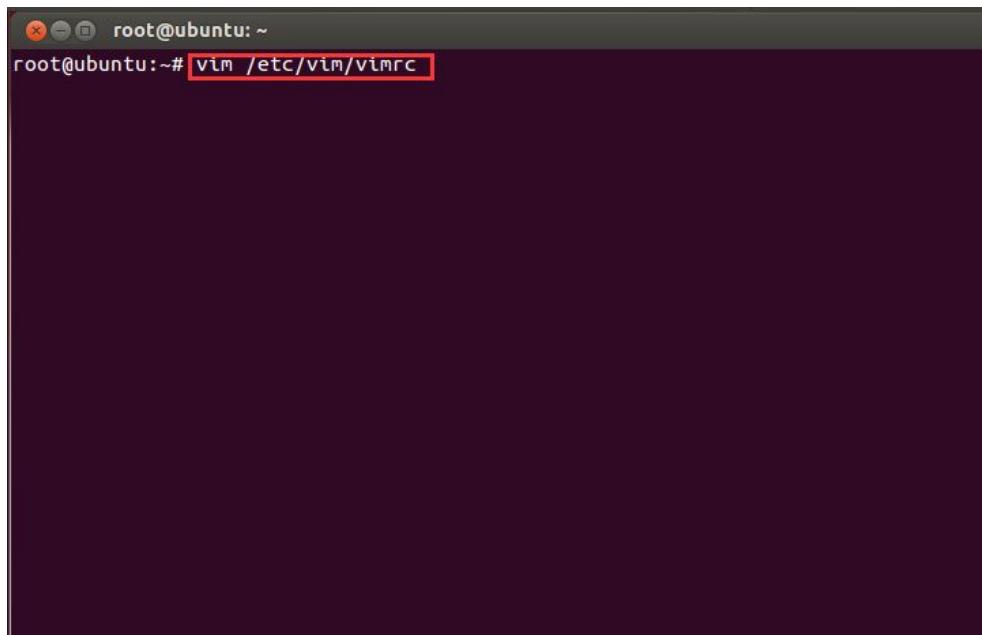
学习如何加载驱动、查看驱动、卸载驱动

## 2.3 Vim 显示中文字符

如果大家在默认的 Ubuntu 系统下使用 Vim 编辑器查看源码，中文字符会显示为乱码。

下面给大家简单介绍一下如何正常显示中文。

打开 vim 编辑器的配置文件 “/etc/vim/vimrc” , 使用命令 “vim /etc/vim/vimrc” , 如下图所示。



如下图所示，进入到最底行，右下角会显示“Bot”。

```
" Uncomment the following to have Vim load indentation rules and plugins
" according to the detected filetype.
"if has("autocmd")
"   filetype plugin indent on
"endif

" The following are commented out as they cause vim to behave a lot
" differently from regular Vi. They are highly recommended though.
"set showcmd          " Show (partial) command in status line.
"set showmatch         " Show matching brackets.
"set ignorecase        " Do case insensitive matching
"set smartcase         " Do smart case matching
"set incsearch          " Incremental search
"set autowrite         " Automatically save before commands like :next and :mak
e
"set hidden            " Hide buffers when they are abandoned
"set mouse=a           " Enable mouse usage (all modes)

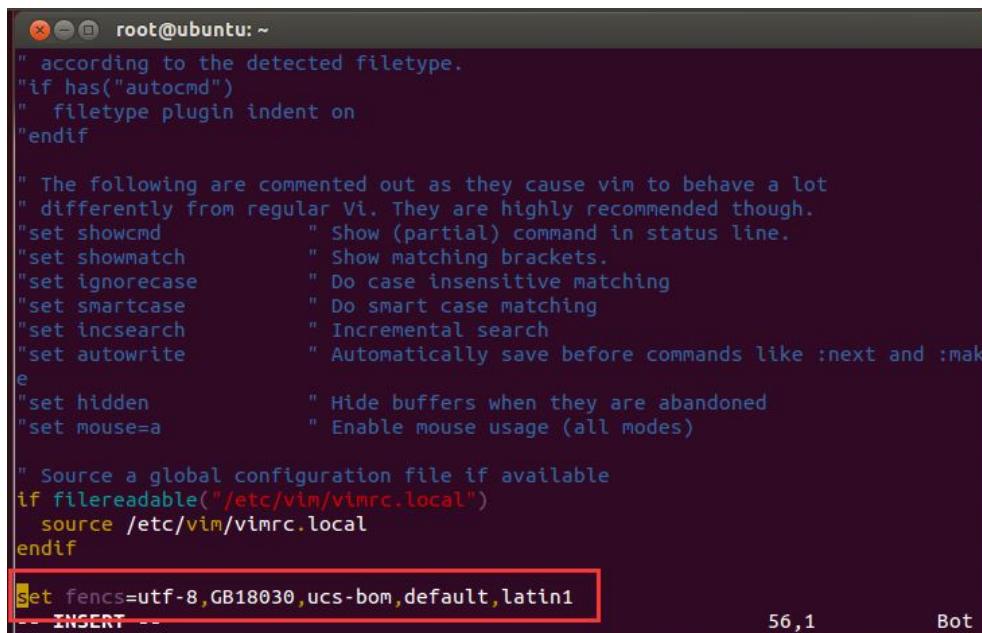
" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif
```

A screenshot of a terminal window titled "root@ubuntu: ~". It displays the contents of the "/etc/vim/vimrc" file. A red arrow points from the left side of the image towards the bottom right corner of the terminal window. In the bottom right corner, the word "Bot" is visible, indicating the current line number. The terminal window has a dark background and light-colored text.

在文件 “/etc/vim/vimrc” 最底端添加以下代码。

```
set fencs=utf-8,GB18030,ucs-bom,default,latin1
```

添加完成之后，如下图所示。



```
root@ubuntu: ~
" according to the detected filetype.
"if has("autocmd")
"  filetype plugin indent on
"endif

" The following are commented out as they cause vim to behave a lot
" differently from regular Vi. They are highly recommended though.
"set showcmd          " Show (partial) command in status line.
"set showmatch         " Show matching brackets.
"set ignorecase        " Do case insensitive matching
"set smartcase         " Do smart case matching
"set incsearch          " Incremental search
"set autowrite          " Automatically save before commands like :next and :mak
e
"set hidden             " Hide buffers when they are abandoned
"set mouse=a            " Enable mouse usage (all modes)

" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif

Set fencs=utf-8,GB18030,ucs-bom,default,latin1
-- INSERT --           56,1           Bot
```

最后保存退出，Vim 编辑器就可以正常显示中文字符。

如下图所示，下面截图的为本节的驱动代码，大家可以看到中文字符可以正常显示。这一步不用测试，到了本章节后面的时候可以使用 Vim 编辑器测试一下。



```
#include <linux/init.h>
/*包含初始化宏定义的头文件，代码中的module_init和module_exit在此文件中*/
#include <linux/module.h>
/*包含初始化加载模块的头文件，代码中的MODULE_LICENSE在此头文件中*/

MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/

static int hello_init(void)
{
    printk(KERN_EMERG "Hello World enter!\n");
    /*打印信息， KERN_EMERG表示紧急信息*/
    return 0;
```

## 2.4 Linux 内核最小模块代码分析

Linux 内核的整体框架非常大，提供给大家的内核源码，大家应该见识过了，包含了各种各样的组件，而且有 1 万多个文件，那么这些文件是怎么添加到内核中的呢？

Linux 内核针对驱动的处理有以下两种方式：

第一种方式：把所有需要的功能全部编译到内核中，这种方式会导致重新添加或者删除功能的时候，需要重新编译内核。

第二种方式：动态的添加模块，也就是这个实验要介绍的“模块的方式添加驱动”。

如下图所示，这是一个非常简单的驱动代码。源码文件“mini\_linux\_module.c”大家可以在视频目录“02\_HelloDriverModule”中找到。

mini\_linux\_module.c 中代码如下。

```
#include <linux/init.h>
/*包含初始化宏定义的头文件,代码中的module_init和module_exit在此文件中*/
#include <linux/module.h>
/*包含初始化加载模块的头文件,代码中的MODULE_LICENSE在此头文件中*/
```

```
MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的,没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/
```

声明模块信息

头文件

```
static int hello_init(void)
{
    printk(KERN_EMERG "Hello World enter!\n");
    /*打印信息, KERN_EMERG表示紧急信息*/
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_EMERG "Hello world exit!\n");
}
```

功能区

```
module_init(hello_init);
/*初始化函数*/
module_exit(hello_exit);
/*卸载函数*/
```

模块的入口

模块的出口

这个内核模块只有简单的几个功能，加载模块、卸载模块、对 GPL 的声明、描述信息例如作者以及加载和卸载时候的打印函数。

可以看到这个文件中的代码被红色框图划分为 5 大块，分别是：

第一部分：必须包含的头文件 `linux/init.h` 和 `linux/module.h`，想要编译成模块就必须使用这两个头文件。

第二部分：驱动申明区。

在所有的声明中下面这一句最重要。

`MODULE_LICENSE("Dual BSD/GPL");`

/\*声明是开源的，没有内核版本限制\*/

这一句代码声明你遵循 GPL 协议，否则模块在加载的时候，内核会提示被污染的“警报”。

MODULE\_AUTHOR("iTOPEET\_dz");

/\*声明作者\*/

作者的声明，声明这个驱动代码是谁写的。这个申明不是必须的。

第三部分：功能区代码。

在功能区里面，加载驱动和卸载驱动时候调用的函数，这两个函数都只是调用了 printk 函数。

第四部分：模块的入口。

加载模块。采用动态的方式添加驱动到内核中，添加驱动的入口就是这个函数。

加载的时候调用了功能区的函数 static int hello\_init(void)

这里特别提醒只接触过单片机用户或者那些没有接触过操作系统的用户，你会发现这里并没有 main 函数。

Linux 操作系统相当于“一个球”，要做的事情就是在这个球上添加驱动来实现具体的功能，不用去管这个球是从哪里开始旋转，转到什么地方了。更简单的理解就是，Linux 只是一个工具，学会使用就可以了，就像学习汽车驾驶，没有教练会从发动机原理开始介绍，只会教“方向盘右转一圈” “方向盘左转一圈” “拉手刹” “换挡”之类的。

如果你非要找一个 main 函数才罢休，那么你可以把模块的初始化函数当做 main 函数，也是模块的入口。

第五部分：模块的出口。

卸载模块。采用动态添加驱动的方式来加载驱动，那么也可以动态的卸载这个驱动，执行的是和添加驱动相反的一个过程。

卸载的时候调用了功能区的函数 static void hello\_exit(void)

编译模块的时候会生成一个 ko 文件，可以使用 insmod 和 rmmod 加载和卸载它。加载的时候会打印 “Hello World enter!” 卸载的时候会打印 “Hello world exit!” 。

这里针对打印函数 printk 做一个简单说明，printk 函数是在内核中使用的，不是用户空间的 printf 函数，不过用法类似。

这个函数在写驱动的时候，几乎都会用到，这是一个必须掌握的调试方法，也是最基本的方法。在后面的例子中，会慢慢的介绍到其它的调试方法，到后期再给大家做个调试 Debug 方法的总结。因为最终的目标是用户能够自己写代码和调试驱动，那么这个 Debug 能力是一定要具备的。

打印函数 printk 向超级终端传递数据，KERN\_EMERG 是紧急情况的标识，加载和卸载驱动模块的时候，分别打印输出 “Hello World enter!” “Hello world exit!” 。

打印函数 printk 是分级的，它的 8 个级别如下：

```
#define KERN_EMERG 0
```

```
/*紧急事件消息，系统崩溃之前提示，表示系统不可用*/
```

```
#define KERN_ALERT 1
```

```
/*报告消息，表示必须立即采取措施*/  
  
#define KERN_CRIT 2  
  
/*临界条件，通常涉及严重的硬件或软件操作失败*/  
  
#define KERN_ERR 3  
  
/*错误条件，驱动程序常用 KERN_ERR 来报告硬件的错误*/  
  
#define KERN_WARNING 4  
  
/*警告条件，对可能出现问题的情况进行警告*/  
  
#define KERN_NOTICE 5  
  
/*正常但又重要的条件，用于提醒*/  
  
#define KERN_INFO 6  
  
/*提示信息，如驱动程序启动时，打印硬件信息*/  
  
#define KERN_DEBUG 7  
  
/*调试级别的消息*/
```

## 2.4 Linux 内核模块结构

一个 Linux 内核模块主要由以下几个部分组成。

### 1 ) 模块加载函数

当通过 insmod 命令加载内核模块的时候，模块的加载函数会自动被调用到内核运行，完成模块的初始化工作

### 2 ) 模块卸载函数

当通过 rmmod 命令卸载内核模块的时候，模块的卸载函数会自动被调用到内核运行，完成模块的卸载工作，完成与模块卸载函数相反的功能。

### 3 ) 模块的许可声明

LICENSE 声明描述内核的许可权限，如果不声明 LICENSE，模块被加载的时候，会受到内核被污染的警告。

Linux 在使用的过程中，需要接受 GPL 协议，体现在代码中就是添加类似 MODULE\_LICENSE("Dual BSD/GPL") 的语句

在 Linux 内核模块领域，可接受的 LICENSE 参数包括 "GPL"、"GPL v2"、"GPL and addtional"、"Dual BSD/GPL"、"Dual MPL/GPL" 等，当参数是前面的几个时，那么就表明你遵循 GPL 协议。

还有一个参数"Proprietary"，这个参数表明是私有的，除非你的模块显式地声明一个开源版本，否则内核也会默认你这是一个私有的模块(Proprietary)。

### 4 ) 模块作者信息等

体现在代码中，就是类似 MODULE\_AUTHOR("iTOPEET\_dz"); 的语句。其中 iTOPEET\_dz 是参数。

## 小贴士：GPL 协议简介

GPL，是 General Public License 的缩写，是一份 GNU 通用公共授权非正式的中文翻译。GPL 协议中一个很核心的内容是：如果你接受这个协议，那么你就可以免费使用 Linux 中的代码，当你免费使用 Linux 的代码开发出新的代码，那么你就应该以源码或者二进制文件的方式

免费发布；如果你不接受这个协议，那么你就无权使用 Linux 源码。更加详细的内容，读者可以上网查一下，关键词是“Linux GPL 协议”。

## 2.5 模块加载函数

模块的加载函数“module\_init(function)”，返回整数型，如果执行成功，则返回 0。否则返回错误信息。

有时候芯片供应商并不提供芯片驱动的源码，只是提供驱动 module 的 ko 文件，这个时候就需要调用 request\_module(module\_name) 来加载驱动。

在 Linux 中，标示为 \_\_init 的函数都是初始化函数，这些函数占用的内存空间，在 Linux 启动或者模块加载初始化之后，是会被释放掉的。除了函数，数据也是可以被定义为 \_\_ 数据，这些数据在 linux 启动或者模块加载初始化之后，也是会被释放掉。这一部分的知识，在后面会逐渐使用到。

## 2.6 模块卸载函数

模块卸载就是模块加载的“逆向过程”，比较容易理解。

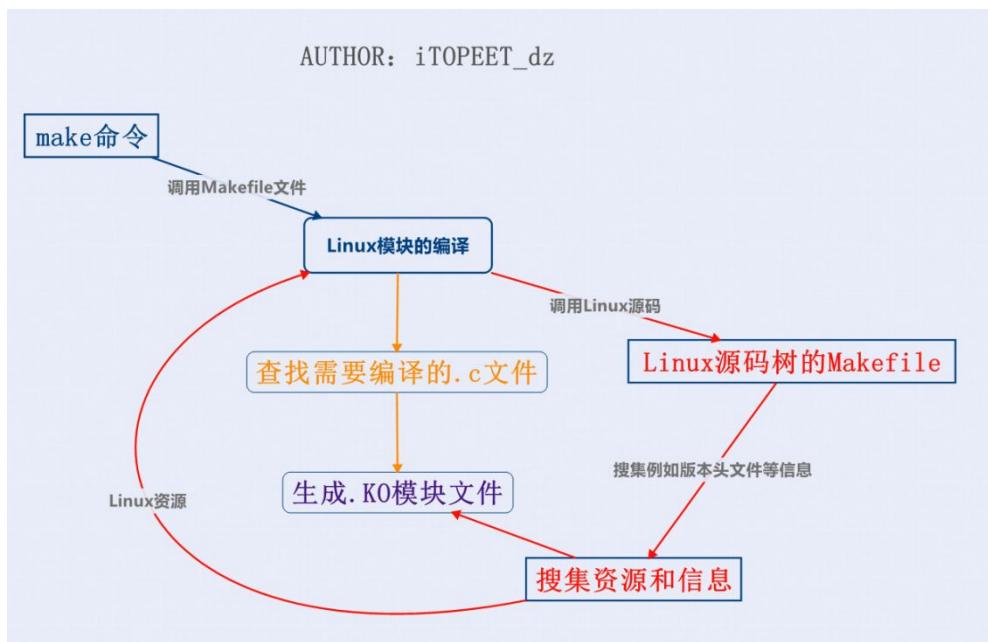
模块的卸载函数“module\_init(function)”不返回任何值。

一般说来，在卸载函数中要完成和加载函数中相反的功能，例如你调用了系统或者硬件的资源，那么你在卸载函数中，就需要释放掉。

## 2.7 模块编译的流程

Linux 模块一般是使用脚本语言来编译，脚本语言的种类非常多，语法丰富，不过只需要学会使用即可，能够仿照着写就可以了，这并不影响开发。

如下图所示，模块的编译流程图。



通过上图可以看到编译中，当执行执行 make 命令之后，调用 Makefile 文件进行 Linux 模块的编译。

Linux 模块的编译分为两个条线。

第一条红色的线：进入 Linux 源码中，调用版本信息以及一些头文件等。

这一条线经过的是整个 Linux 的源码文件。

第二条橙色的线：搜集完 Linux 源码树的信息之后，Makefile 继续执行，调用编译.KO 文件的源码文件，这里是 mini\_linux\_module.c 这个 “iTop4412\_Kernel\_3.0” 整个源码。

这一条线走的是 mini\_linux\_module.c，虽然都是源码，但是此源码非彼源码。

Makefile 文件通过执行上面两条线，通过搜集到信息，最终编译生成.KO 模块

这里要学习和理解的重点就是，编译模块也必须用到内核的源码，因为这涉及到内核版本以及头文件。如果版本不对，那么模块有可能无法加载和运行；如果没有头文件，编译就无法通过。

## 2.8 脚本文件 Makefile

在单片机或者上位机编程的时候，都有集成开发工具。程序员按照开发工具的规则，将代码放入指定的位置，通常是一个 main.c 文件加上很多.c 文件，代码写好了，开发工具中某个按钮一点，就给你自动编译成了二进制文件。

在 Linux 中，并没有这样的集成开发工具，这里还需要自己写编译文件 Makefile。

编译文件一般是用脚本编写，脚本成千上万，脚本语言学也学不完，脚本的学习最好是用到哪里学到哪。

Makefile 编译文件如下，下面的这个文件可以在视频目录 “02\_HelloDriverModule” 找到。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += mini_linux_module.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)}Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)}当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

如上图所示，就是编译 mini\_linux\_module 的脚本文件。下面详细的介绍每一句的含义。

**#!/bin/bash**

通知编译器这个脚本使用的是那个脚本语言

**obj-m += mini\_linux\_module.o**

这是一个标准用法，表示要将 mini\_linux\_module.c 文件编译成 mini\_linux\_module.o 文件，如果还需要编译其它的文件，则在后面添加即可。

**KDIR := /home/topeet/android4.0/iTop4412\_Kernel\_3.0**

这一行代码表示内核代码的目录，如果没有内核源码，那么模块的编译无法进行，因为会缺乏版本支持和头文件。KDIR 是一个变量。

**PWD ?= \$(shell pwd)**

这一句是提供一个变量，然后将当前目录的路径传给这个变量。pwd 是一个命令，表示当前目录，PWD 是一个变量。

all:

```
make -C $(KDIR) M=$(PWD) modules
```

在使用执行脚本编译命令 “make” 的时候，它会默认来寻找这一句，make -C 表示调用执行的路径，也就是变量 KDIR，变量 KDIR 中有内核源码目录的路径。  
PWD 表示当前目录。

modules 表示将驱动编译成模块的形式，也是就是最终生成 KO 文件。

clean:

```
rm -rf *.o
```

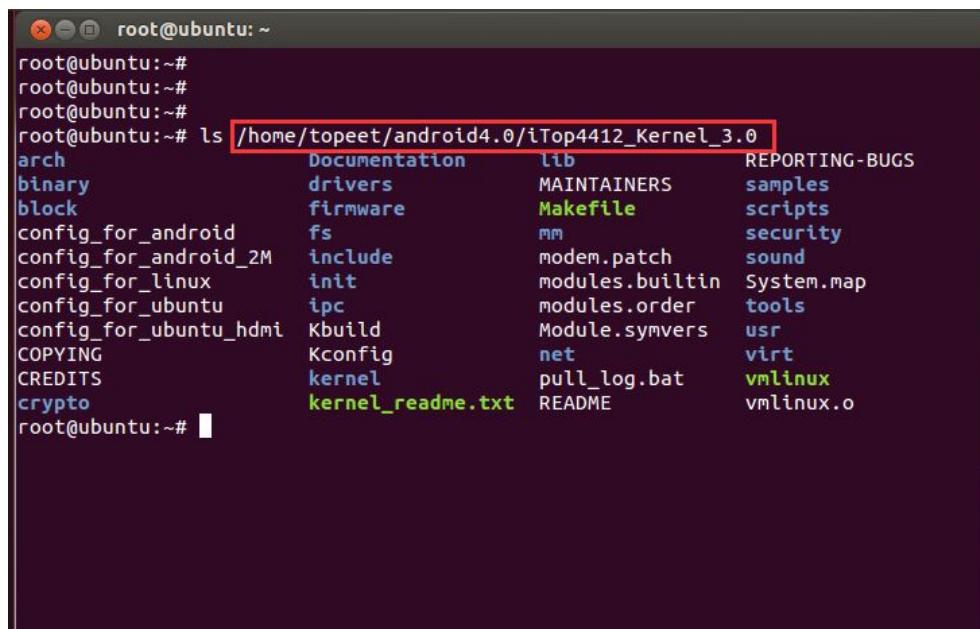
在重新修改了源码之后，可以执行 “make clean” 命令来清除一些无用的中间文件，这里选择的是清除后缀为 “o” 的文件。

## 2.9 实验操作

### 2.9.1 内核目录的确认

用户必须要确认内核源码解压的目录，如果目录不正确，模块是无法编译的。

如下图所示，作者的内核源码目录是 “/home/topeet/android4.0/iTop4412\_Kernel\_3.0” 中。



```
root@ubuntu:~# ls /home/topeet/android4.0/iTop4412_Kernel_3.0
arch           Documentation      lib          REPORTING-BUGS
binary         drivers           MAINTAINERS   samples
block          firmware          Makefile     scripts
config_for_android  fs            mm          security
config_for_android_2M include        init        modem.patch
config_for_linux    ipc           modules.builtin System.map
config_for_ubuntu   kernel        modules.order  tools
config_for_ubuntu_hdmi Kbuild       Module.symvers  usr
COPYING         Kconfig        net          virt
CREDITS        kernel        pull_log.bat  vmlinuz
crypto          kernel_readme.txt README      vmlinuz.o
root@ubuntu:~#
```

确认了内核目录，那么 Makefile 文件中就需要针对性修改引用的内核目录。如下图所示，脚本文件 Makefile 中变量 “KDIR” 的值就是

“/home/topeet/android4.0/iTop4412\_Kernel\_3.0”。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += mini_linux_module.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)}Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)}当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

确认完之后，就可以进行下一步了。

## 2.9.2 内核以及文件系统的烧写

用户需要针对性的使用内核以及文件系统，教程是基于最小系统实现的。

首先 uboot 不用修改，也不用去烧写。

其次是内核，如下图所示，内核需要用到用户光盘目录 “iTOP-4412 精英版光盘资料\04\_镜像\_QT 文件系统\zImage” (如果是 SCP 的核心板则使用 SCP 目录下的，POP 的则使用 POP 目录下的)下的 “zImage” 。

最后是文件系统，如下图所示，使用的是最小文件系统，一个是启动文件 “ramdisk-uboot.img” ，一个是最小文件系统 “system.img” ，它们在 “视频 02-DriverModule\_01\最小系统” 中。

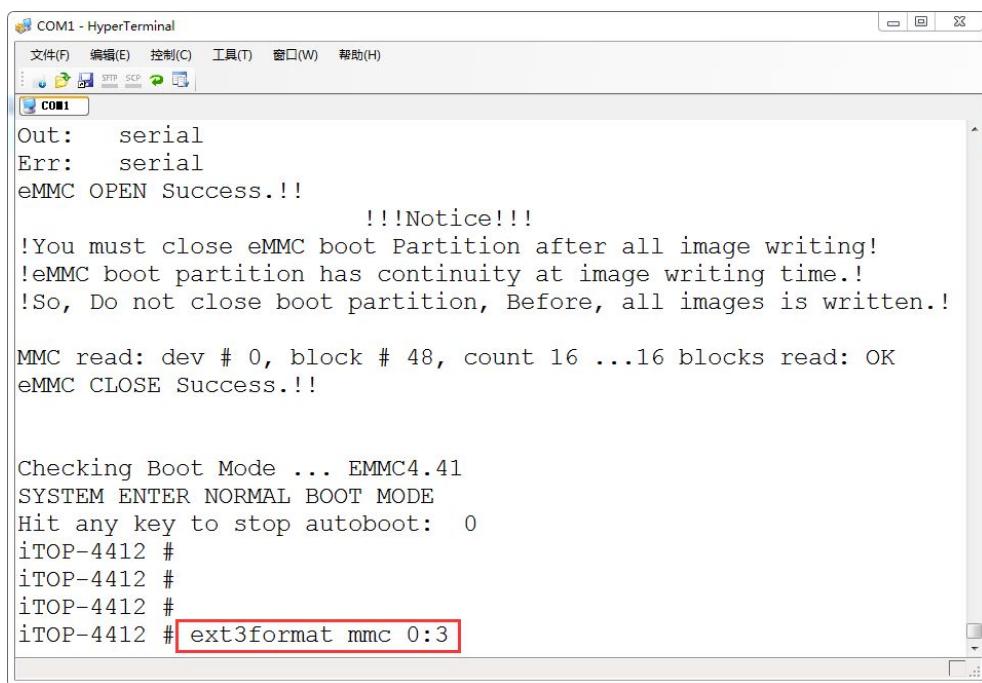
名称	修改日期	类型	大小
ramdisk-uboot.img	2015/7/3 ...	光盘映像...	637 KB
Read Me.txt	2015/8/3 ...	TXT 文件	1 KB
system.img	2014/10/...	光盘映像...	15,19...

## 2.9.3 烧写镜像

如果用户当前使用的是其它系统，那么则需要重新烧写一下。

下面介绍一下烧写过程。

在启动开发板之后，如下图所示，使用擦除命令 “ext3format mmc 0:3” 。

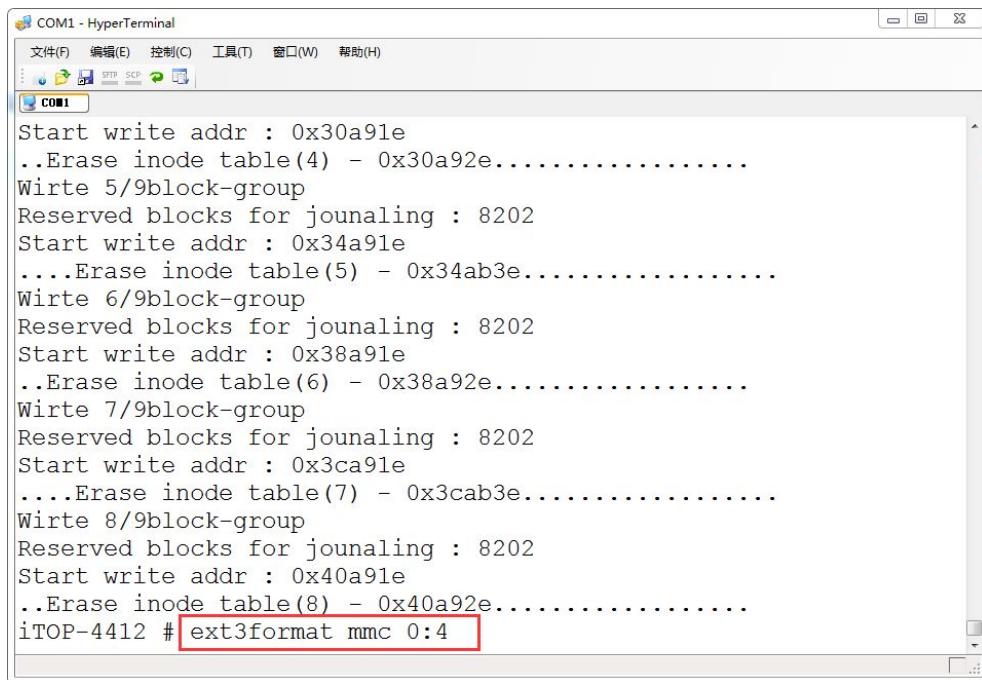


```
COM1 - HyperTerminal
文件(F) 编辑(E) 控制(C) 工具(T) 窗口(W) 帮助(H)
COM1
Out: serial
Err: serial
eMMC OPEN Success.!!
!!!Notice!!!
!You must close eMMC boot Partition after all image writing!
!eMMC boot partition has continuity at image writing time.!
!So, Do not close boot partition, Before, all images is written.!

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
eMMC CLOSE Success.!!

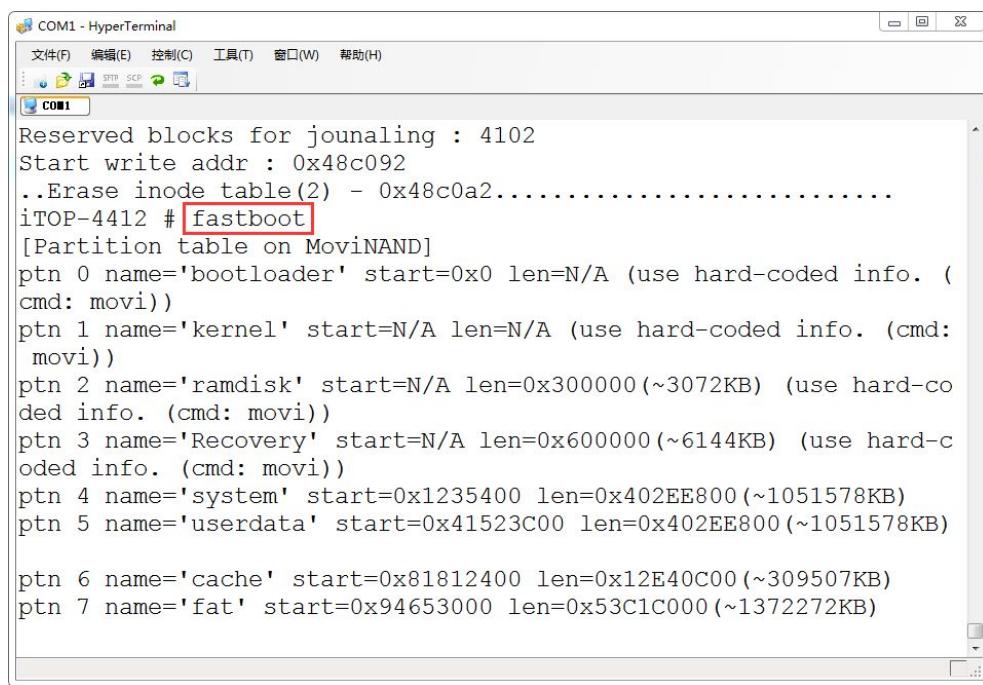
Checking Boot Mode ... EMMC4.41
SYSTEM ENTER NORMAL BOOT MODE
Hit any key to stop autoboot: 0
iTOP-4412 #
iTOP-4412 #
iTOP-4412 #
iTOP-4412 # ext3format mmc 0:3
```

上图中的命令执行完毕之后，如下图所示，使用擦除命令“ext3format mmc 0:4”。



```
COM1 - HyperTerminal
文件(F) 编辑(E) 控制(C) 工具(T) 窗口(W) 帮助(H)
COM1
Start write addr : 0x30a91e
..Erase inode table(4) - 0x30a92e.....
Wirte 5/9block-group
Reserved blocks for journaling : 8202
Start write addr : 0x34a91e
....Erase inode table(5) - 0x34ab3e.....
Wirte 6/9block-group
Reserved blocks for journaling : 8202
Start write addr : 0x38a91e
....Erase inode table(6) - 0x38a92e.....
Wirte 7/9block-group
Reserved blocks for journaling : 8202
Start write addr : 0x3ca91e
....Erase inode table(7) - 0x3cab3e.....
Wirte 8/9block-group
Reserved blocks for journaling : 8202
Start write addr : 0x40a91e
....Erase inode table(8) - 0x40a92e.....
iTOP-4412 # ext3format mmc 0:4
```

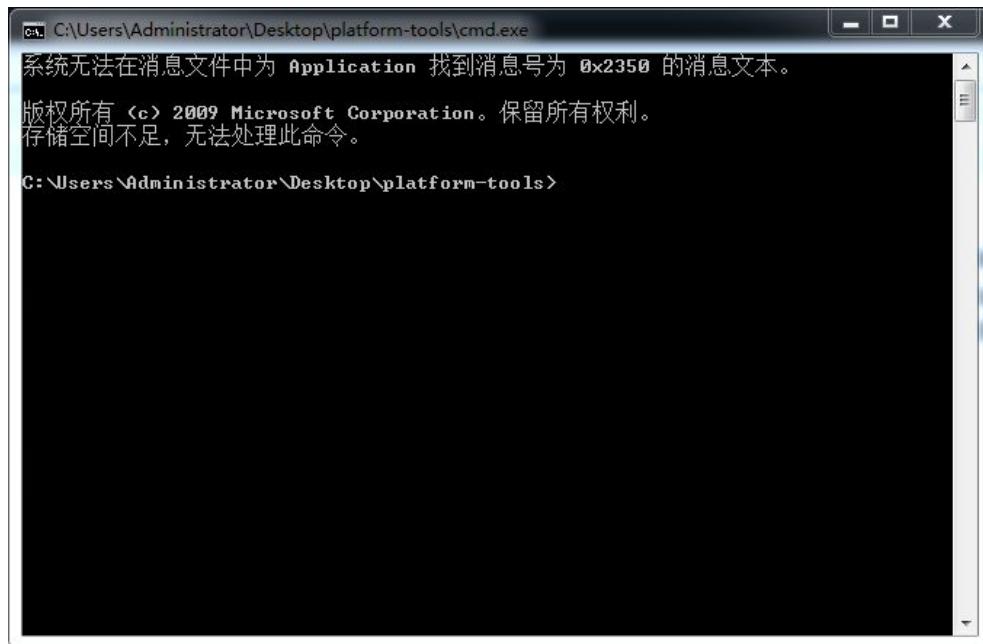
上图中的命令执行完毕之后，如下图所示，执行“fastboot”命令，连接PC机器。这里需要将开发板的OTG接口和PC机的USB连接。



Reserved blocks for journaling : 4102  
Start write addr : 0x48c092  
.Erase inode table(2) - 0x48c0a2.....  
iTOP-4412 # **fastboot**  
[Partition table on MoviNAND]  
ptn 0 name='bootloader' start=0x0 len=N/A (use hard-coded info. (cmd: movi))  
ptn 1 name='kernel' start=N/A len=N/A (use hard-coded info. (cmd: movi))  
ptn 2 name='ramdisk' start=N/A len=0x300000 (~3072KB) (use hard-coded info. (cmd: movi))  
ptn 3 name='Recovery' start=N/A len=0x600000 (~6144KB) (use hard-coded info. (cmd: movi))  
ptn 4 name='system' start=0x1235400 len=0x402EE800 (~1051578KB)  
ptn 5 name='userdata' start=0x41523C00 len=0x402EE800 (~1051578KB)  
  
ptn 6 name='cache' start=0x81812400 len=0x12E40C00 (~309507KB)  
ptn 7 name='fat' start=0x94653000 len=0x53C1C000 (~1372272KB)

将上一小节中介绍的内核文件以及文件系统等三个文件拷贝到烧写文件夹

“platform-tools” 中，打开 “cmd.exe” 程序，如下图所示。



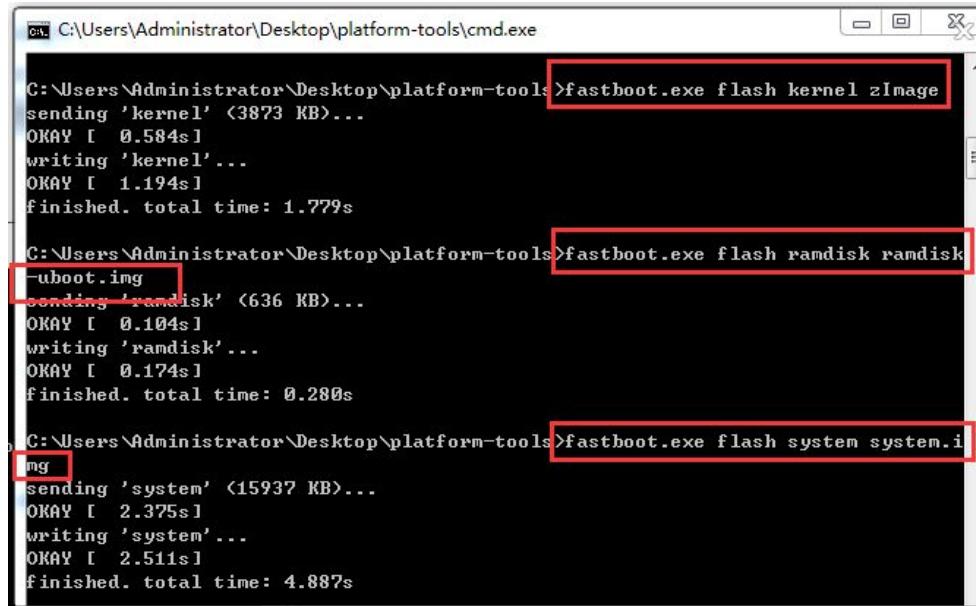
依次输入烧写命令

“fastboot.exe flash kernel zImage”

“fastboot.exe flash ramdisk ramdisk-uboot.img”

“fastboot.exe flash system system.img”

如下图所示。

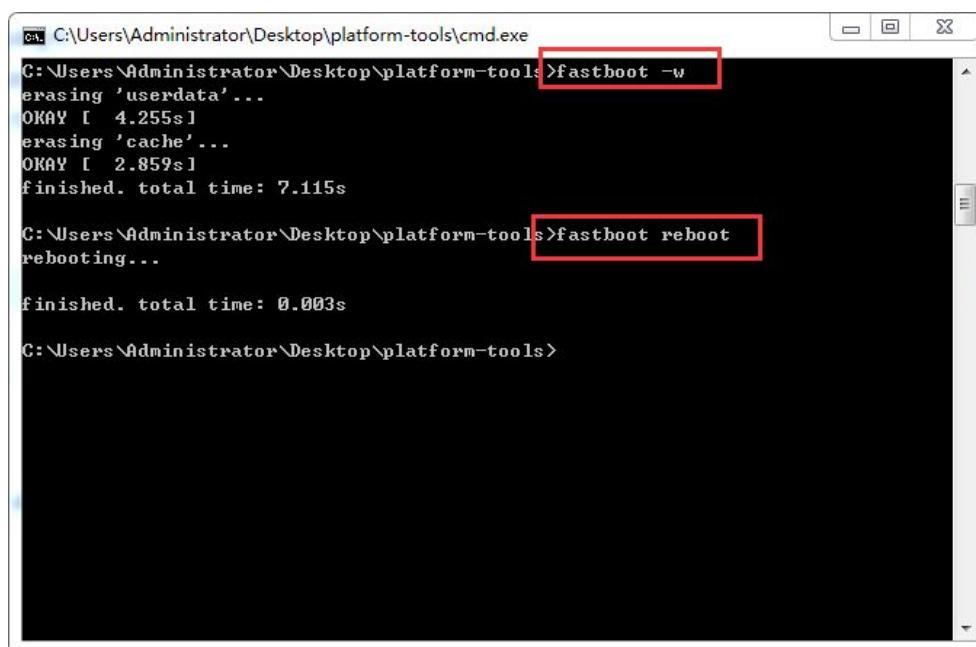


```
C:\Users\Administrator\Desktop\platform-tools>fastboot.exe flash kernel zImage
sending 'kernel' <3873 KB>...
OKAY [ 0.584s]
writing 'kernel'...
OKAY [ 1.194s]
finished. total time: 1.779s

C:\Users\Administrator\Desktop\platform-tools>fastboot.exe flash ramdisk ramdisk-uboot.img
sending 'ramdisk' <636 KB>...
OKAY [ 0.104s]
writing 'ramdisk'...
OKAY [ 0.174s]
finished. total time: 0.280s

C:\Users\Administrator\Desktop\platform-tools>fastboot.exe flash system system.img
sending 'system' <15937 KB>...
OKAY [ 2.375s]
writing 'system'...
OKAY [ 2.511s]
finished. total time: 4.887s
```

烧写完成之后，输入擦除命令 “fastboot -w” 和重启命令 “fastboot reboot” ，如下图所示。



```
C:\Users\Administrator\Desktop\platform-tools>fastboot -w
erasing 'userdata'...
OKAY [ 4.255s]
erasing 'cache'...
OKAY [ 2.859s]
finished. total time: 7.115s

C:\Users\Administrator\Desktop\platform-tools>fastboot reboot
rebooting...

finished. total time: 0.003s

C:\Users\Administrator\Desktop\platform-tools>
```

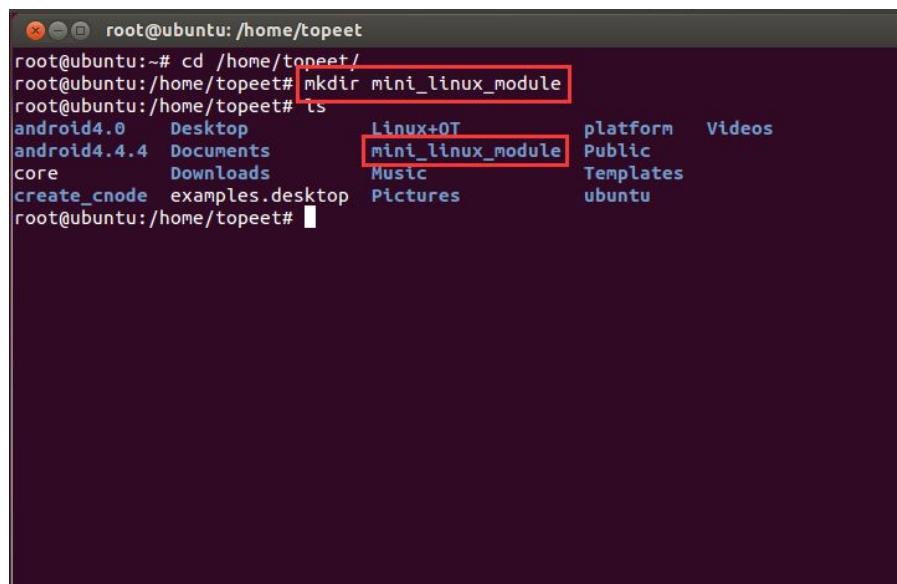
重启之后，超级终端会运行最小系统，如下图所示。

The screenshot shows a window titled "COM1 - HyperTerminal". The terminal output displays a Linux boot log from the dm9620 driver. It includes messages about file existence, MAC addresses, and network interface registration. The log ends with "Done". Below the log, a red box highlights the text "Please press Enter to activate this console.". At the bottom of the terminal window, there are two empty command-line prompts: "[root@iTOP-4412]#" and "[root@iTOP-4412]#".

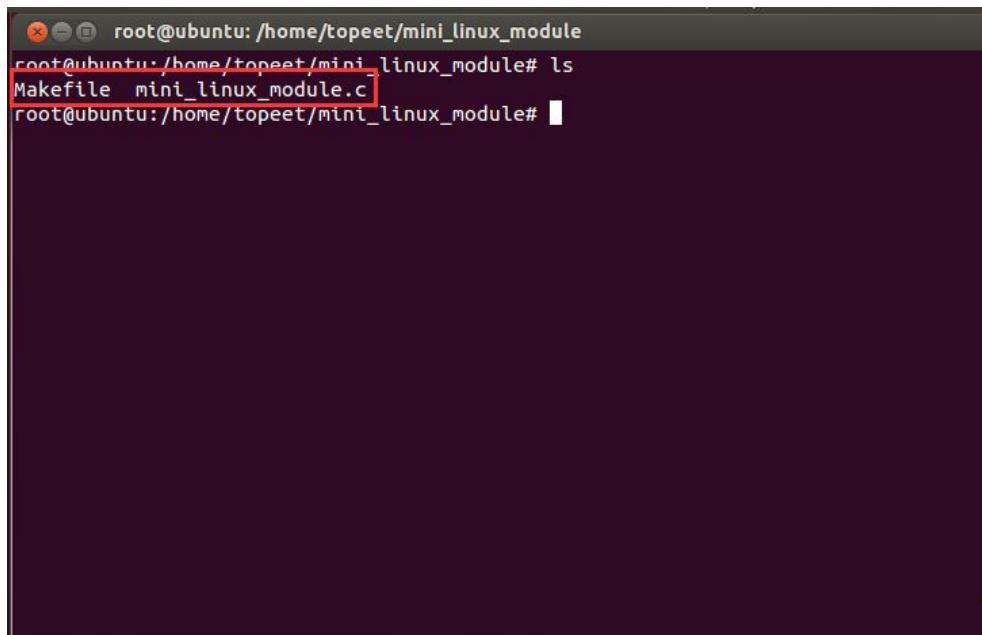
到这一步，烧写文件系统的准备工作就完成了。

#### 2.9.4 编译驱动模块

如下图所示，在“/home/topeet”目录下新建文件夹“mini\_linux\_module”。

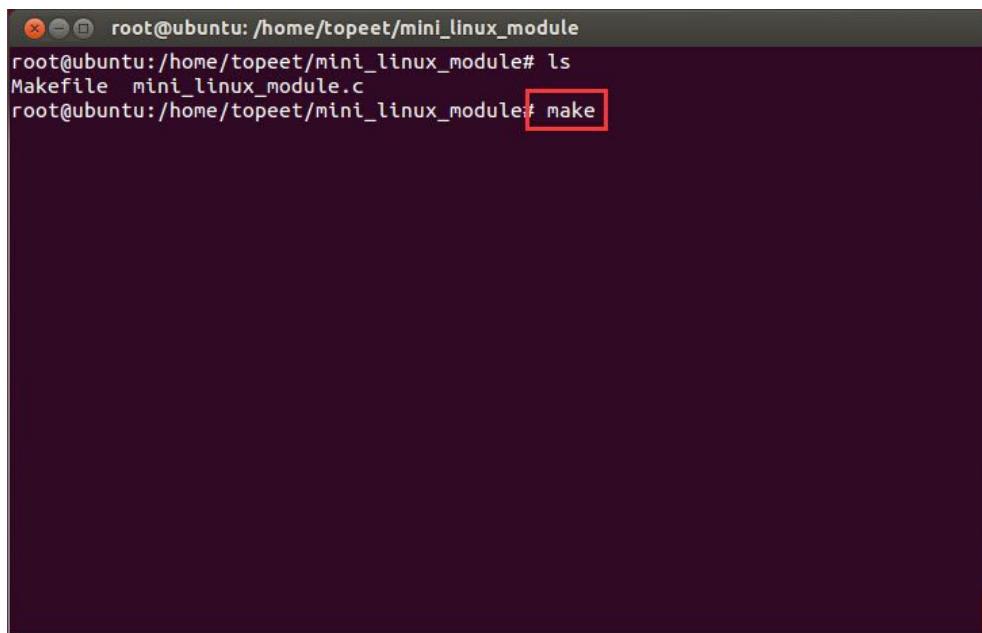


将文件拷贝前面编写好的“mini\_linux\_module.c”文件和对应的“Makefile”拷贝到该文件夹中，如下图所示。



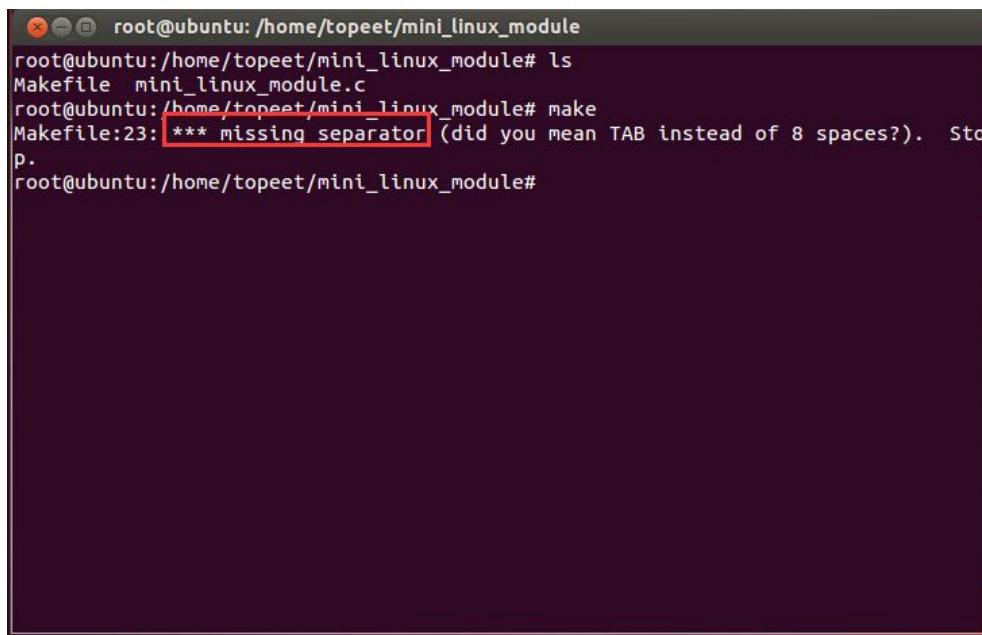
```
root@ubuntu:/home/topeet/mini_linux_module
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile mini_linux_module.c
root@ubuntu:/home/topeet/mini_linux_module#
```

如下图所示，在“/home/topeet/mini\_linux\_module”目录下执行命令“make”。



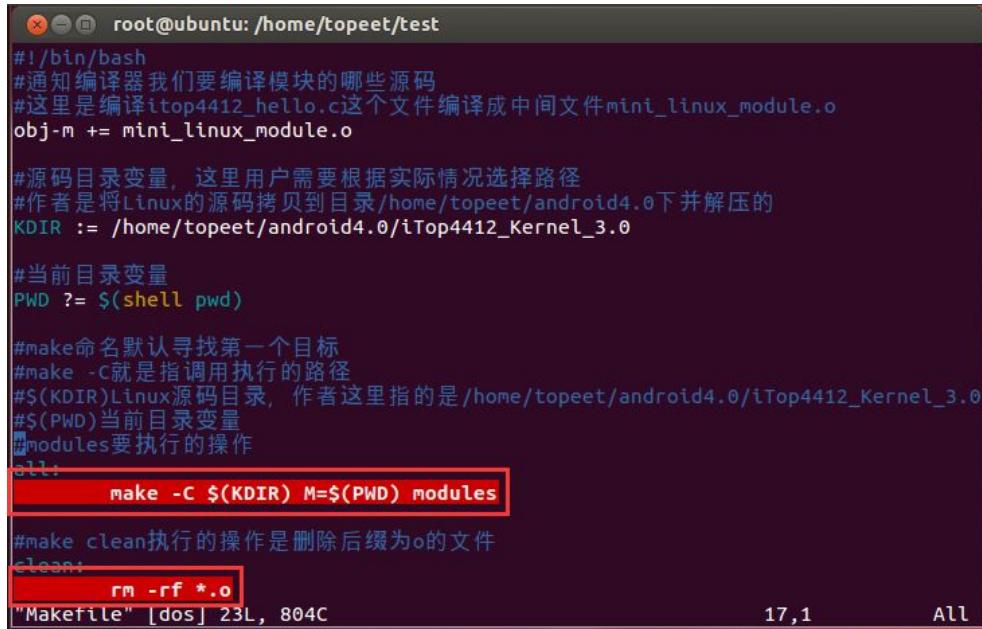
```
root@ubuntu:/home/topeet/mini_linux_module
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile mini_linux_module.c
root@ubuntu:/home/topeet/mini_linux_module# make
```

如下图所示，有可能出现错误“\*\*\* missing separator”。



```
root@ubuntu:/home/topeet/mini_linux_module
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile  mini_linux_module.c
root@ubuntu:/home/topeet/mini_linux_module# make
Makefile:23: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.
root@ubuntu:/home/topeet/mini_linux_module#
```

如果编译不通过，打开文件 Makefile，如下图所示。



```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += mini_linux_module.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${KDIR}Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${PWD}当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
"Makefile" [dos] 23L, 804C
```

如下图所示，在 Ubuntu 下使用 Vim 编辑器将红色部分处理一下，删除红色部分代码的空格部分，再输入 “Tab” 。

```
root@ubuntu:/home/topeet/mini_linux_module
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += mini_linux_module.o

#源码目录变量, 这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR)Linux源码目录, 作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${PWD}当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

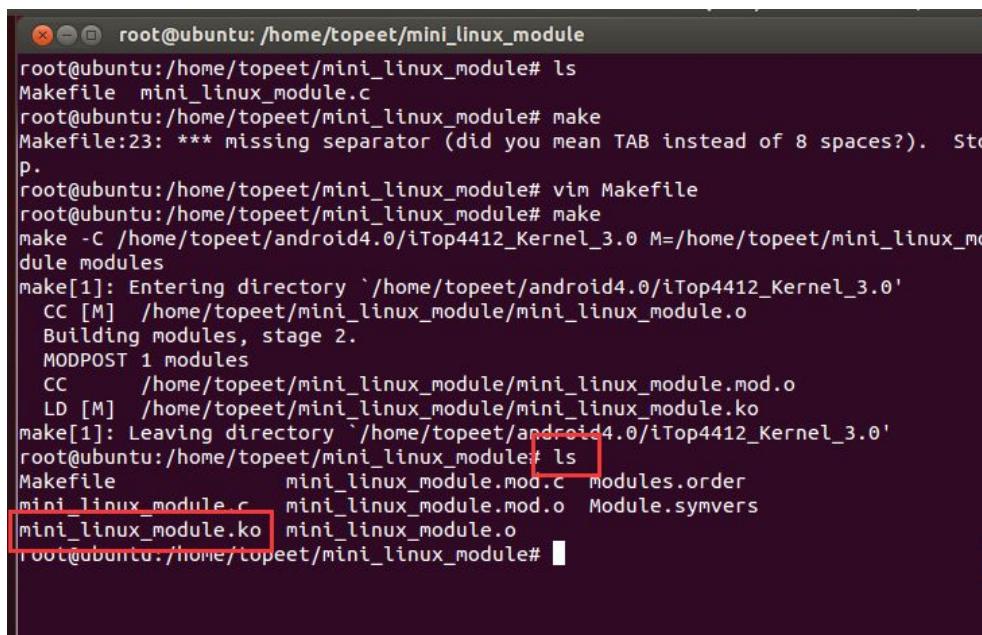
#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
-- INSERT --
```

保存退出，然后再执行编译命令“make”，如下图所示，编译成功。

```
root@ubuntu:/home/topeet/mini_linux_module
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile  mini_linux_module.c
root@ubuntu:/home/topeet/mini_linux_module# make
Makefile:23: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.
root@ubuntu:/home/topeet/mini_linux_module# vim Makefile
root@ubuntu:/home/topeet/mini_linux_module# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/mini_linux_mo
dule modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M]  /home/topeet/mini_linux_module/mini_linux_module.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/mini_linux_module/mini_linux_module.mod.o
  LD [M]  /home/topeet/mini_linux_module/mini_linux_module.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/mini_linux_module#
```

如下图所示，使用查看命令“ls”，可以看到生成了驱动模块文件“mini\_linux\_module.ko”，

这个文件就是最后要加载到开发板中的文件。



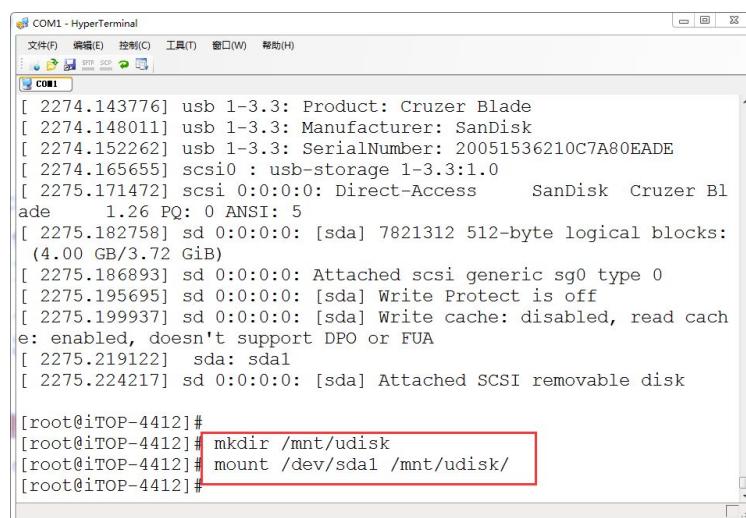
```
root@ubuntu:/home/topeet/mini_linux_module
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile  mini_linux_module.c
root@ubuntu:/home/topeet/mini_linux_module# make
Makefile:23: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.
root@ubuntu:/home/topeet/mini_linux_module# vim Makefile
root@ubuntu:/home/topeet/mini_linux_module# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/mini_linux_mo
dule modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
CC [M]  /home/topeet/mini_linux_module/mini_linux_module.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/topeet/mini_linux_module/mini_linux_module.mod.o
LD [M]  /home/topeet/mini_linux_module/mini_linux_module.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/mini_linux_module# ls
Makefile          mini_linux_module.mod.c  modules.order
mini_linux_module.c  mini_linux_module.mod.o  Module.symvers
mini_linux_module.ko  mini_linux_module.o
root@ubuntu:/home/topeet/mini_linux_module#
```

## 2.9.5 加载驱动

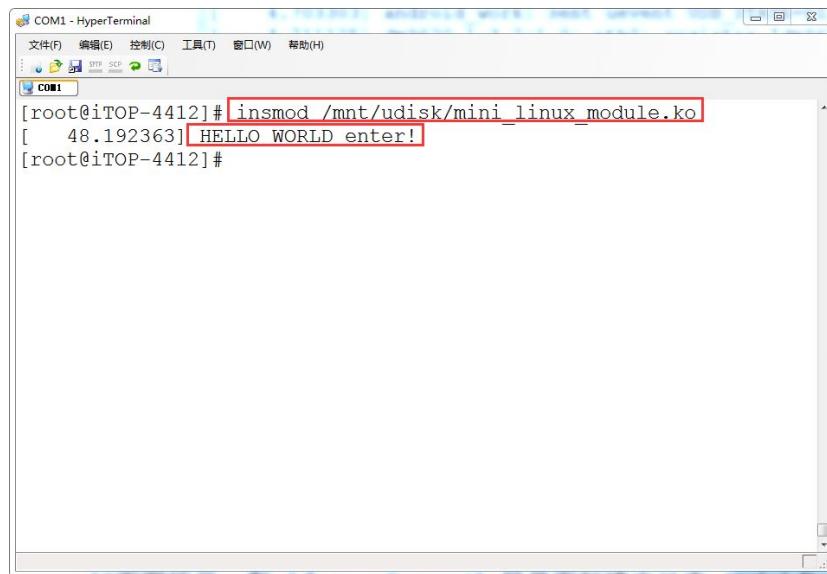
将 mini\_linux\_module.ko 驱动模块文件拷贝到 U 盘或者 TF，然后接到开发板上。

将 U 盘或者 TF 卡加载，可以参考使用手册 11.3.3 qt 挂载盘符。

作者这里使用的是 U 盘，挂载之后需要的加载模块驱动 KO 文件，在 mnt/udisk 下。如下图所示。



然后使用命令 “`insmod /mnt/udisk/mini_linux_module.ko`” 加载模块，如下图所示。



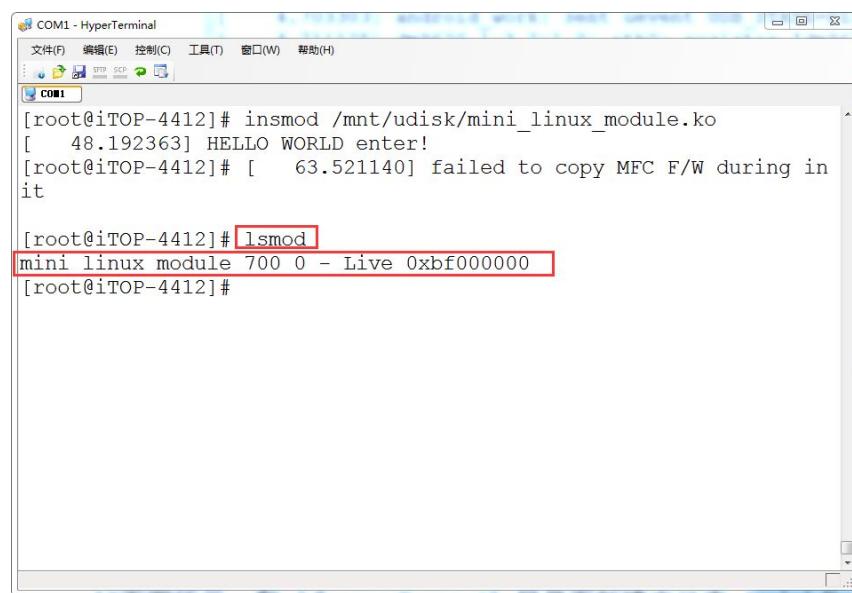
The screenshot shows a HyperTerminal window titled "COM1 - HyperTerminal". The terminal window has a menu bar with "文件(F)", "编辑(E)", "控制(C)", "工具(T)", "窗口(W)", and "帮助(H)". Below the menu is a toolbar with icons for file operations. The main pane displays a command-line session:

```
[root@iTOP-4412]# insmod /mnt/udisk/mini_linux_module.ko
[ 48.192363] HELLO WORLD enter!
[root@iTOP-4412]#
```

如上图所示，可以看到加载过程中打印了信息 “Hello World enter!”，说明驱动已经加载到 Linux 中去了。

## 2.9.6 卸载驱动

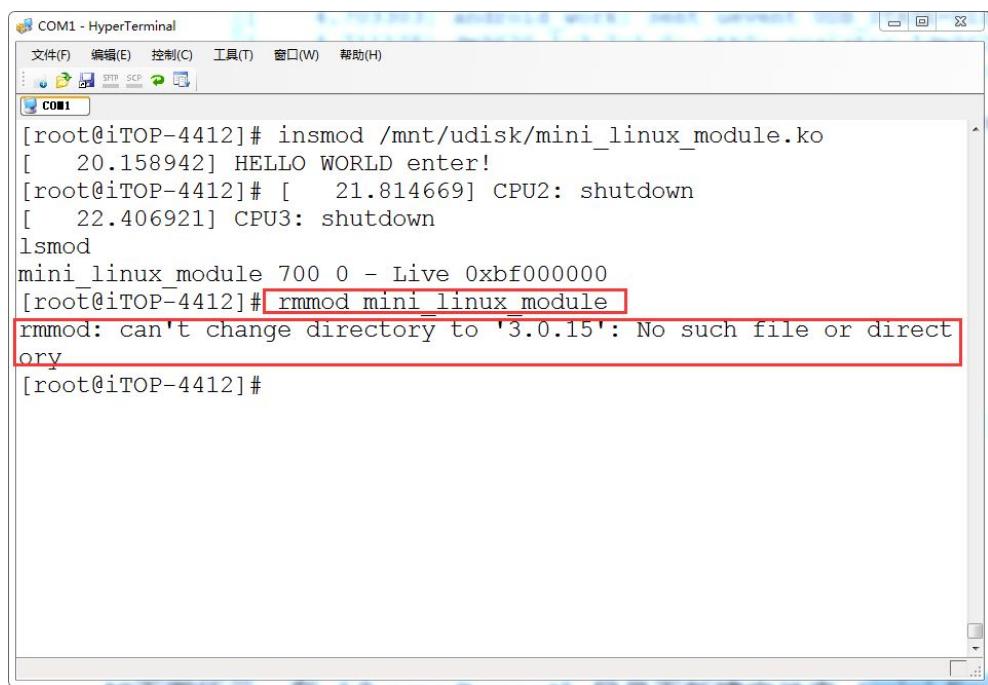
卸载驱动前面，先使用命令 “`lsmod`” ，查看一下模块信息，如下图所示。



The screenshot shows a HyperTerminal window titled "COM1 - HyperTerminal". The terminal window has a menu bar with "文件(F)", "编辑(E)", "控制(C)", "工具(T)", "窗口(W)", and "帮助(H)". Below the menu is a toolbar with icons for file operations. The main pane displays a command-line session:

```
[root@iTOP-4412]# insmod /mnt/udisk/mini_linux_module.ko
[ 48.192363] HELLO WORLD enter!
[root@iTOP-4412]# [ 63.521140] failed to copy MFC F/W during init
[root@iTOP-4412]# lsmod
mini linux module 700 0 - Live 0xbff000000
[root@iTOP-4412]#
```

如下图所示，使用命令“rmmod mini\_linux\_module”卸载驱动模块。

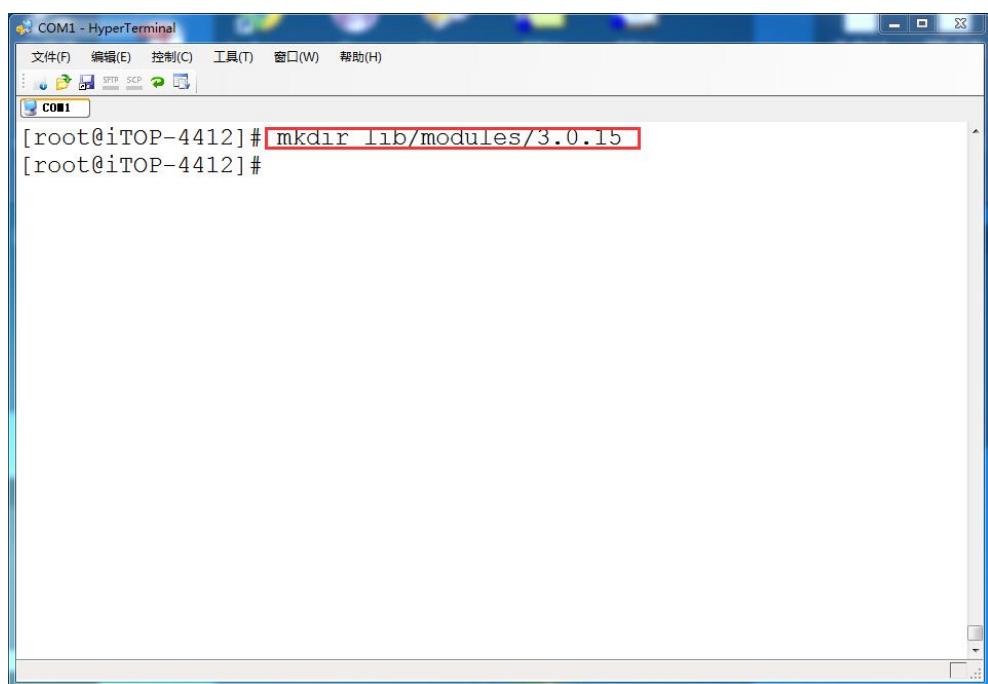


The screenshot shows a HyperTerminal window titled "COM1 - HyperTerminal". The terminal session output is as follows:

```
[root@iTOP-4412]# insmod /mnt/udisk/mini_linux_module.ko
[ 20.158942] HELLO WORLD enter!
[root@iTOP-4412]# [ 21.814669] CPU2: shutdown
[ 22.406921] CPU3: shutdown
lsmod
mini_linux_module 700 0 - Live 0xbff000000
[root@iTOP-4412]# rmmod mini_linux_module
rmmod: can't change directory to '3.0.15': No such file or directory
[root@iTOP-4412]#
```

The command "rmmod mini\_linux\_module" is highlighted with a red box, and the resulting error message "rmmod: can't change directory to '3.0.15': No such file or directory" is also highlighted with a red box.

如上图所示，会报错无法卸载，提示没有文件夹。这里根据提示，使用命令“mkdir lib/modules/3.0.15”新建目录“lib/modules/3.0.15”，如下图所示。



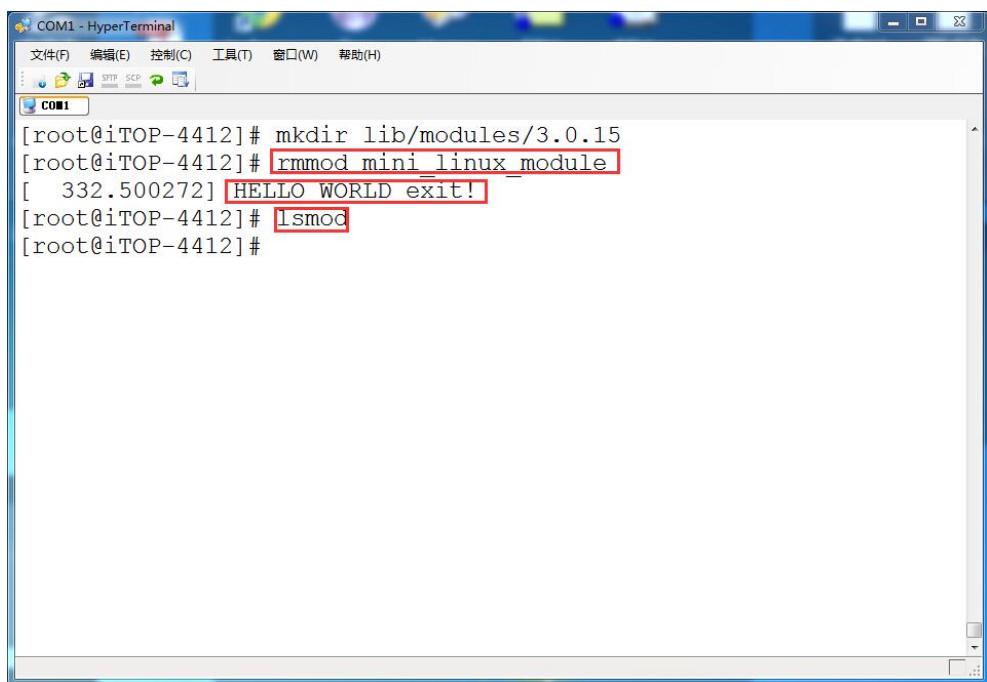
The screenshot shows a HyperTerminal window titled "COM1 - HyperTerminal". The terminal session output is as follows:

```
[root@iTOP-4412]# mkdir lib/modules/3.0.15
[root@iTOP-4412]#
```

The command "mkdir lib/modules/3.0.15" is highlighted with a red box.

新建文件夹之后，再使用卸载驱动模块的命令“`rmmmod mini_linux_module`”，如下图所示，可以看到打印出了在卸载驱动函数里面添加的打印信息：`Hello world exit !`

最后使用命令：`lsmod`，对比前面的 `lsmod`，发现已经没有了加载的模块驱动了。



# 实验 03 Menuconfig\_Kconfig

## 3.1 本章导读

Linux 驱动工程师一定要掌握 Linux 内核的编译方法，也就是将 Linux 内核源码，编译成针对特定硬件的二进制镜像。

在前面入门视频“01-烧写、编译以及基础知识视频”→“实验 10-搭建编译环境 uboot\_linux\_Android”中，简单的介绍过如何将 Linux 源码编译生成二进制 zImage。

在本章中，将更加详细的介绍这部分内容，然后介绍 Kconfig 配置文件，Kconfig 文件是和编译的 Makemenuconfig 工具配合使用的。最后还需要掌握 “.config” 文件的作用。

### 3.1.1 工具

#### 3.1.1.1 硬件工具

1 ) PC 机

#### 3.1.1.2 软件工具

1 ) 虚拟机 Vmware

2 ) Ubuntu12.04.2

3 ) Ubuntu 系统下解压生成的 Linux 源码

### 3.1.2 预备课程

入门视频“01-烧写、编译以及基础知识视频”→“实验 10-搭建编译环境 uboot\_linux\_Android”或者使用手册“五 Android 开发环境搭建以及编译”

### 3.1.3 视频资源

本节配套视频为“视频 03\_Menuconfig\_Kconfig”

## 3.2 学习目标

本章需要学习以下内容：

掌握 Menuconfig 的用法

理解 Kconfig 文件并掌握修改 Kconfig 的方法

理解配置文件“.config”

Linux 内核配置裁减

## 3.3 Linux 内核配置系统

Linux 内核配置系统由三个部分组成。

Makefile 文件：分布在 Linux 内核源码中的 Makefile 文件，定义了 Linux 内核的编译规则。

Kconfig 文件：给用户提供配置选择的功能。

配置工具：这里使用的是 menuconfig，相比其它工具，这个工具使用的比较多，也比较容易上手，无论哪个机构发布的 Linux 版本应该都是支持 menuconfig 的。

## 3.4 Menuconfig 的操作

Linux 的裁减配置是通过 menuconfig 工具来实现的，本节介绍如何使用这个工具。

### 3.4.1 Menuconfig 发展历史简介

在 Linux 发展过程中，配置内核可以使用以下工具。

#make config

这是基于文本的最为传统的配置界面，不推荐使用

#make menuconfig

基于文本菜单的配置界面，现在大部分都是使用这个工具来裁减配置内核的，本章节也是介绍这种方法。

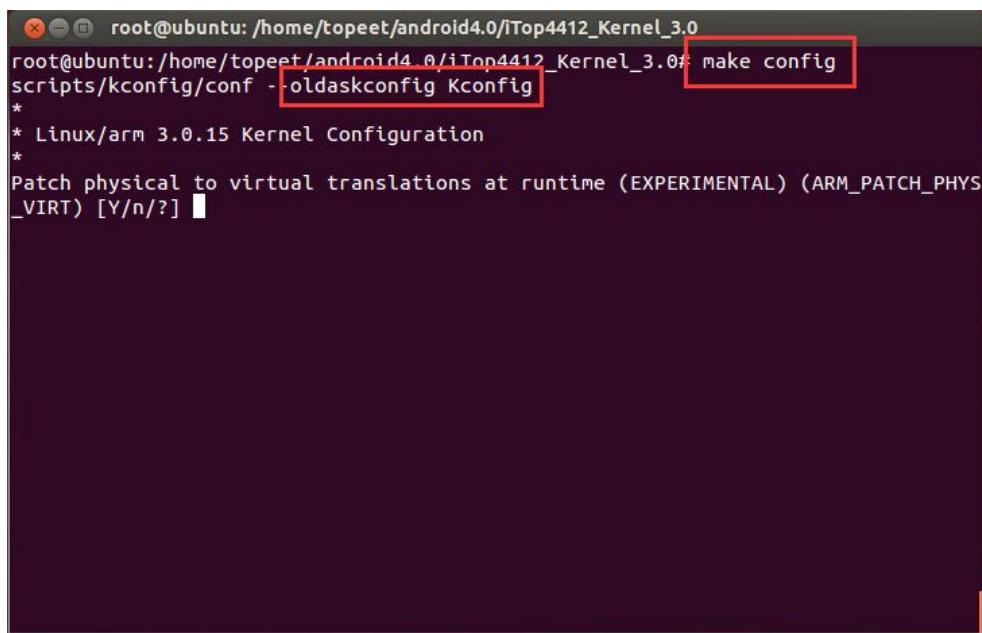
#make xconfig

要求 QT 被安装，用的比较少。

#make gconfig

要求 GTK，用的比较少。

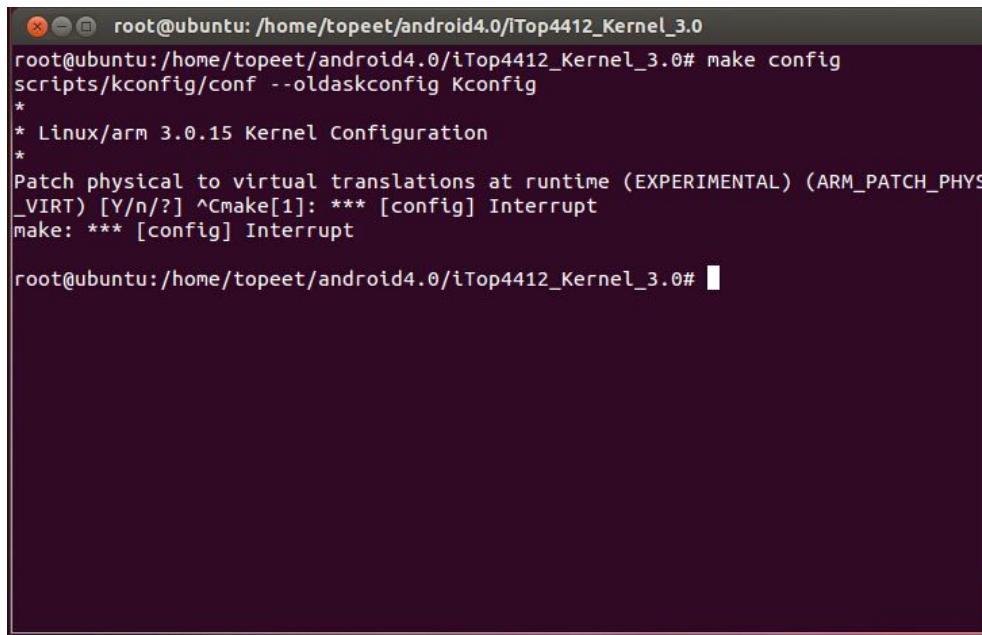
上面不同的命令代表使用不同的工具，如下图所示，在源码目录下，输入命令 “make config” 。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/arm 3.0.15 Kernel Configuration
*
Patch physical to virtual translations at runtime (EXPERIMENTAL) (ARM_PATCH_PHYS_VIRT) [Y/n/?] ■
```

如上图所示，这是一个文本类型的配置工具，根据提示“scripts/kconfig/conf --oldaskconfig Kconfig”可以知道，这种方法是旧的配置方法，虽然 Linux 内核可能会长时间的支持，但是不人性化的操作方式，会降低效率，所以现在几乎淘汰了。

使用“Ctrl+c”可以退出配置界面，退出后，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/arm 3.0.15 Kernel Configuration
*
Patch physical to virtual translations at runtime (EXPERIMENTAL) (ARM_PATCH_PHYS_VIRT) [Y/n/?] ^Cmake[1]: *** [config] Interrupt
make: *** [config] Interrupt

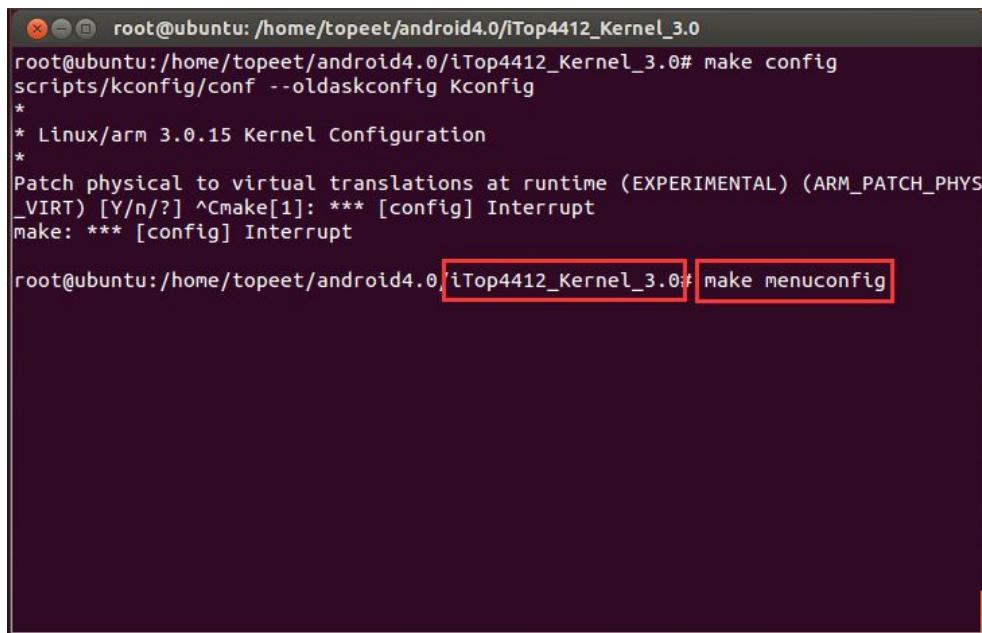
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ■
```

### 3.4.2 Menuconfig 操作方法

前面实验提到过 menuconfig 实现的代码在源码 “scripts” 目录下，不过这里根本不用关心它是怎么实现的，只需要掌握怎么操作即可，就像学习开车，要知道怎么打方向盘，而不需要知道方向盘和轮子之间是怎么传动的。

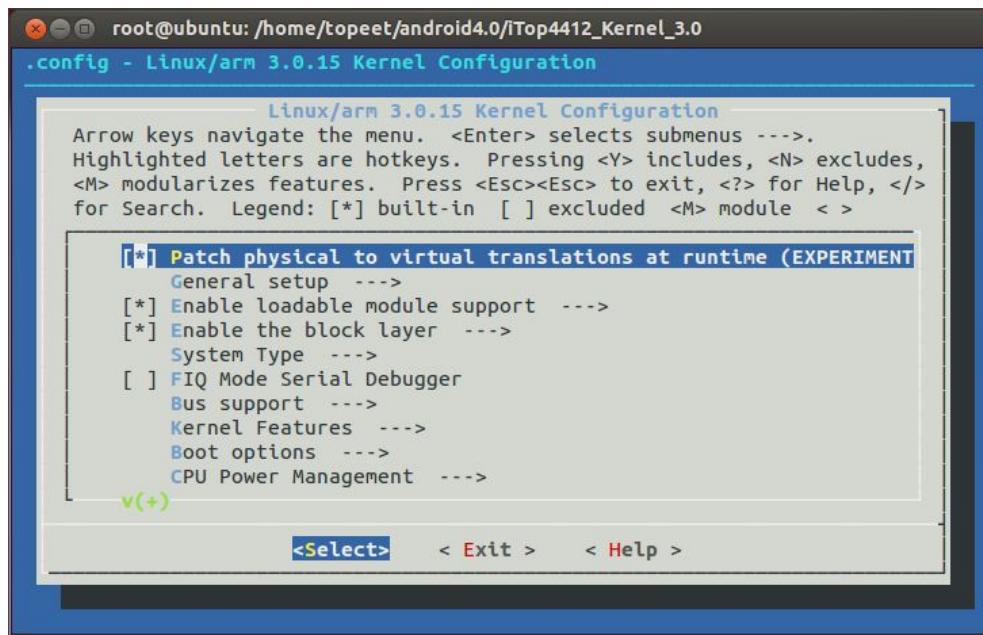
下面介绍 menuconfig 的操作方法。

如下图所示，在源码目录下，输入命令 “make Menuconfig” 。

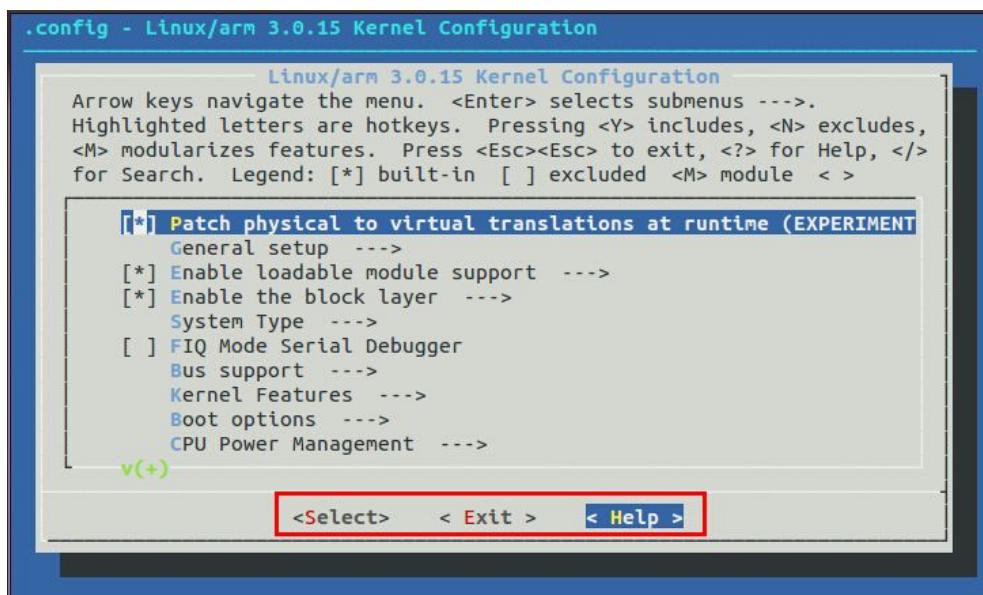


```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/arm 3.0.15 Kernel Configuration
*
Patch physical to virtual translations at runtime (EXPERIMENTAL) (ARM_PATCH_PHYS
_VIRT) [Y/n/?] ^Cmake[1]: *** [config] Interrupt
make: *** [config] Interrupt
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make menuconfig
```

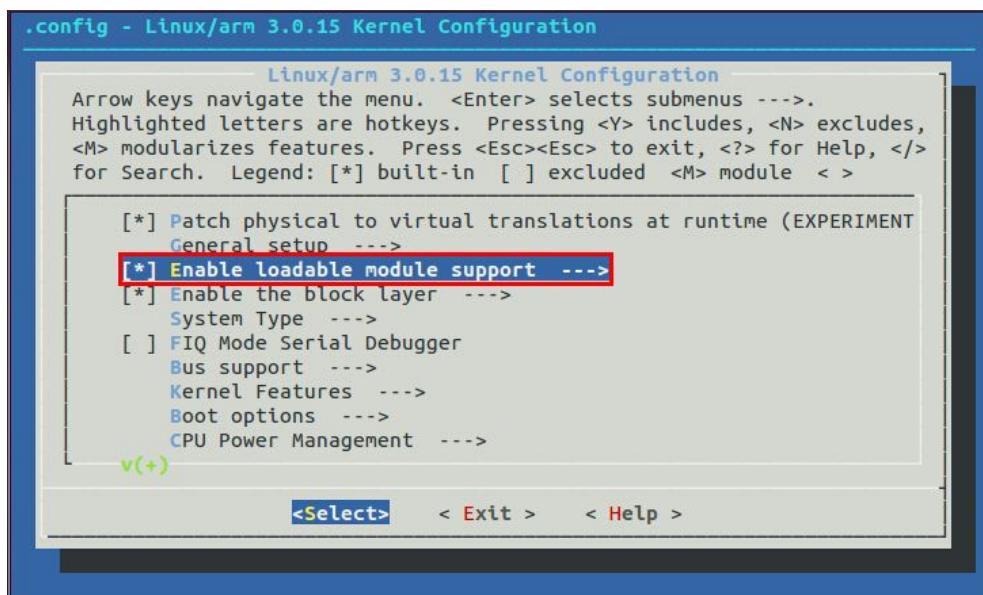
如下图所示，进入配置界面。这个功能界面对应配置工具，它包含配置命令解释器，对配置脚本中使用的命令进行解释；还包含了配置用户界面，用来提供字符界面和图形界面。这些配置工具都是使用脚本语言编写的，不过只用关心怎么使用。



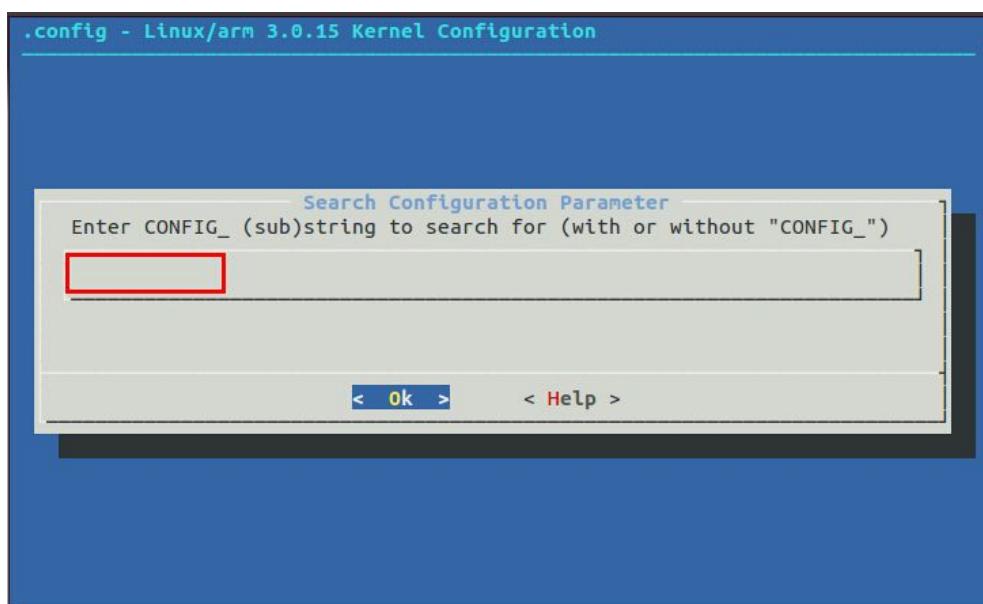
如下图所示，“方向按键”中的“左右”可以选择你需要的操作。“<Select>”表示进入选择的配置界面，“< Exit >”表示返回，“< Help >”可以阅读帮助文档。



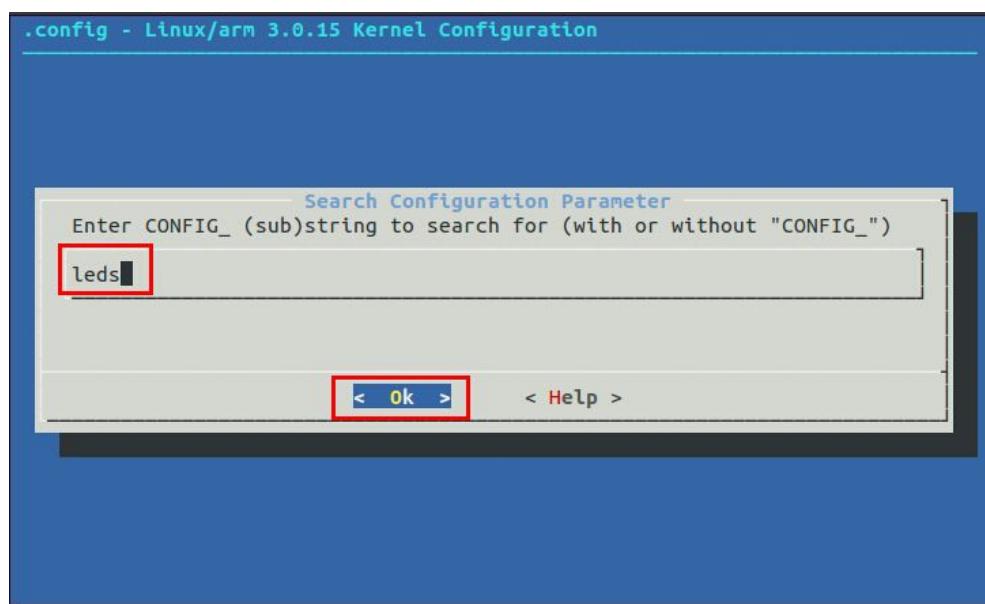
如下图所示，“方向按键”中的“上下”可以选择配置的选项。



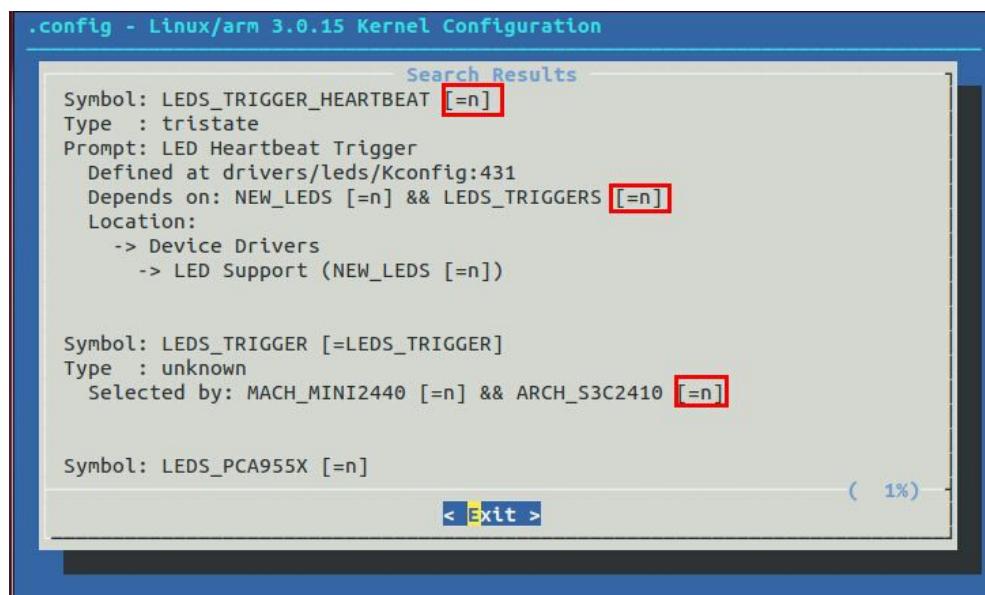
如下图所示，输入“/”，可以进入搜索界面。



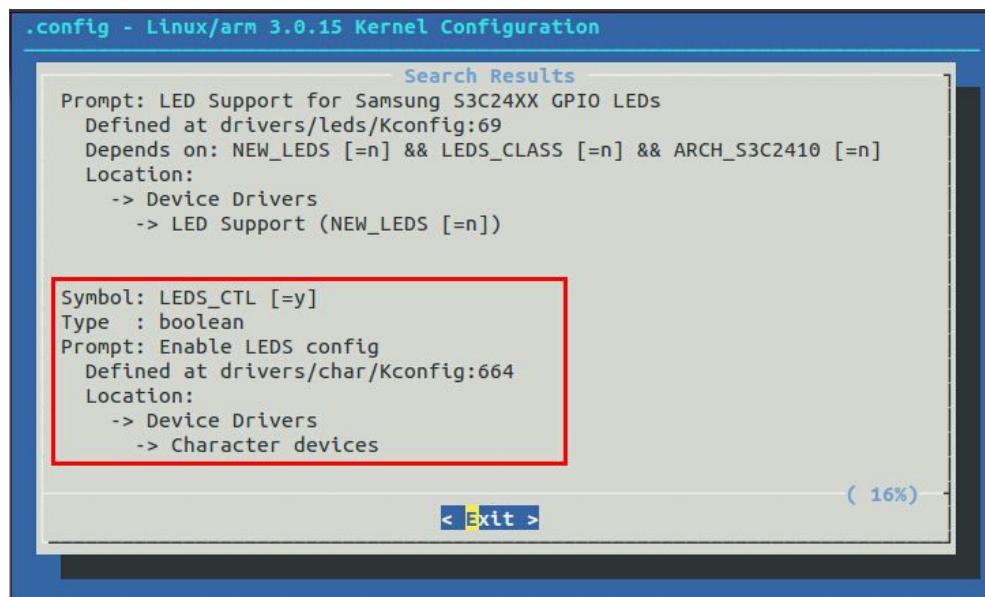
如下图所示，这里来查找一下“leds”的驱动，输入“leds”，然后按“回车”。



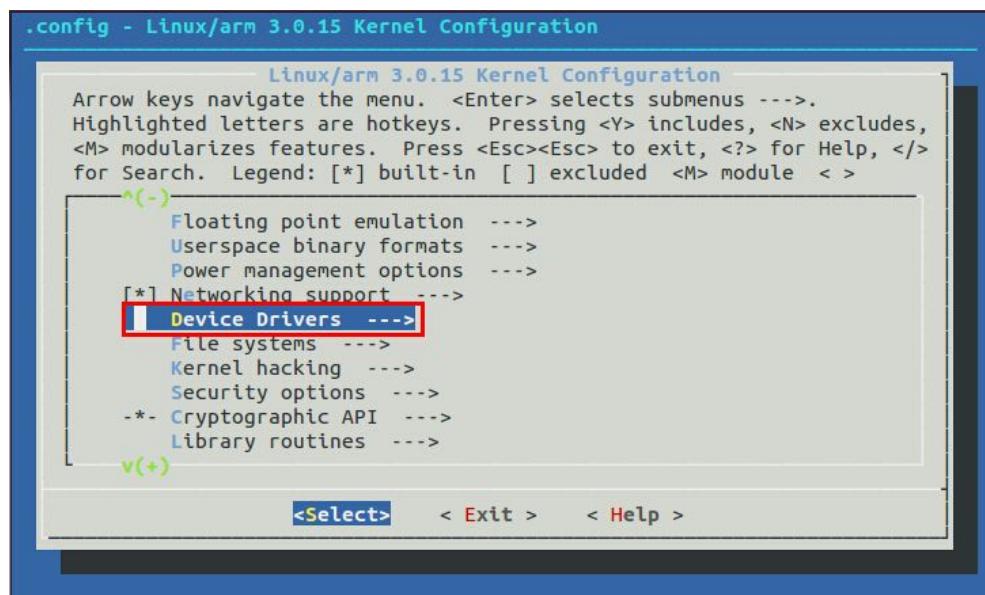
如下图所示，发现很多配置都是“=n” ，通过方向按键，控制向下翻页，然后观察那个选项配置成了“=y”。



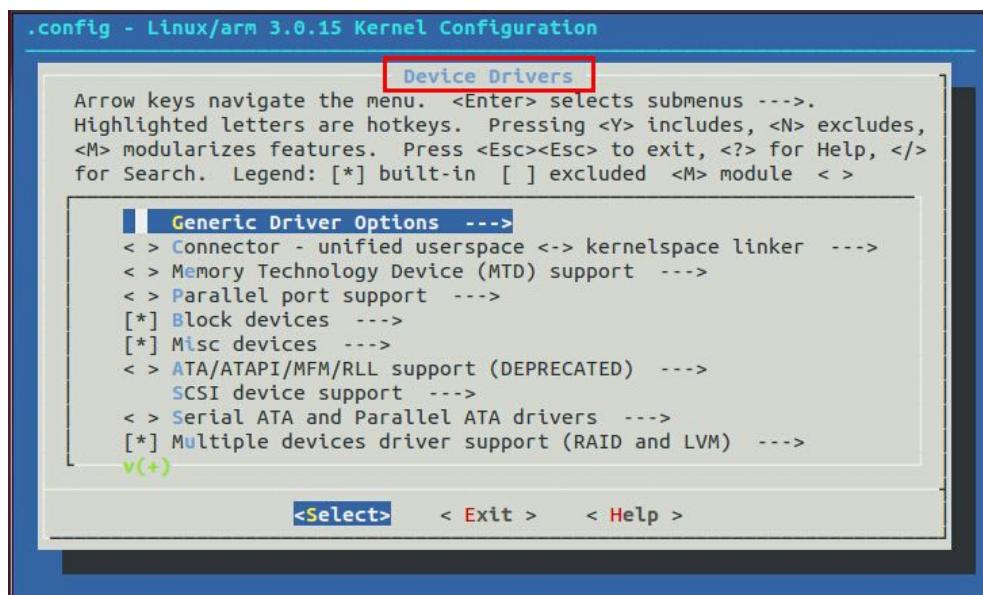
如下图所示 ,这里可以看到这个 leds 驱动的目录 “Device Drivers” “Character devices”。



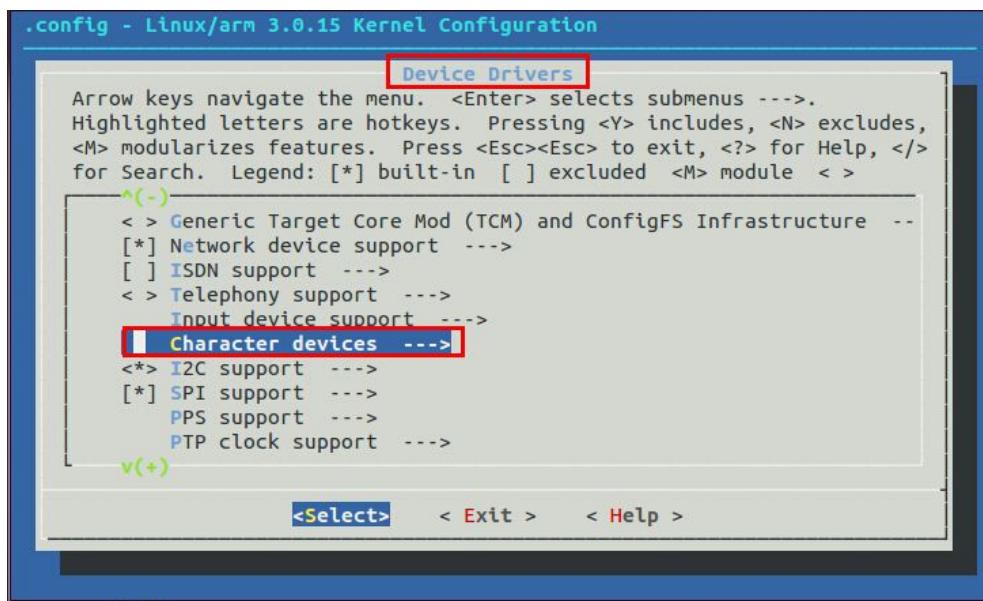
然后，根据查找出来的信息，找到对应的 leds 驱动。如下图，返回配置界面。找到“Device Drivers”目录。输入“回车”。



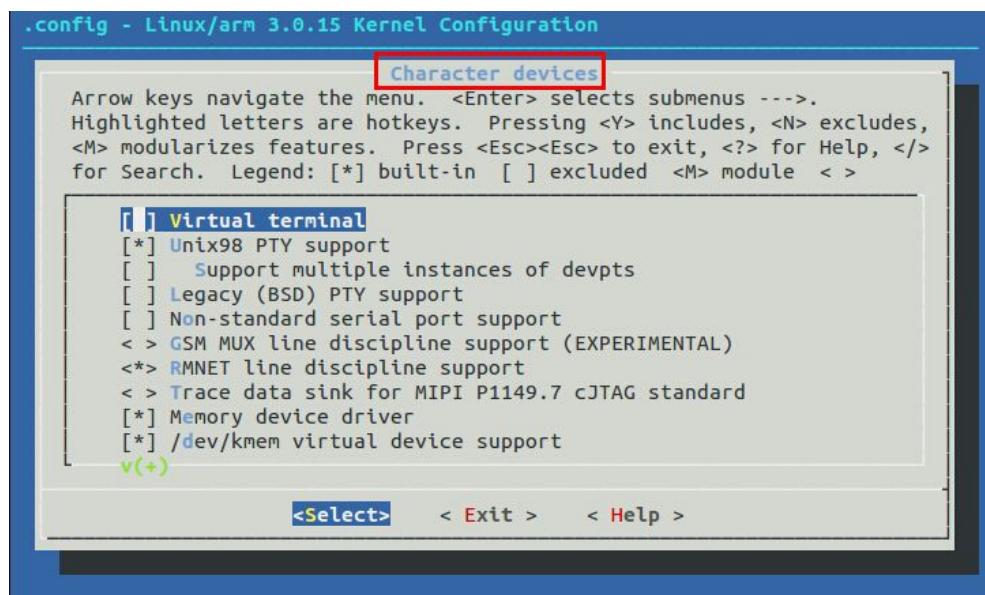
如下图所示，进入“Device Drivers”对应的配置界面。



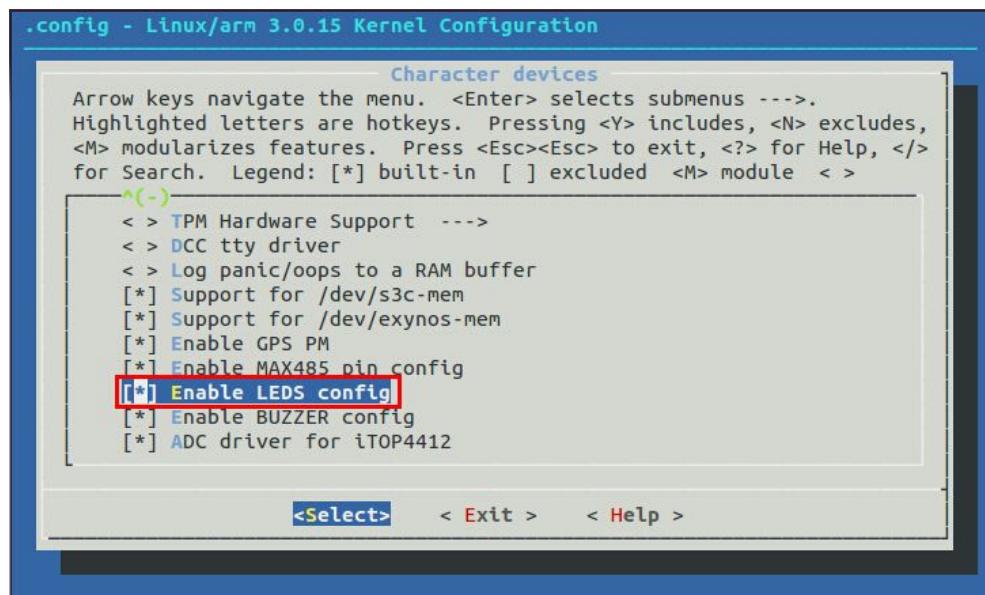
然后，如下图所示，找到“Character devices”，输入“回车”。



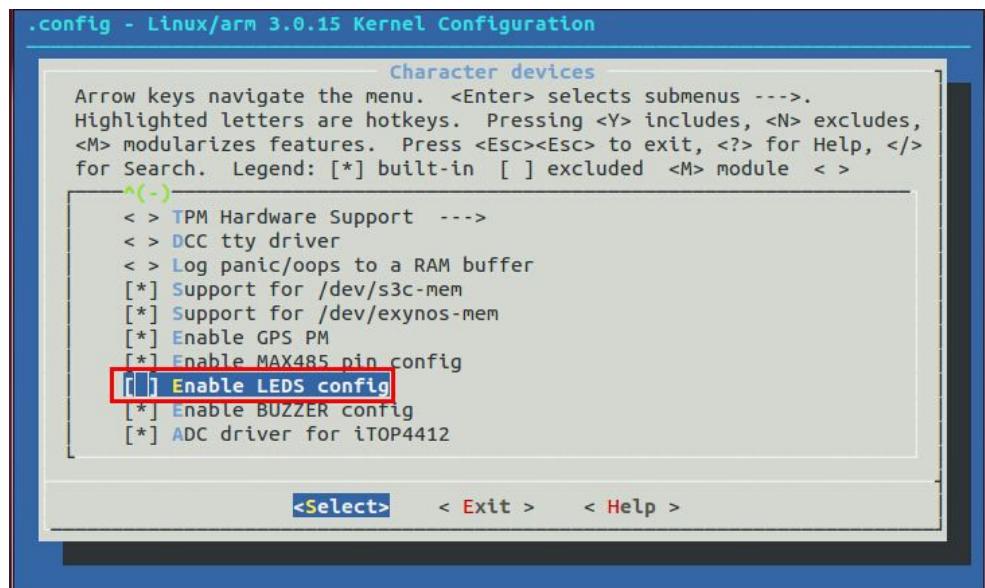
如下图所示，进入“Character devices”配置界面。



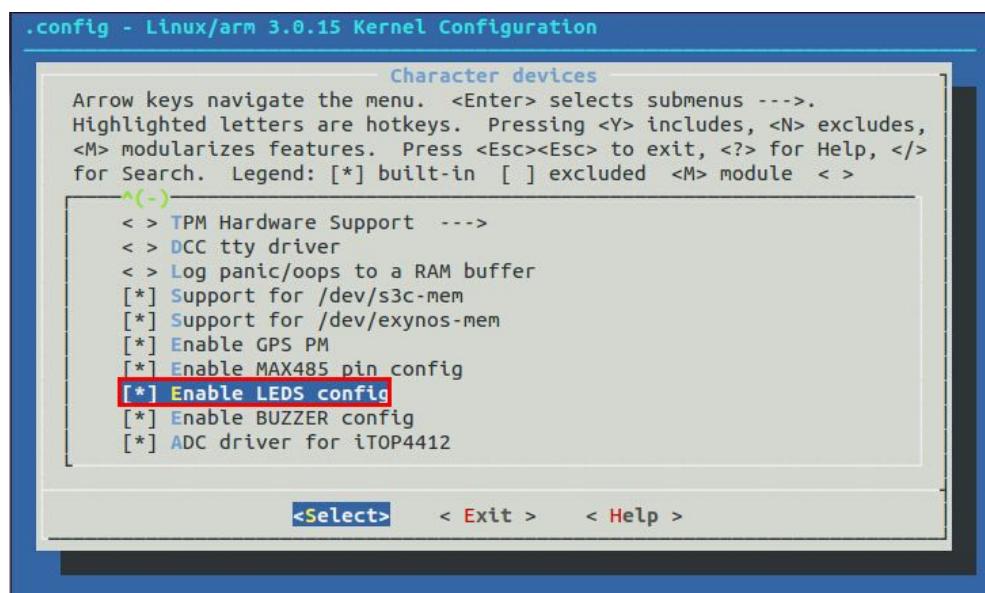
如下图所示，“Enable LEDS config” 找到对应的 leds 驱动配置选项。缺省配置文件里，这个已经选上了。



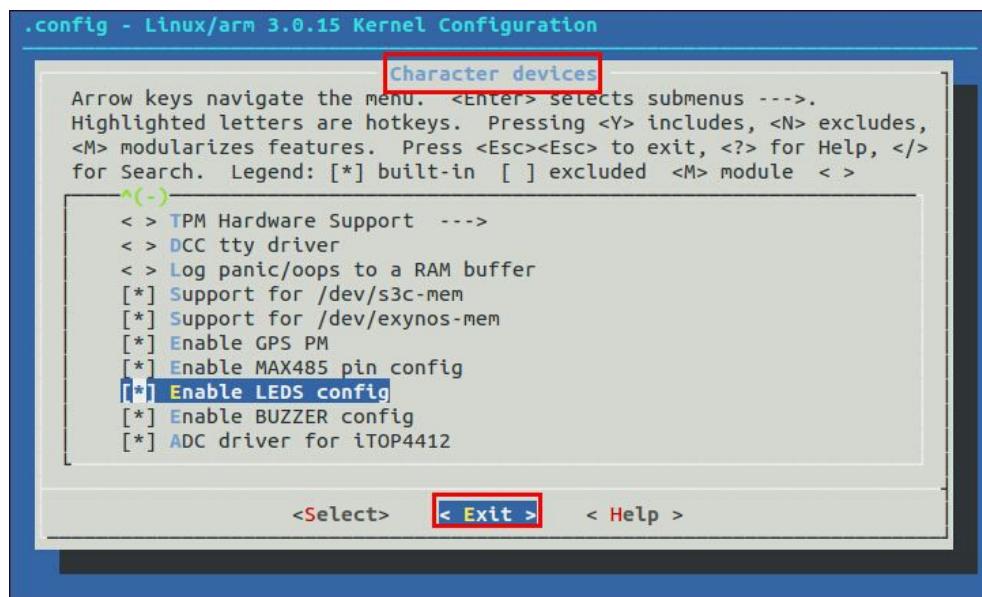
单击“空格”键后，去掉 leds 驱动选项。



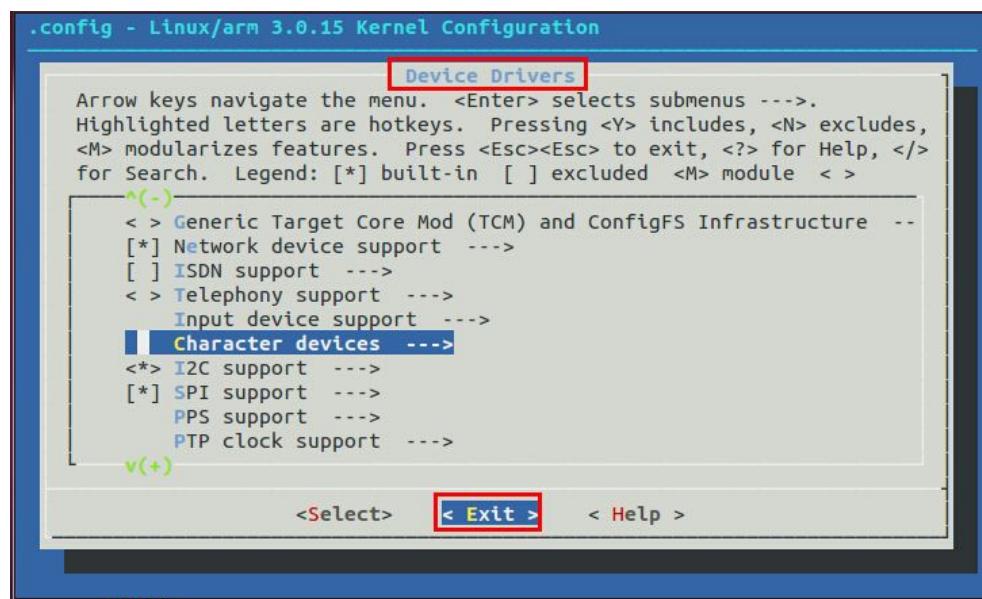
再次敲击“空格”，选上leds驱动的选项。



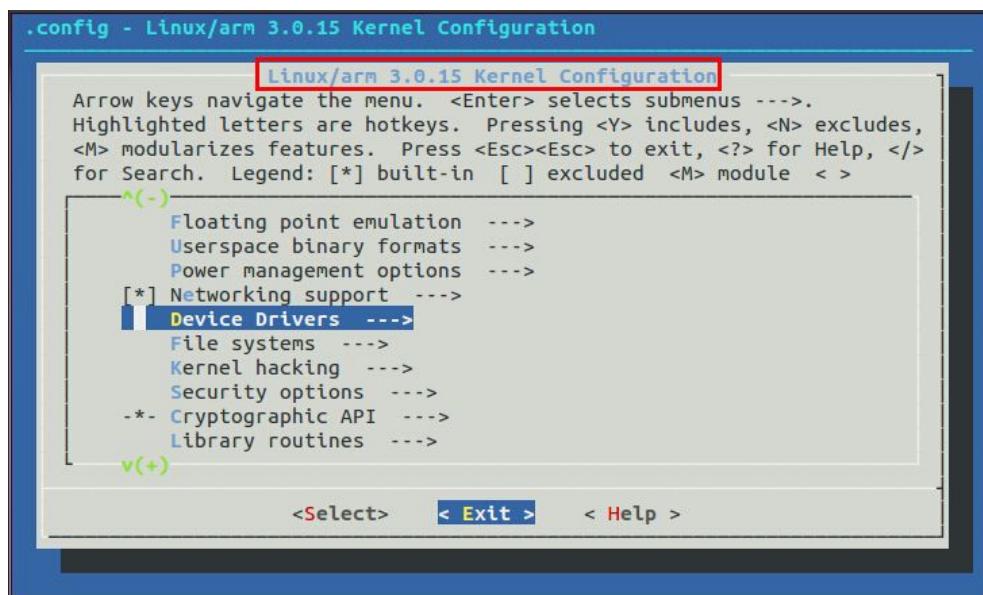
然后，选上“Exit”，如下图所示，输入“回车”。



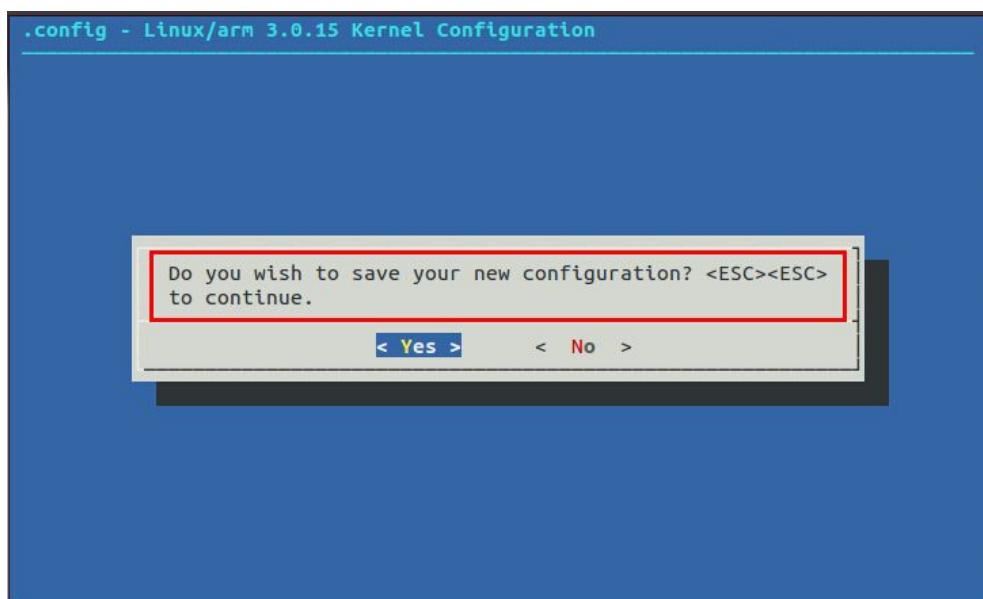
如下图所示，继续退出。



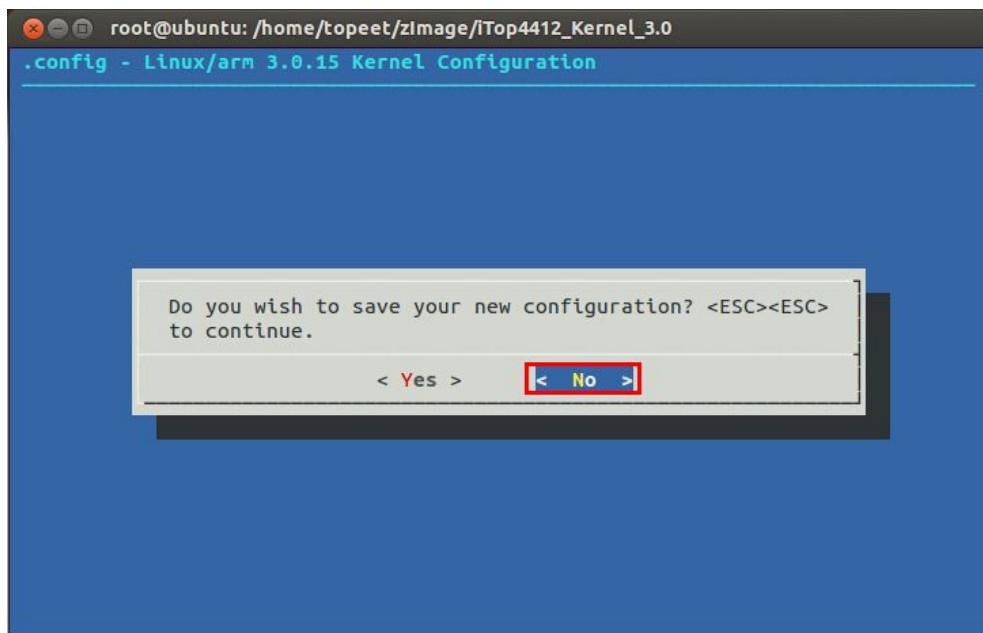
如下图所示，继续退出。



如下图所示，因为修改过配置选项，所以退出的时候会提醒“是否保存新的配置”。



如下图所示，因为第一次操作，担心用户在无意间动了某个配置选项，编译后无法启动，建议选择“No”，不保存退出。



到这里，整个 Menuconfig 配置的操作以及流程就完全介绍完了。如果修改了配置文件，如下图所示的 “.config” 文件就会被修改。再次编译内核的时候，系统会根据新的 config 文件来编译整个内核。

内核的配置非常多，大家可以看一下使用手册 9.4 小节，里面有详细的介绍。

### 3.5 .config 文件和 menuconfig 的关系

menuconfig 最终是为了生成一个.config 文件，这么说大家可能不是很理解，下面给大家举个例子，大家理解了这个例子，对.config 和 menuconfig 的关系就清楚了。

.config 文件在 linux 源码顶层目录，默认是隐藏的，使用查看命令 “ls -a” ，如下图所示。

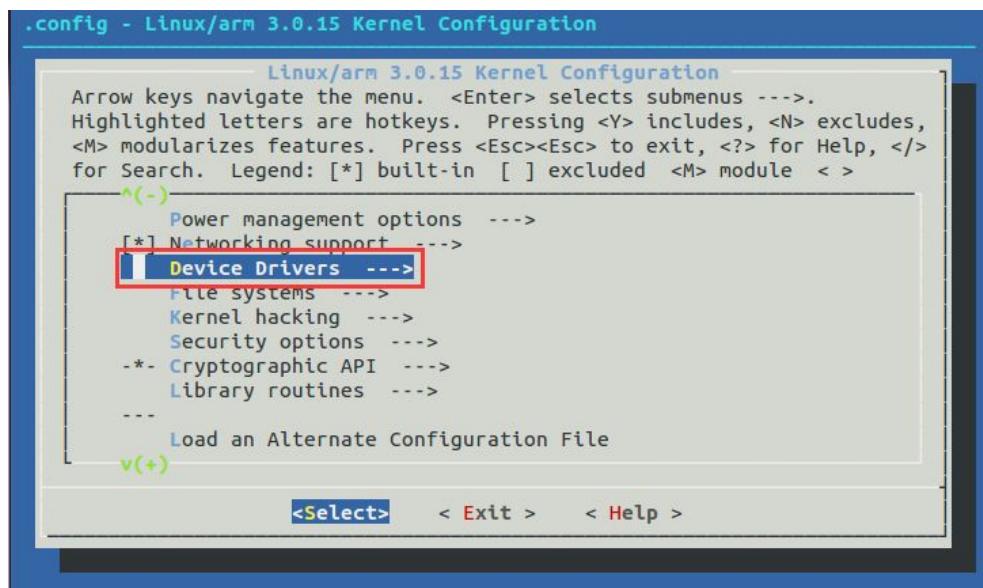
```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls -a
.
..
arch
binary
block
.config
config_for_android
config_for_android_2M
Kbuild
Kconfig
kernel
kernel_readme.txt
lib
MAINTAINERS
Makefile
missing-syscalls.d
tmp_kallsyms1.o
..tmp_kallsyms1.o.cmd
..tmp_kallsyms1.S
..tmp_kallsyms2.o
..tmp_kallsyms2.o.cmd
..tmp_kallsyms2.S
..tmp_System.map
tmp_versions
```

以 leds 小灯的驱动配置为例，介绍 menuconfig 具体执行过程。

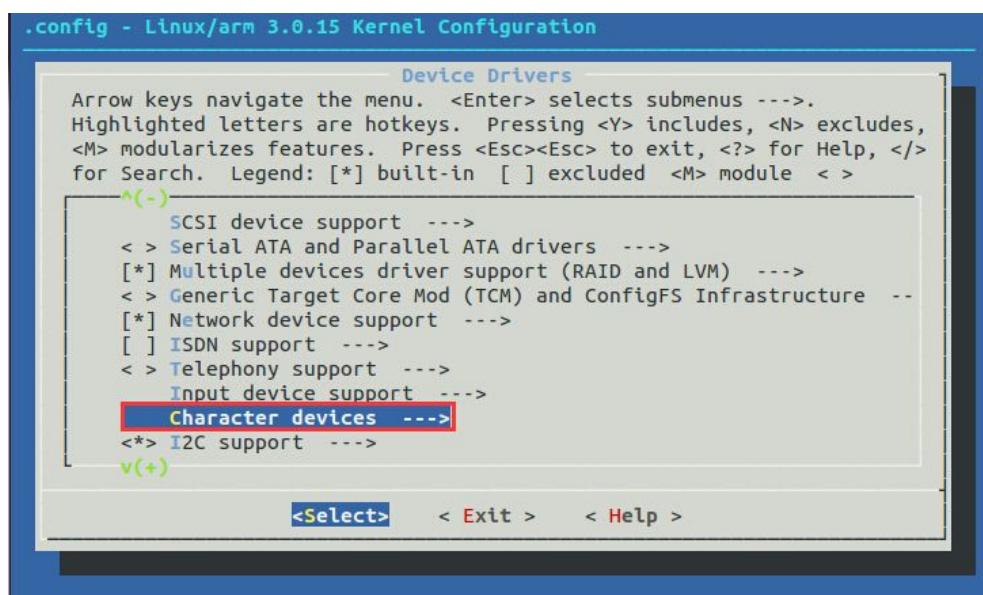
使用命令 “vim .config” ，打开文件 “.config” 配置文件，在配置间中搜索“CONFIG\_LEDS\_CTL” ,这里注意要大写，配置文件里面都是一些宏定义，宏定义一般使用英文大写。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
CONFIG_HW_RANDOM=y
# CONFIG_HW_RANDOM_TIMERIOMEM is not set
# CONFIG_R3964 is not set
# CONFIG_RAW_DRIVER is not set
# CONFIG_TCG_TPM is not set
# CONFIG_DCC_TTY is not set
# CONFIG_RAMOOPS is not set
CONFIG_S3C_MEM=y
CONFIG_EXYNOS_MEM=y
CONFIG_GPS_PM=y
CONFIG_MAX485_CTL=y
CONFIG_LEDS_CTL=y
# CONFIG_BUZZER_CTL is not set
CONFIG_ADC_CTL=y
CONFIG_RELAY_CTL=y
CONFIG_HELLO_CTL=y
CONFIG_I2C=y
CONFIG_I2C_BOARDINFO=y
CONFIG_I2C_COMPAT=y
CONFIG_I2C_CHARDEV=y
# CONFIG_I2C_MUX is not set
# CONFIG_I2C_HELPER_AUTO is not set
# CONFIG_I2C_SMBUS is not set
1516,8      50%
```

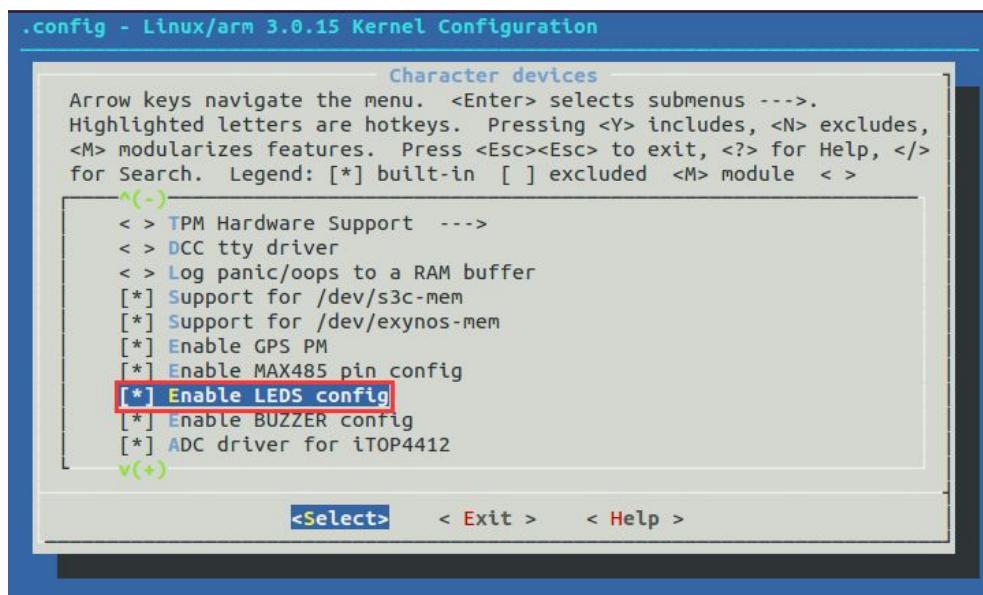
然后来看一下 menuconfig 中要怎么操作。如下图所示，进入 menuconfig 配置界面，找到 “Device Drivers” ，输入回车进入 “Device Drivers” 的配置界面。



如下图所示，找到“Character devices”，并进入下一级配置目录。

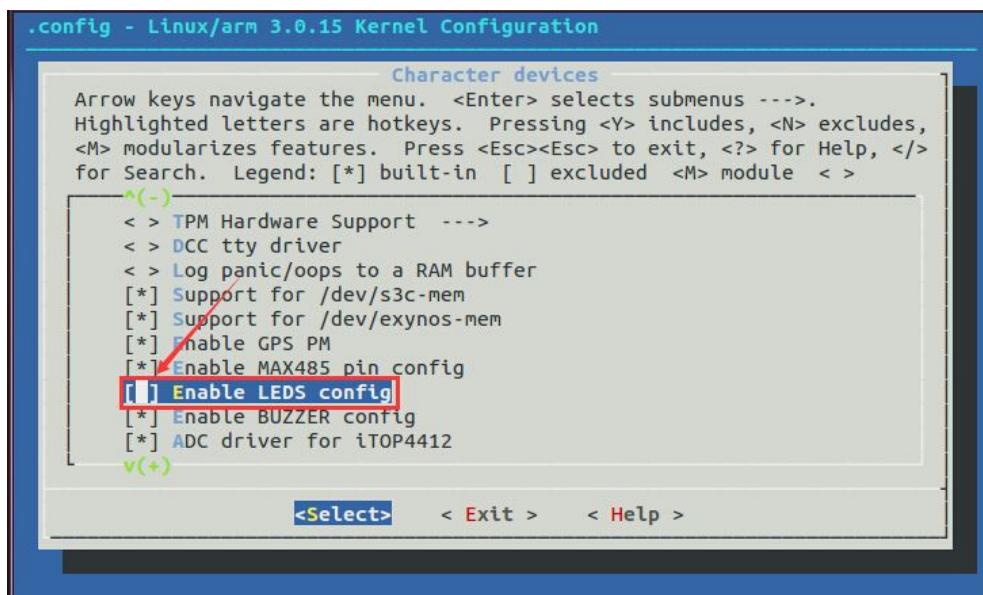


如下图所示，找到“Enable LEDS config”配置界面。

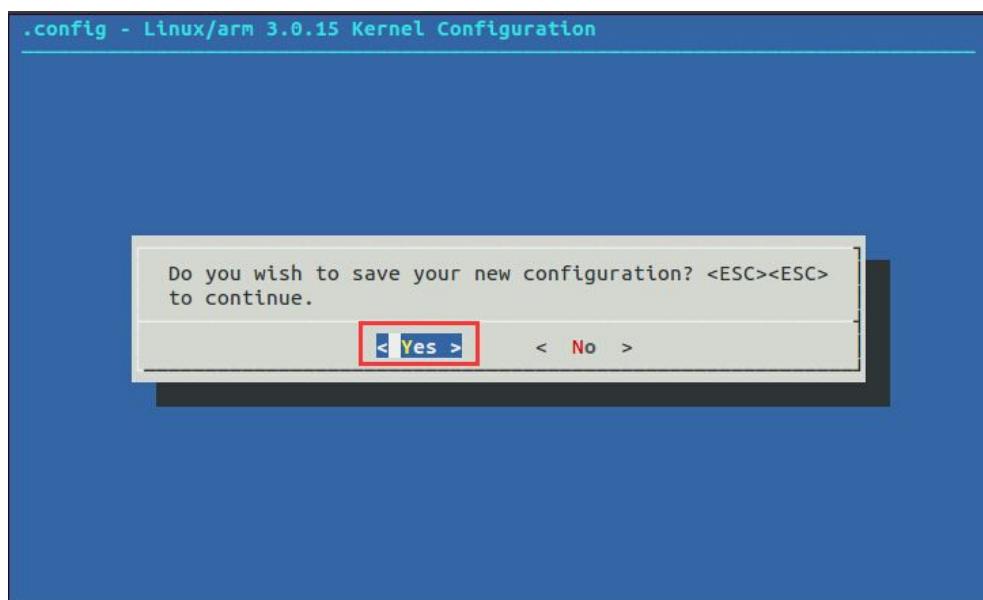


如上图所示，可以看到这个选项选为了“\*”（通过空格选择，前面已经介绍了 menuconfig 如何操作）。这里选择为“\*”，那么对应 “.config” 配置文件中的宏定义 “CONFIG\_LEDCTL”。

如下图，把默认配置的“\*”去掉，改成如下图所示的“空白”。



如下图所示，保存退出。



这个时候在看一下 “.config” 配置文件中的宏定义 “CONFIG\_LEDS\_CTL” 。

A screenshot of a terminal window displaying the contents of a ".config" file. The file contains numerous kernel configuration options. One specific line, "# CONFIG\_LEDS\_CTL is not set", is highlighted with a red rectangular box. The terminal window also shows the status bar at the bottom right with "1516,3" and "50%".

```
CONFIG_HW_RANDOM=y
# CONFIG_HW_RANDOM_TIMERIOMEM is not set
# CONFIG_R3964 is not set
# CONFIG_RAW_DRIVER is not set
# CONFIG_TCG_TPM is not set
# CONFIG_DCC_TTY is not set
# CONFIG_RAMOOPS is not set
CONFIG_S3C_MEM=y
CONFIG_EXYNOS_MEM=y
CONFIG_GPS_PM=y
CONFIG_MAX485_CTL=y
# CONFIG_LEDS_CTL is not set
CONFIG_BUZZER_CTL=y
CONFIG_ADC_CTL=y
CONFIG_RELAY_CTL=y
CONFIG_HELLO_CTL=y
CONFIG_I2C=y
CONFIG_I2C_BOARDINFO=y
CONFIG_I2C_COMPAT=y
CONFIG_I2C_CHARDEV=y
# CONFIG_I2C_MUX is not set
# CONFIG_I2C_HELPER_AUTO is not set
# CONFIG_I2C_SMBUS is not set
```

如上图所示，这个宏定义已经被注释掉了。也就是无法在编译过程中，需要用这个宏才能编译的 LEDS 小灯驱动，已经被裁减掉了。

最后大家还是要把 LEDS 的驱动配置上，后面还会用到。

### 3.6 Kconfig 和 menuconfig

这一小节来探讨一下 Kconfig 和 menuconfig 之间的关系。

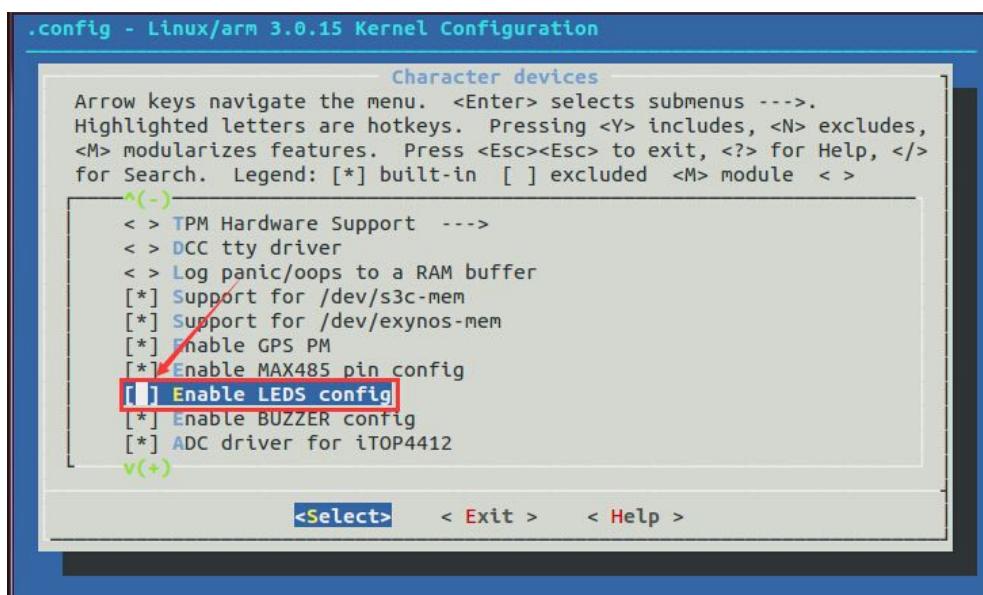
打开源码下的 “drivers/char/Kconfig” 文件，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/char/Kconfig
```

在这个文件里面搜索 “LEDS\_CTL” ，如下图所示。

```
config LEDS_CTL
    bool "Enable LEDS config"
    default y
    help
        Enable LEDS config
```

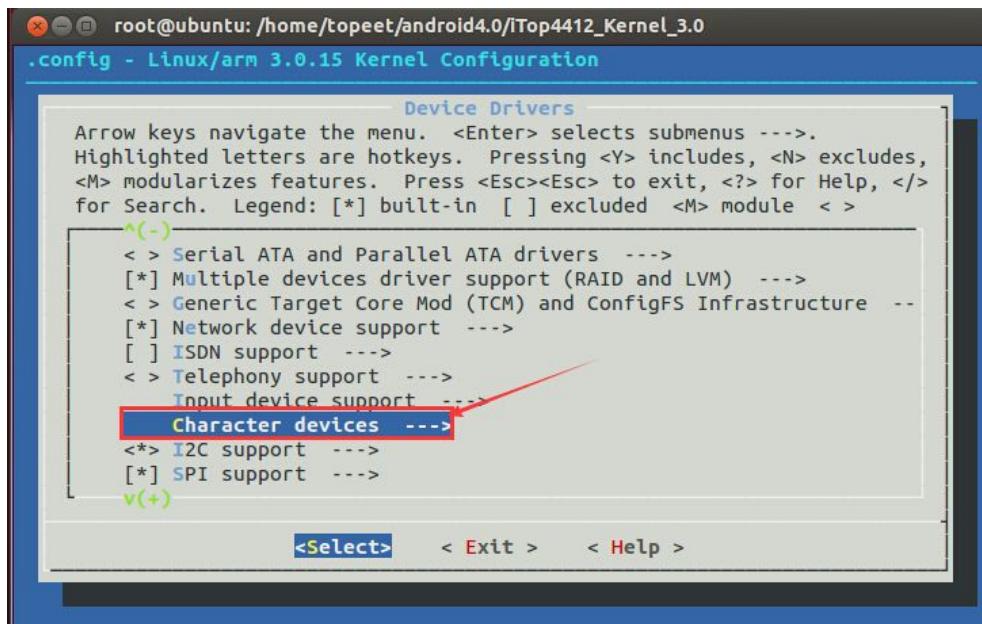
这里的 config LEDS\_CTL 就和 menuconfig 图形配置界面中 “Enable LEDS config” 对应，如下图所示。



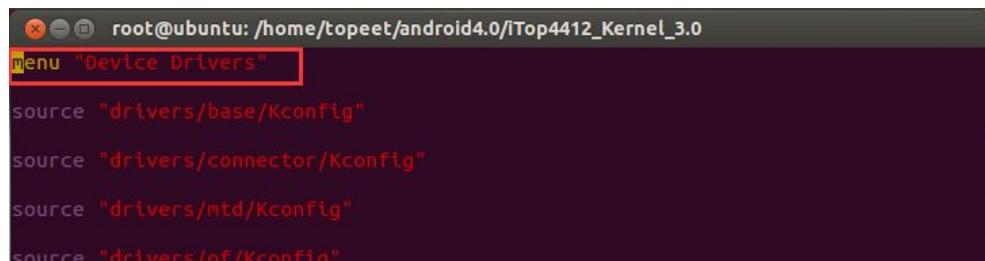
当然，讲到这里大家可能还有一个疑问，在 menuconfig 里面还有选择好几项才进入这个配置界面，其实那些选项是和 LEDS config 类似的，也有对应的配置。  
进入 “drivers/char/Kconfig” 文件的第一行，可以看到 “menu "Character devices”  
如下图所示。

```
# Character device configuration
#
menu "Character devices"
source "drivers/tty/Kconfig"
config DEVMEM
    bool "Memory device driver"
    default y
    help
        The memory driver provides two character devices, mem and kmem, which
```

如上图所示，这个 “Character devices” 就和 menuconfig 中的菜单对应，如下图所示。



然后打开 “drivers/Kconfig” 文件，如下图所示是 “Character devices” 的上一级菜单  
"Device Drivers"。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
menu "Device Drivers"
source "drivers/base/Kconfig"
source "drivers/connector/Kconfig"
source "drivers/mtd/Kconfig"
source "drivers/pf/Kconfig"
```

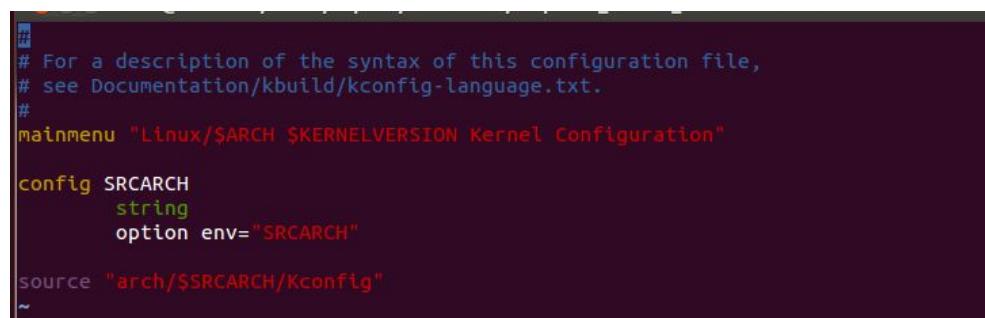
如下图所示，对应“Character devices”中的配置界面。



```
source "drivers/char/Kconfig"
source "drivers/i2c/Kconfig"
source "drivers/spi/Kconfig"
source "drivers/pps/Kconfig"
/char
```

这里特别提醒：无论是 Kconfig 最底层的配置，还是这里菜单的语法，用户只需要仿照者写就可以了。以为无论将来你从哪里拿到一份内核源码，这些框架都已经搭建好了，这一块都是学会仿写即可。千万不要花费太多的时间去学习这个脚本语法，到使用的时候简单的看一下就可以理解的，即使不能理解，依葫芦画瓢就可以了。

最后还有最顶层的 Kconfig 文件，打开来看一下，如下图所示。



```
#
# For a description of the syntax of this configuration file,
# see Documentation/kbuild/kconfig-language.txt.
#
mainmenu "Linux/$ARCH $KERNELVERSION Kernel Configuration"

config SRCARCH
    string
    option env="SRCARCH"

source "arch/$SRCARCH/Kconfig"
~
```

如上图所示，这里简单的介绍一下具体含义。

注释主要说的是配置文件的帮助文档。

mainmenu "Linux/\$ARCH \$KERNELVERSION Kernel Configuration"

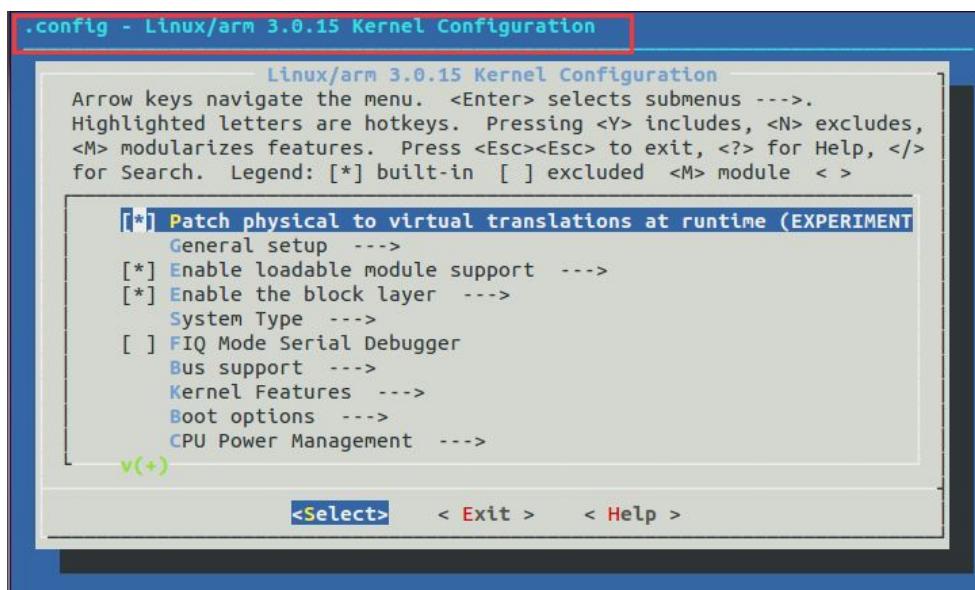
是主菜单的标题

\$ARCH 是定义在 Makefile 中的一个变量。指的是代码运行的具体环境，linux 运行在开发板上，毫无疑问是 arm。

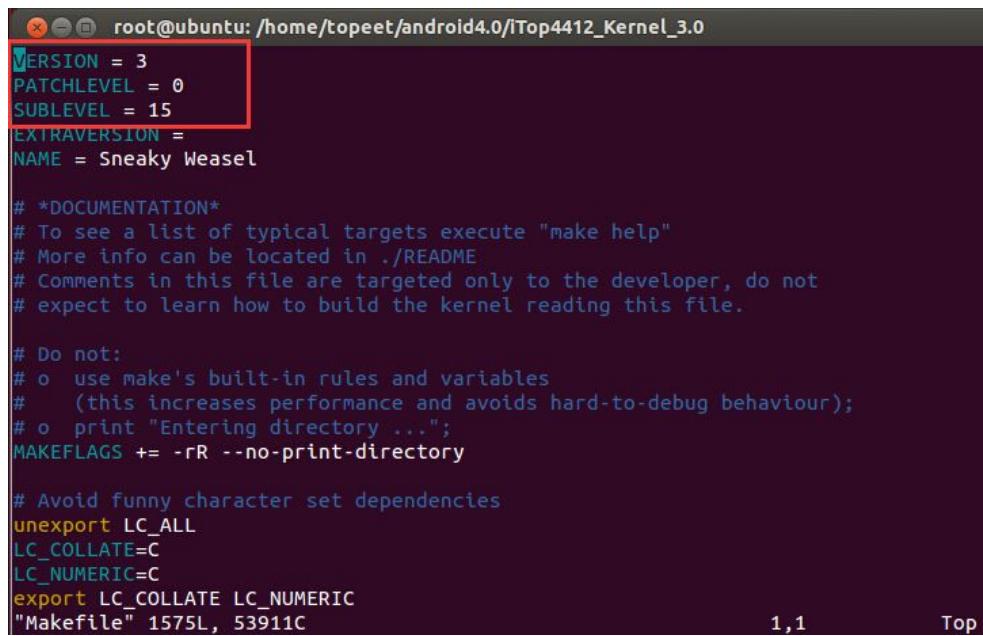
\$KERNELVERSION 是内核版本号

\$SRCARCH 也是 arm，在 Makefile 中可以看到定义，值和 ARCH 一样。

和下图中 menuconfig 顶层菜单对应。



Menuconfig 会读取 Makefile 中的参数，其中之一是内核版本号，如果修改一下参数，menuconfig 下主菜单标题就会改变，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 15
EXTRAVERSION =
NAME = Sneaky Weasel

# *DOCUMENTATION*
# To see a list of typical targets execute "make help"
# More info can be located in ./README
# Comments in this file are targeted only to the developer, do not
# expect to learn how to build the kernel reading this file.

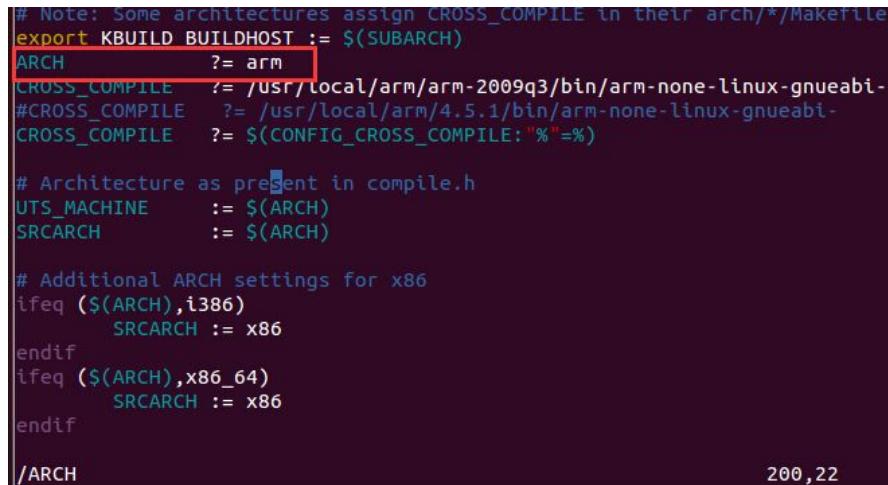
# Do not:
# o use make's built-in rules and variables
#   (this increases performance and avoids hard-to-debug behaviour);
# o print "Entering directory ...";
MAKEFLAGS += -rR --no-print-directory

# Avoid funny character set dependencies
unexport LC_ALL
LC_COLLATE=C
LC_NUMERIC=C
export LC_COLLATE LC_NUMERIC
"Makefile" 1575L, 53911C
```

1,1

Top

在 Makefile 文件中，ARCH 变量定义如下，在 ARCH 下定义七行的地方 SRCARCH 指定为 ARCH,也就是 ARM , 如下图所示。



```
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
export KBUILD_BUILDDHOST := ${SUBARCH}
ARCH ?= arm
CROSS_COMPILE ?= /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-
#CROSS_COMPILE ?= /usr/local/arm/4.5.1/bin/arm-none-linux-gnueabi-
CROSS_COMPILE ?= ${CONFIG_CROSS_COMPILE}%"=%)

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

# Additional ARCH settings for x86
ifeq ($(ARCH),i386)
    SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
    SRCARCH := x86
endif
```

/ARCH

200,22

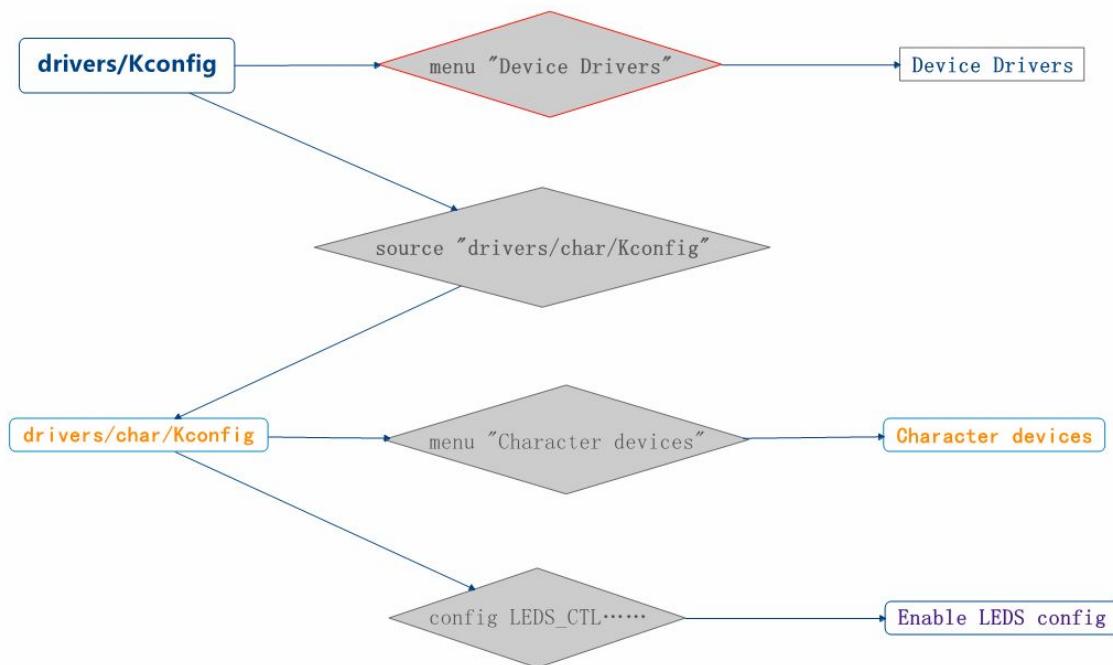
12%

上图中的路径就是前面搭建环境时候新建文件夹的路径 ,将来碰到什么版本的 Linux 内核 , 搭建环境的时候 , 这两个地方都需要对应起来 , 不然刚刚编译就会报一个错误 , 提示找到不编译器。

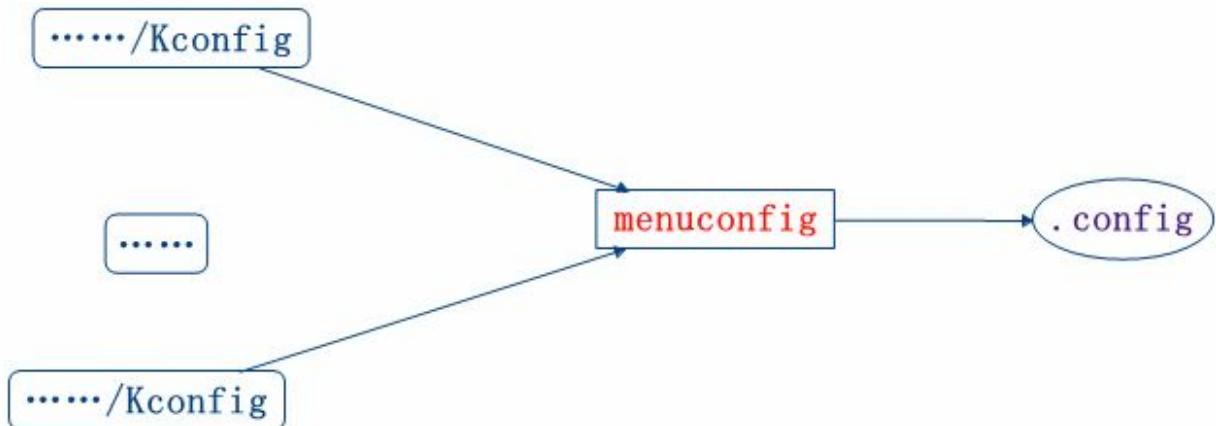
配置里面还有其它的一些知识点，例如依赖、反向依赖、三态变量等，到后面用到的时候再介绍。

### 3.7 图解 Kconfig 和 menuconfig 的关系

下图是以 LEDS 小灯驱动为例，图解 Kconfig 和 menuconfig 的关系。



如下图所示，最终得到配置需要的.config 文件。



### 3.8 其它配置文件

现在看一下提供源码中的，除了 “.config” 文件以外，还有其它的 config\_for\_xxx,如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch                               Documentation      lib                   REPORTING-BUGS
binary                             drivers          MAINTAINERS       samples
block                             firmware        Makefile           scripts
config_for_android                 fs              mm                  security
config_for_android_2M               include         init              modem.patch
config_for_linux                   ipc             Kbuild            modules.builtin
config_for_ubuntu                 kernel          Kconfig           modules.order
config_for_ubuntu_hdmi             kernel_readme.txt  net              Module.symvers
COPYING                           Documentation    kernel           README
CREDITS                           lib             kernel           pull_log.bat
crypto                            Documentation  lib             README
kernel                           lib             kernel           vmlinux
kernel_readme.txt                  Documentation lib             vmlinux.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

这些都是为了不同的文件系统准备的，有 Android 的配置文件有 Qt 的配置文件等等，或者特殊功能的 config 文件。这些 config 文件都是通过 menuconfig 生成，然后改成易识别的名称。它们公用一套代码，通过 menuconfig 裁减组合成不同功能.config，下一小节介绍 Make 命令，通过 Make 命令生成不同的内核。

注意用户实际源码下的 config\_for\_xxx 可能和上图有区别，以用户实际解压后的为准。

# 实验 04\_Makefile 编译

## 4.1 本章导读

在前面入门视频第十期“搭建编译环境 uboot\_linux\_Android”中，只介绍了输入 Make 命令就可以编译出内核，并没有介绍它是如何运行的。

在向内核中添加驱动的时候要完成 3 项工作，包括：

- 1 ) 在 Kconfig 中添加新代码对应项目的编译条件，这一部分在实验 3 已经介绍过；
- 2 ) 将驱动源码添加到对应的目录中，这一步比较好理解，在本章实验中将给大家演示一下；
- 3 ) 在目录 Makefile 中文件中增加针对新代码的编译条目，这一部分将在本章节介绍到。

### 4.1.1 工具

#### 4.1.1.1 硬件工具

- 1 ) PC 机

#### 4.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) Ubuntu 系统下解压生成的 Linux 源码

### 4.1.2 预备课程

入门视频“01-烧写、编译以及基础知识视频”

→ “实验 10-搭建编译环境 uboot\_linux\_Android” 或者使用手册 “五 Android 开发环境搭建以及编译”；

### 实验三 Menuconfig\_Kconfig

#### 4.1.3 视频资源

本节配套视频为 “视频 04\_Makefile 编译”

#### 4.2 学习目标

本章需要学习以下内容：

掌握 Linux 内核编译命令

掌握编译器路径设置的方法

理解环境变量路径、编译器、源码 Makefile 文件中编译器路径三者之间的关系

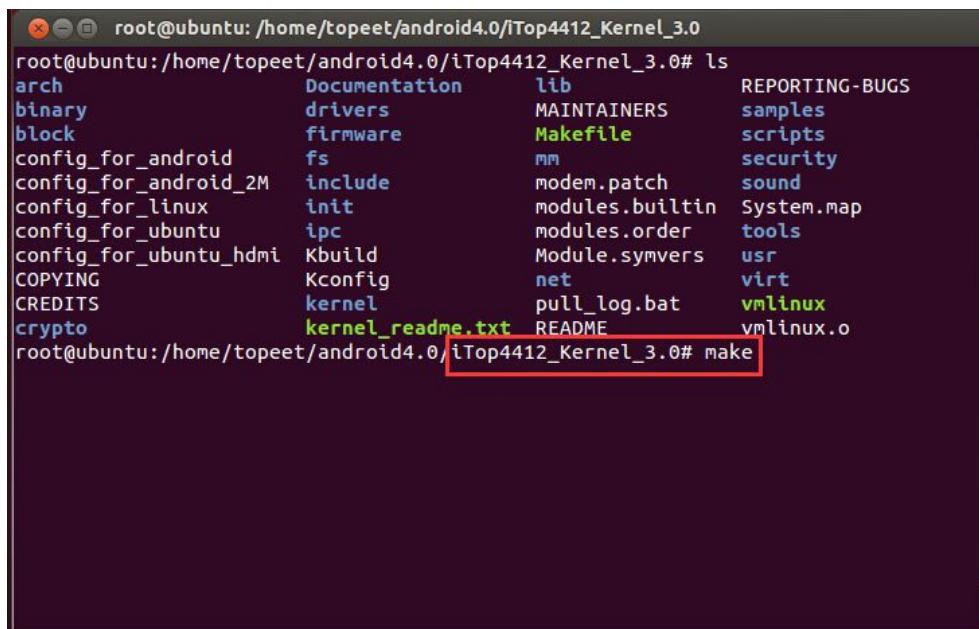
理解 Makefile 文件结构

掌握向 Makefile 文件中添加脚本命令的方法，能够看懂 Makefile 脚本

#### 4.3 编译器路径的设置

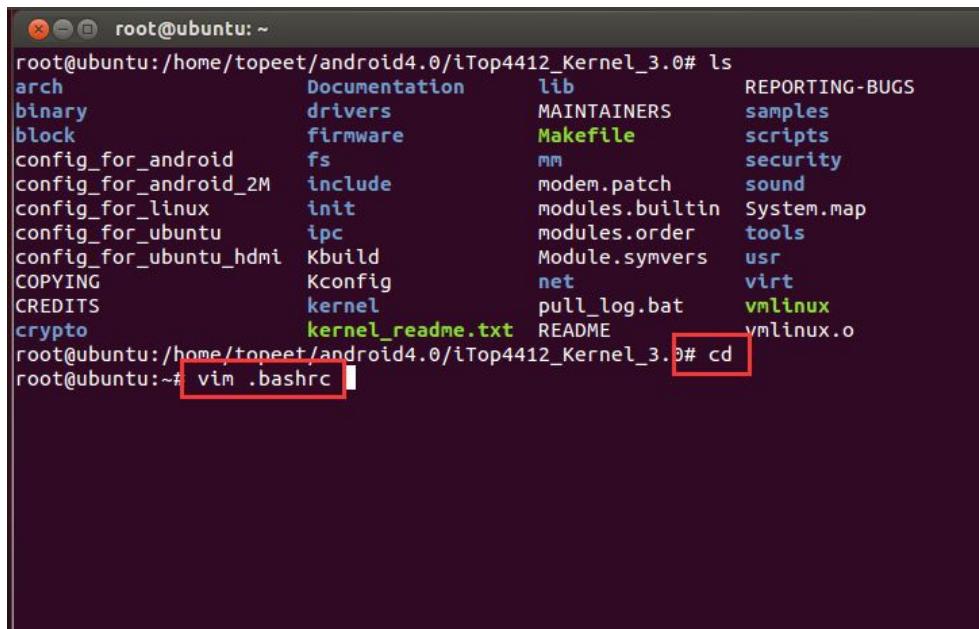
在前面入门知识中，只介绍了在什么目录解压编译器，在环境变量文件中添加路径，就可以编译出内核，并没有介绍他们之间是如何对应的。

如下图所示，按照前面提供的基础教程，配置好 Ubuntu 中的编译器和库文件，输入命令“#make”，就可以编译生成内核的二进制镜像。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile      scripts
config_for_android  fs           mm           security
config_for_android_2M include       modules.builtin System.map
config_for_linux   init          modules.order  tools
config_for_ubuntu  ipc           Module.symvers  usr
config_for_ubuntu_hdmi Kbuild      net           virt
COPYING        Kconfig        pull_log.bat vmlinux
CREDITS        kernel        README        vmlinux.o
crypto         kernel_readme.txt
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make
```

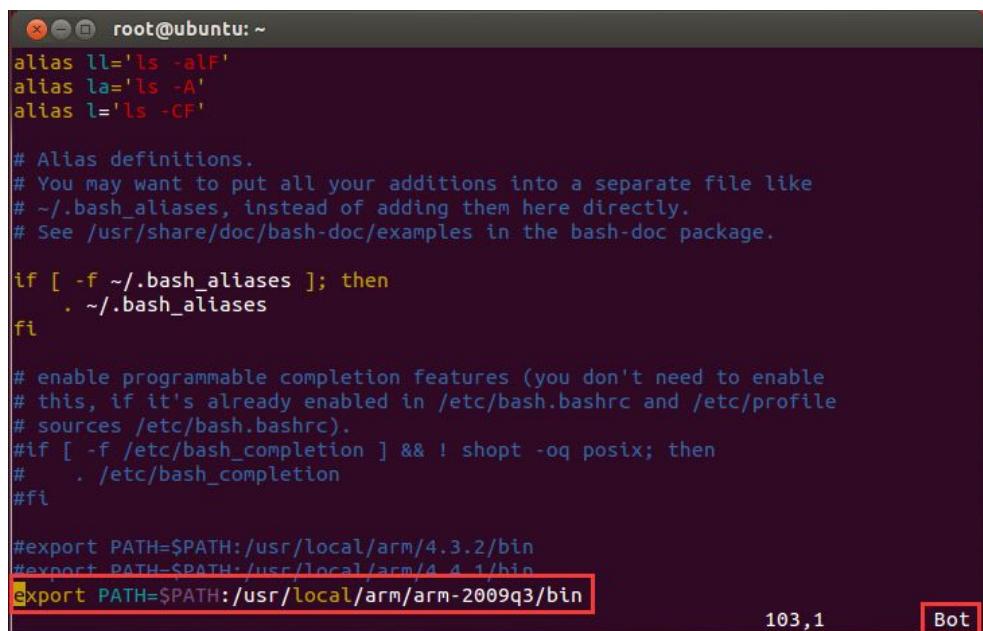
如下图所示，使用命令 "#cd" ，使用命令 "#vim .bashrc" 打开环境变量文件 ".bashrc"。



```
root@ubuntu:~#
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile      scripts
config_for_android  fs           mm           security
config_for_android_2M include       modules.builtin System.map
config_for_linux   init          modules.order  tools
config_for_ubuntu  ipc           Module.symvers  usr
config_for_ubuntu_hdmi Kbuild      net           virt
COPYING        Kconfig        pull_log.bat vmlinux
CREDITS        kernel        README        vmlinux.o
crypto         kernel_readme.txt
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# cd
root@ubuntu:~# vim .bashrc
```

如下图所示，进入最底行，可以看到，在环境变量文件中编译器路径设置为 "export

PATH=\$PATH:/usr/local/arm/arm-2009q3/bin" 。



```
root@ubuntu: ~
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
#if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
#    . /etc/bash_completion
#fi

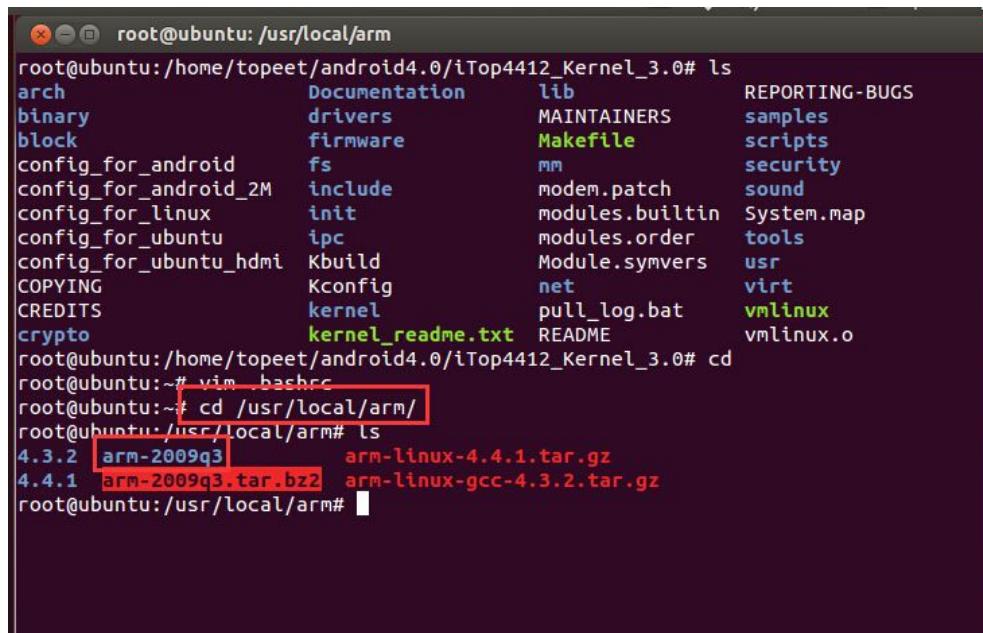
#export PATH=$PATH:/usr/local/arm/4.3.2/bin
#export PATH=$PATH:/usr/local/arm/4.4.1/bin
export PATH=$PATH:/usr/local/arm/arm-2009q3/bin
```

103,1

Bot

然后使用命令 “#cd /usr/local/arm/” , 进入解压缩译器 “arm-2009q3.tar.bz2” 的文件

目录。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation   lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware       Makefile      scripts
config_for_android  fs          mm           security
config_for_android_2M include      modem.patch
config_for_linux   init         modules.builtin
config_for_ubuntu  ipc          modules.order
config_for_ubuntu_hdmi Kbuild      Module.symvers
COPYING        Kconfig       net           virt
CREDITS        kernel       pull_log.bat
crypto         kernel_readme.txt README
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# cd
root@ubuntu:~# vim .bashrc
root@ubuntu:~# cd /usr/local/arm/
root@ubuntu:/usr/local/arm# ls
4.3.2  arm-2009q3          arm-linux-4.4.1.tar.gz
4.4.1  arm-2009q3.tar.bz2  arm-linux-gcc-4.3.2.tar.gz
root@ubuntu:/usr/local/arm#
```

可以明显的看到，环境变量中设置的路径和解压的路径是对应的。

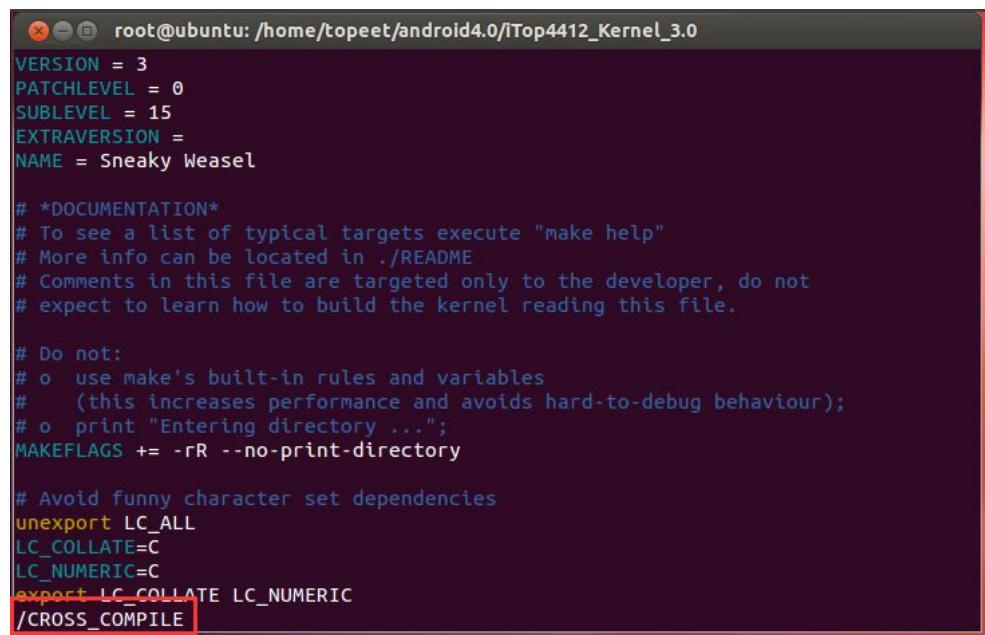
这里回到内核源码的目录，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
config_for_ubuntu_hdmi Kbuild      Module.symvers  usr
COPYING                 Kconfig      net            virt
CREDITS                kernel      pull_log.bat  vmlinu
crypto                 kernel_readme.txt README      vmlinu.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# cd
root@ubuntu:~# vim .bashrc
root@ubuntu:~# cd /usr/local/arm/
root@ubuntu:/usr/local/arm# ls
4.3.2 arm-2009q3      arm-linux-4.4.1.tar.gz
4.4.1 arm-2009q3.tar.bz2 arm-linux-gcc-4.3.2.tar.gz
root@ubuntu:/usr/local/arm# cd /home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile     scripts
config_for_android fs            mm           security
config_for_android_2M include       modem.patch sound
config_for_linux  init          modules.builtin System.map
config_for_ubuntu ipc           modules.order  tools
config_for_ubuntu_hdmi Kbuild      Module.symvers  usr
COPYING       Kconfig      pull_log.bat  virt
CREDITS      kernel      README      vmlinu
crypto       kernel_readme.txt vmlinu.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

如下图所示，使用命令 “#vim Makefile” 打开内核目录下的 Makefile 文件。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
COPYING                 Kconfig      net            virt
CREDITS                kernel      pull_log.bat  vmlinu
crypto                 kernel_readme.txt README      vmlinu.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# cd
root@ubuntu:~# vim .bashrc
root@ubuntu:~# cd /usr/local/arm/
root@ubuntu:/usr/local/arm# ls
4.3.2 arm-2009q3      arm-linux-4.4.1.tar.gz
4.4.1 arm-2009q3.tar.bz2 arm-linux-gcc-4.3.2.tar.gz
root@ubuntu:/usr/local/arm# cd /home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile     scripts
config_for_android fs            mm           security
config_for_android_2M include       modem.patch sound
config_for_linux  init          modules.builtin System.map
config_for_ubuntu ipc           modules.order  tools
config_for_ubuntu_hdmi Kbuild      Module.symvers  usr
COPYING       Kconfig      pull_log.bat  virt
CREDITS      kernel      README      vmlinu
crypto       kernel_readme.txt vmlinu.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

这里通过 vim 编辑器查找参数 “CROSS\_COMPILE” ，如下图所示。



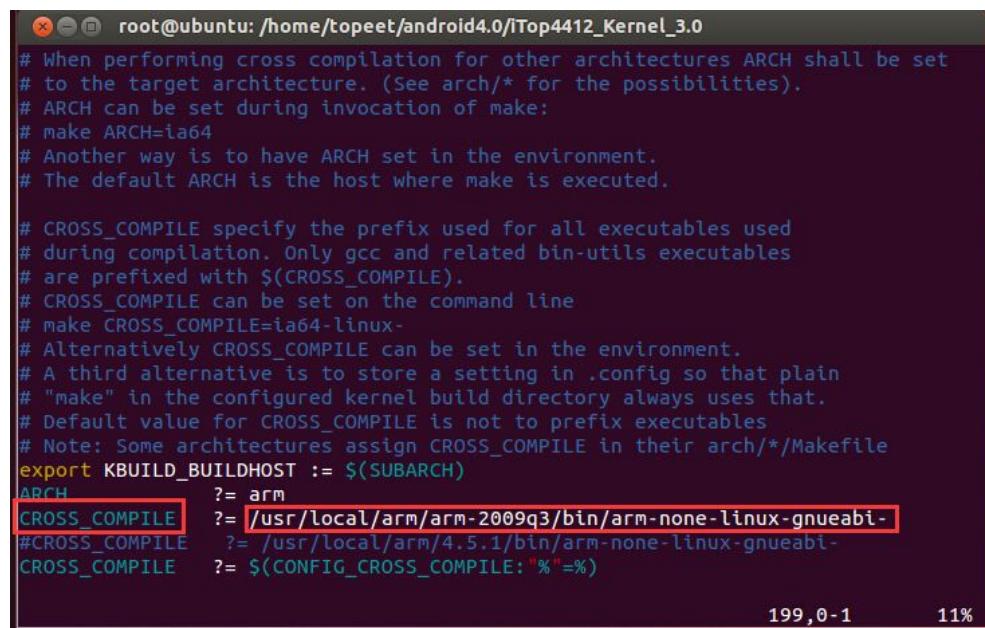
```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 15
EXTRAVERSION =
NAME = Sneaky Weasel

# *DOCUMENTATION*
# To see a list of typical targets execute "make help"
# More info can be located in ./README
# Comments in this file are targeted only to the developer, do not
# expect to learn how to build the kernel reading this file.

# Do not:
# o use make's built-in rules and variables
#   (this increases performance and avoids hard-to-debug behaviour);
# o print "Entering directory ...";
MAKEFLAGS += -fR --no-print-directory

# Avoid funny character set dependencies
unexport LC_ALL
LC_COLLATE=C
LC_NUMERIC=C
export LC_CCOLLATE LC_NUMERIC
/CROSS_COMPILE
```

可以查到参数“CROSS\_COMPILE”，如下图所示。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
# When performing cross compilation for other architectures ARCH shall be set
# to the target architecture. (See arch/* for the possibilities).
# ARCH can be set during invocation of make:
# make ARCH=ia64
# Another way is to have ARCH set in the environment.
# The default ARCH is the host where make is executed.

# CROSS_COMPILE specify the prefix used for all executables used
# during compilation. Only gcc and related bin-utils executables
# are prefixed with $(CROSS_COMPILE).
# CROSS_COMPILE can be set on the command line
# make CROSS_COMPILE=ia64-linux-
# Alternatively CROSS_COMPILE can be set in the environment.
# A third alternative is to store a setting in .config so that plain
# "make" in the configured kernel build directory always uses that.
# Default value for CROSS_COMPILE is not to prefix executables
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH      ?= arm
CROSS_COMPILE ?= /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-
#CROSS_COMPILE ?= /usr/local/arm/4.5.1/bin/arm-none-linux-gnueabi-
CROSS_COMPILE ?= $(CONFIG_CROSS_COMPILE: "%=%")
```

如上图所示，可以看到这个参数是

“/usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-”。

再看一下参数 “CROSS\_COMPILE” 的下一行，可以看出这里有修改过的痕迹。很容易推断出，原来三星是用的 “4.5.1” 版本的编译器，不过这个没关系，只要能编译通过就可以了。

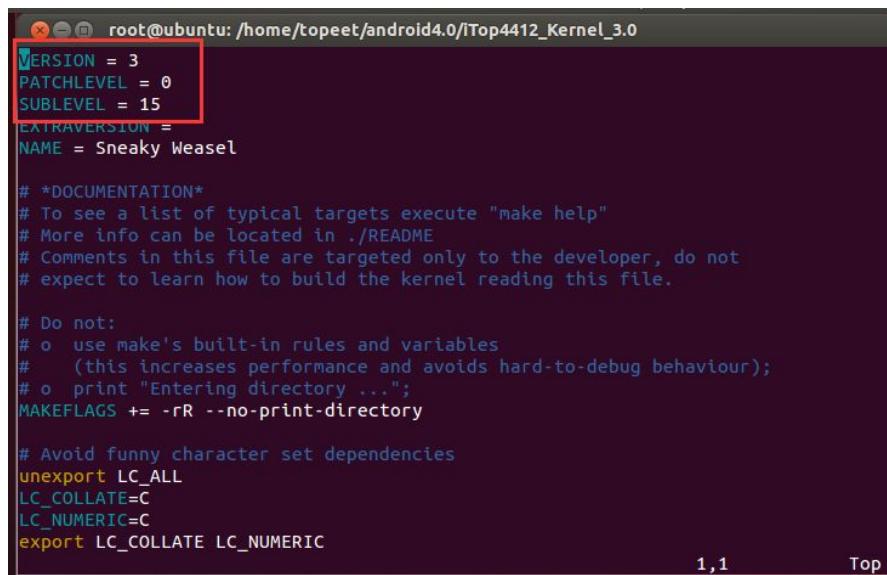
一般说来，拿到源码之后的第一步，是要先要先将源码编译通过。

设置上图中的变量 “CROSS\_COMPILE” 以及环境变量还有编译器实际解压路径三者对应。这三者对应之后，就能确保执行 Make 命令之后，系统能够找到这个编译器。

在编译命令执行的过程中，会提示一些错误，然后根据提示的错误，挨个去排查修改，去添加库文件或者修改库文件。这个过程有长有短，考验的只是耐心，并没有太多技巧。由于已经将缺少的库文件写成脚本，这样大家在编译器执行以下脚本就能够编译通过。

针对内核目录下的 Makefile 文件，提醒大家一下，这个文件中内容很多，除了上面编译器路径变量以外的其它部分几乎不用关注。

另外还有一个地方需要了解一下，如下图所示，在该文件的第一行，可以看到内核的版本。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 15
EXTRAVERSION =
NAME = Sneaky Weasel

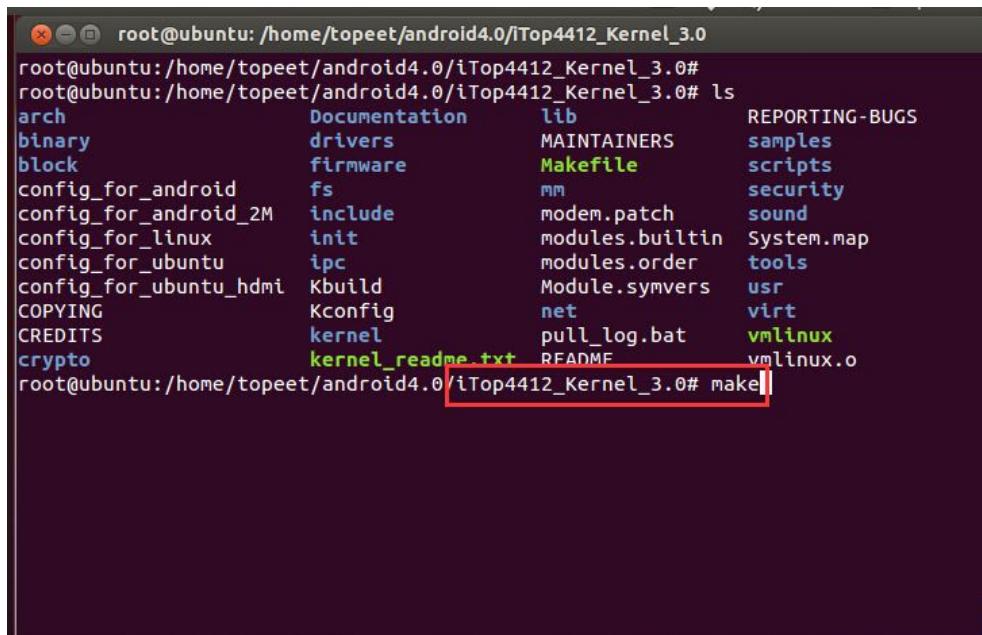
# *DOCUMENTATION*
# To see a list of typical targets execute "make help"
# More info can be located in ./README
# Comments in this file are targeted only to the developer, do not
# expect to learn how to build the kernel reading this file.

# Do not:
# o use make's built-in rules and variables
#   (this increases performance and avoids hard-to-debug behaviour);
# o print "Entering directory ...";
MAKEFLAGS += -rR --no-print-directory

# Avoid funny character set dependencies
unexport LC_ALL
LC_COLLATE=C
LC_NUMERIC=C
export LC_COLLATE LC_NUMERIC
```

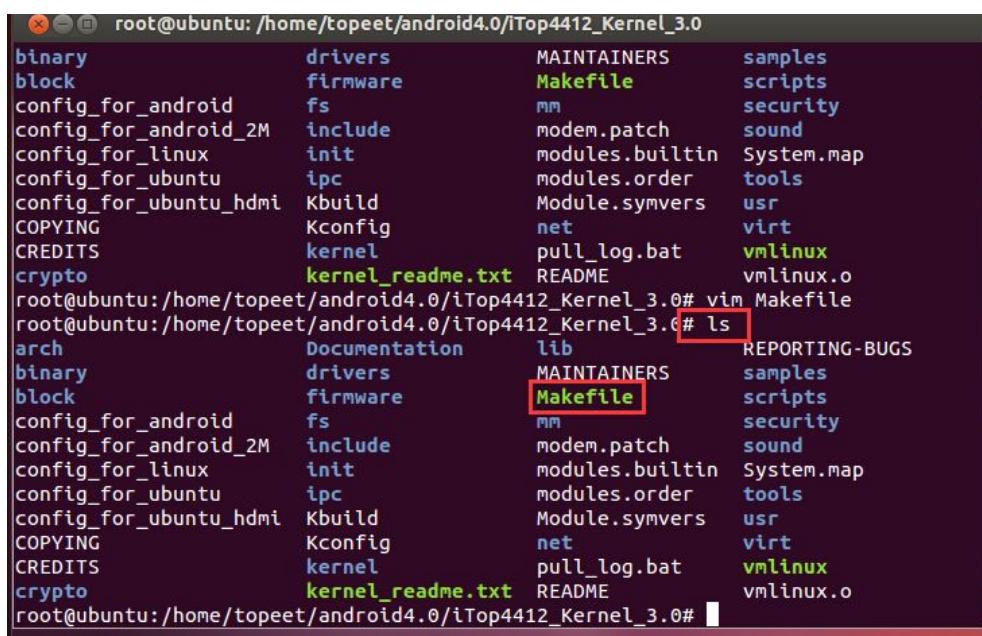
## 4.4 Make 内核编译命令执行过程简介

如下图所示，编译内核需要执行 make 命令，在执行 make 命令之后，并没有介绍它是如何执行的。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile      scripts
config_for_android  fs           mm           security
config_for_android_2M include       modem.patch sound
config_for_linux   init          modules.builtin System.map
config_for_ubuntu  ipc           modules.order  tools
config_for_ubuntu_hdmi Kbuild      Module.symvers usr
COPYING        Kconfig        net           virt
CREDITS        kernel        pull_log.bat vmlinux
crypto         kernel_readme.txt README      vmlinux.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make
```

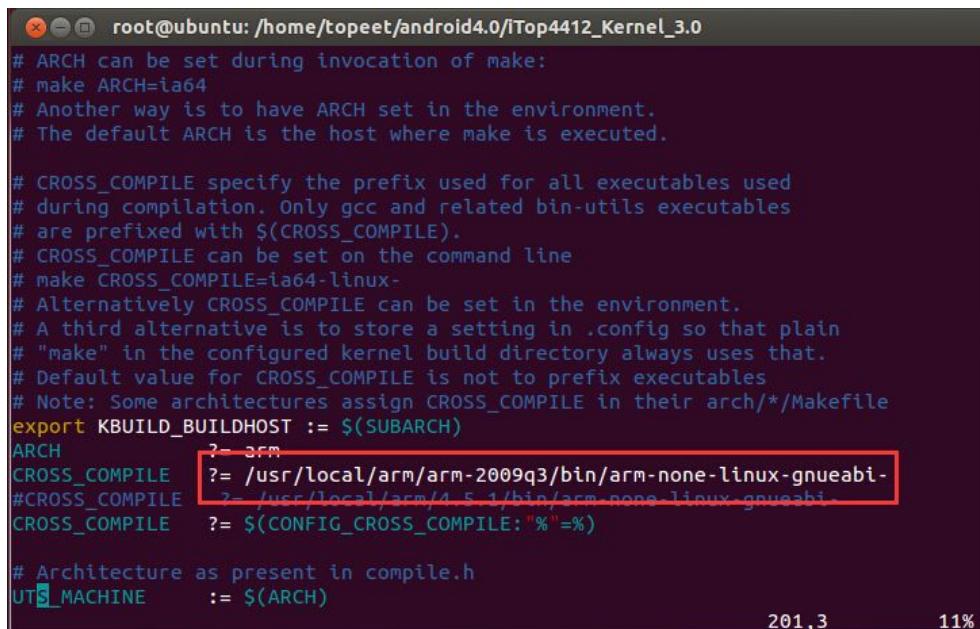
在 make 命令执行之后，它会首先找到当前目录下的“Makefile”文件，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
arch          Documentation    lib           REPORTING-BUGS
binary        drivers         MAINTAINERS   samples
block         firmware        Makefile      scripts
config_for_android  fs           mm           security
config_for_android_2M include       modem.patch sound
config_for_linux   init          modules.builtin System.map
config_for_ubuntu  ipc           modules.order  tools
config_for_ubuntu_hdmi Kbuild      Module.symvers usr
COPYING        Kconfig        net           virt
CREDITS        kernel        pull_log.bat vmlinux
crypto         kernel_readme.txt README      vmlinux.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls
```

运行 Makefile 文件之后，它会在 Makefile 文件中找到编译器的路径

“/usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-” 如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
# ARCH can be set during invocation of make:
# make ARCH=ia64
# Another way is to have ARCH set in the environment.
# The default ARCH is the host where make is executed.

# CROSS_COMPILE specify the prefix used for all executables used
# during compilation. Only gcc and related bin-utils executables
# are prefixed with $(CROSS_COMPILE).
# CROSS_COMPILE can be set on the command line
# make CROSS_COMPILE=ia64-linux-
# Alternatively CROSS_COMPILE can be set in the environment.
# A third alternative is to store a setting in .config so that plain
# "make" in the configured kernel build directory always uses that.
# Default value for CROSS_COMPILE is not to prefix executables
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH ?= arm
CROSS_COMPILE ?= /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-
#CROSS_COMPILE ?= /usr/local/arm/4.5.1/bin/arm-none-linux_gnueabi-
CROSS_COMPILE ?= $(CONFIG_CROSS_COMPILE:"%")=%

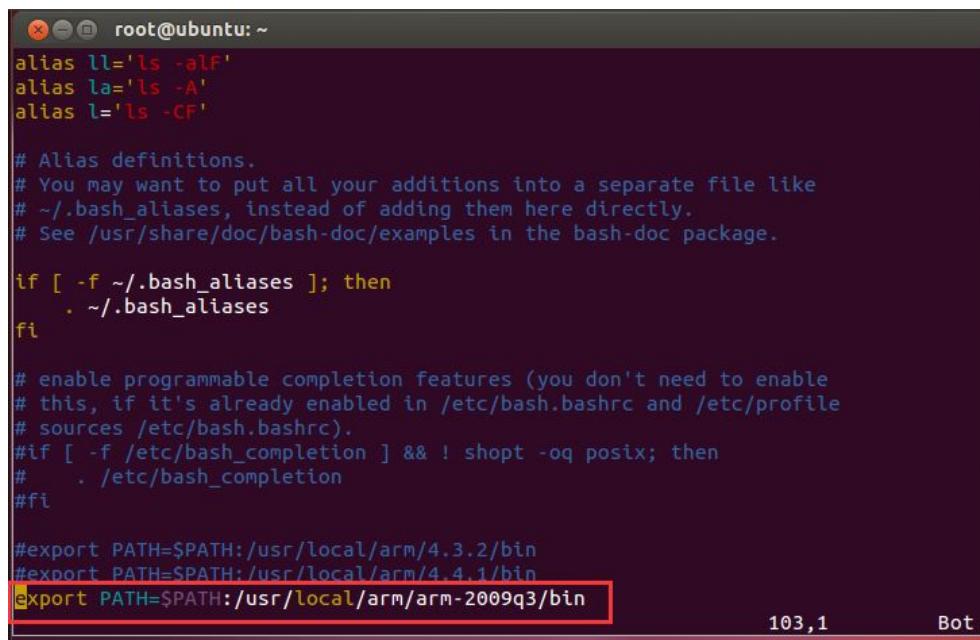
# Architecture as present in compile.h
UTS_MACHINE := $(ARCH)
```

201,3

11%

然后系统根据环境变量找到 “export PATH=\$PATH:/usr/local/arm/arm-2009q3/bin”

编译器的路径，如下图所示。



```
root@ubuntu:~
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
#if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
#    . /etc/bash_completion
#fi

#export PATH=$PATH:/usr/local/arm/4.3.2/bin
#export PATH=$PATH:/usr/local/arm/4.4.1/bin
export PATH=$PATH:/usr/local/arm/arm-2009q3/bin
```

103,1

Bot

如下图所示，使用命令 “ls /usr/local/arm/arm-2009q3/bin/” 可以查看到 arm2009q3  
解压之后有哪些具体编译器。

```

root@ubuntu:~#
config_for_linux      init          modules.builtin  System.map
config_for_ubuntu     ipc           modules.order    tools
config_for_ubuntu_hdmi Kbuild        Module.symvers  usr
COPYING               Kconfig       net             virt
CREDITS              kernel       pull_log.bat   vmlinu
crypto                kernel_readme.txt README        vmlinu.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# cd
root@ubuntu:~#
root@ubuntu:~# vim .hachrc
root@ubuntu:~# ls /usr/local/arm/arm-2009q3/bin/
arm-none-linux-gnueabi-addr2line  arm-none-linux-gnueabi-gprof
arm-none-linux-gnueabi-ar         arm-none-linux-gnueabi-ld
arm-none-linux-gnueabi-as         arm-none-linux-gnueabi-nm
arm-none-linux-gnueabi-c++       arm-none-linux-gnueabi-objcopy
arm-none-linux-gnueabi-c++filt   arm-none-linux-gnueabi-objdump
arm-none-linux-gnueabi-cpp       arm-none-linux-gnueabi-ranlib
arm-none-linux-gnueabi-g++       arm-none-linux-gnueabi-readelf
arm-none-linux-gnueabi-gcc       arm-none-linux-gnueabi-size
arm-none-linux-gnueabi-gcc-4.4.1  arm-none-linux-gnueabi-sprite
arm-none-linux-gnueabi-gcov      arm-none-linux-gnueabi-strings
arm-none-linux-gnueabi-gdb       arm-none-linux-gnueabi-strip
arm-none-linux-gnueabi-gdbtui
root@ubuntu:~#

```

如上图，系统找到编译器之后，同时基础的库文件也是和编译器在一起的，有时候编译一个新内核，还有可能需要修改库文件。

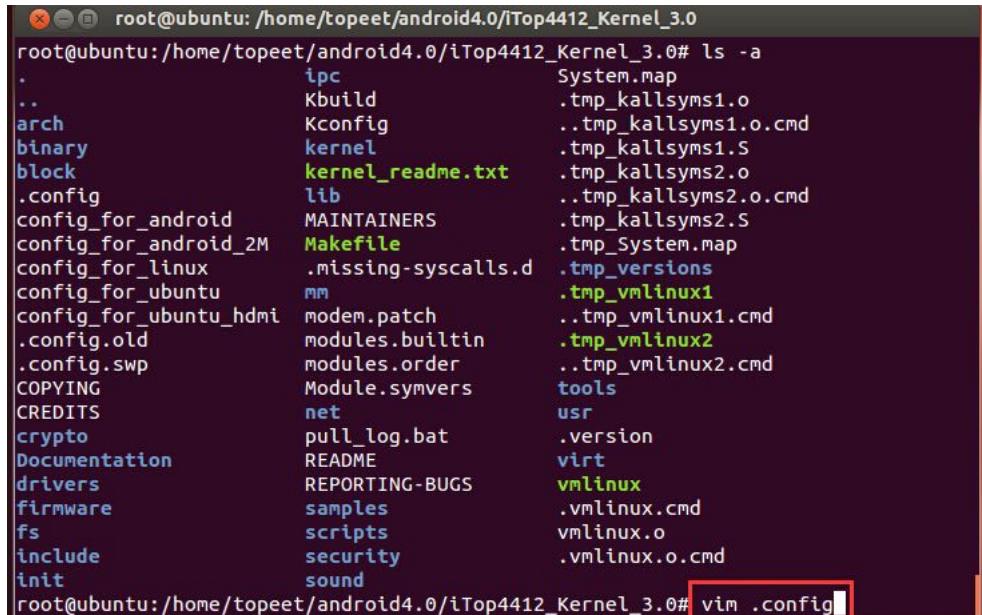
编译执行前还需要找到 “.config” 文件，默认是放在源码目录下的。使用命令 “ls -a” 就可以看到，如下图所示。

```

root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls -a
.                      ipc          system.map
..                     Kbuild      .tmp_kallsyms1.o
arch                  Kconfig      .tmp_kallsyms1.o.cmd
binary                kernel      .tmp_kallsyms1.S
block                 kernel_readme.txt
.config               lib          .tmp_kallsyms2.o
.config.old           lib          .tmp_kallsyms2.o.cmd
.config.swp          lib          .tmp_kallsyms2.S
COPYING              missing-syscalls.d .tmp_System.map
config_for_android   Makefile    .tmp_versions
config_for_android_2M config_for_android_2M .tmp_vmlinux1
config_for_linux      .tmp_vmlinux1
config_for_ubuntu     mm          .tmp_vmlinux1.cmd
config_for_ubuntu_hdmi config_for_ubuntu_hdmi .tmp_vmlinux2
.config.old           modules.builtin .tmp_vmlinux2
.config.swp          modules.order  .tmp_vmlinux2.cmd
COPYING              Module.symvers tools
crypto                net          usr
Documentation        pull_log.bat .version
drivers               README      virt
firmware              REPORTING-BUGS vmlinu
fs                   samples     .vmlinu.cmd
include               scripts     vmlinu.o
init                 security    .vmlinu.o.cmd
sound
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#

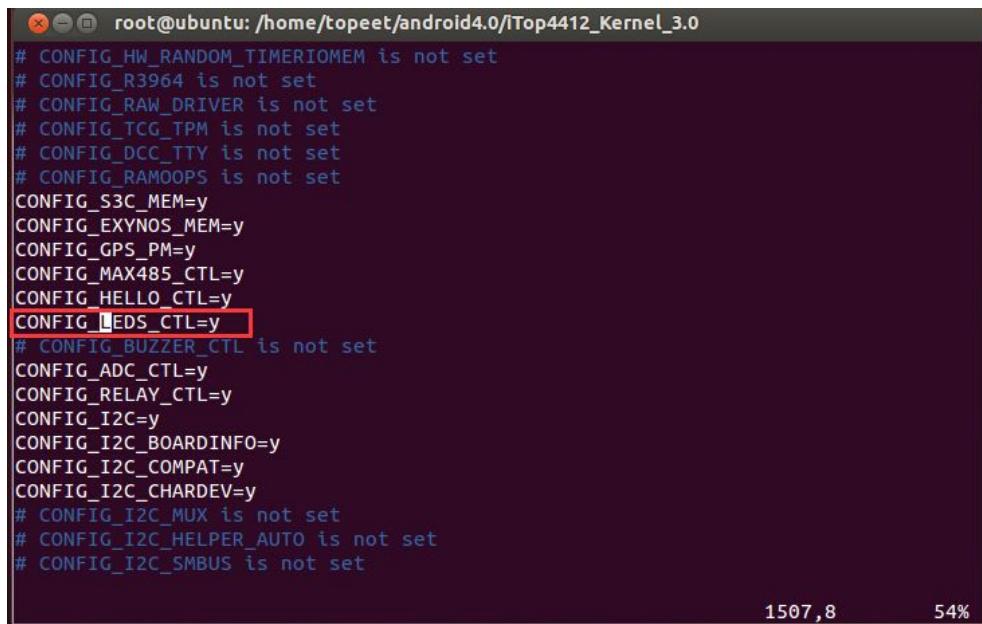
```

从前面的实验 3.5 小节，学习到这个 “.config” 是通过 menuconfig 工具生成的，里面只是一些宏定义，如下图所示，打开这个文件。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls -a
.
..
arch
binary
block
.config
config_for_android
config_for_android_2M
config_for_linux
config_for_ubuntu
config_for_ubuntu_hdmi
.config.old
.config.swp
COPYING
CREDITS
crypto
Documentation
drivers
firmware
fs
include
init
ipc
Kbuild
Kconfig
kernel
kernel_readme.txt
lib
MAINTAINERS
Makefile
.missing-sysscalls.d
mm
modem.patch
modules.builtin
modules.order
Module.symvers
net
pull_log.bat
README
REPORTING-BUGS
samples
scripts
security
sound
System.map
.tmp_kallsyms1.o
..tmp_kallsyms1.o.cmd
.tmp_kallsyms1.S
.tmp_kallsyms2.o
..tmp_kallsyms2.o.cmd
.tmp_kallsyms2.S
.tmp_System.map
.tmp_versions
.tmp_vmlinux1
..tmp_vmlinux1.cmd
.tmp_vmlinux2
..tmp_vmlinux2.cmd
.tools
usr
.version
virt
vmlinux
.vmlinux.cmd
.vmlinux.o
.vmlinux.o.cmd
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim .config
```

查找宏定义“LEDS”，是编译 led 驱动的宏定义“LEDS\_CTL”，如下图所示。



```
# CONFIG_HW_RANDOM_TIMERIOMEM is not set
# CONFIG_R3964 is not set
# CONFIG_RAW_DRIVER is not set
# CONFIG_TCG_TPM is not set
# CONFIG_DCC_TTY is not set
# CONFIG_RAMOOPS is not set
CONFIG_S3C_MEM=y
CONFIG_EXYNOS_MEM=y
CONFIG_GPS_PM=y
CONFIG_MAX485_CTL=y
CONFIG_HELLO_CTL=y
CONFIG_LEDCTL=y
# CONFIG_BUZZER_CTL is not set
CONFIG_ADC_CTL=y
CONFIG_RELAY_CTL=y
CONFIG_I2C=y
CONFIG_I2C_BOARDINFO=y
CONFIG_I2C_COMPAT=y
CONFIG_I2C_CHARDEV=y
# CONFIG_I2C_MUX is not set
# CONFIG_I2C_HELPER_AUTO is not set
# CONFIG_I2C_SMBUS is not set
```

现在系统找到这个宏定义“LEDS\_CTL”，在编译具体中间文件的时候会用到。

具体是怎么实现的，继续看下一小节的内容就可以搞清楚这个宏定义有什么用处。

## 4.5 Makefile 文件

接前一小节的内容，继续介绍系统是如何一步一步编译出内核镜像的。

在这里仍然以 LEDS 小灯为例。

### 4.5.1 宏定义 LEDS\_CTL 的使用

如下图所示，led 驱动属于字符驱动，字符驱动一般是在源码目录 “drivers/char/” 下。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  mem.o          raw.c
apm-emulation.c  hpet.c        misc.c        rtc.c
applicom.c    hw_random       misc.o        s3c_mem.c
applicom.h    i8k.c         mmtimer.c    s3c_mem.h
bfin-otp.c     ipmi          modules.builtin  s3c_mem.o
briq_panel.c   itop4412_adc.c  modules.order  scc.h
bsr.c         itop4412_adc.o  msm_smd_pkt.c scx200_gpio.c
built-in.o     itop4412_buzzer.c mspec.c      snscc.c
dcc_tty.c      itop4412_buzzer.o  mwave        snscc_event.c
ds1302.c       itop4412_leds.c  nsc_gpio.c  snscc.h
ds1620.c       itop4412_leds.o  nvram.c      sonypi.c
dsp56k.c       itop4412_relay.c nwbutton.c  tb0219.c
dtlk.c         itop4412_relay.o nwbutton.h  tclk.c
efirc.c        Kconfig        nwflash.c   toshiba.c
exynos_mem.c   lp.c          pc8736x_gpio.c  tpm
exynos_mem.o   Makefile      pcmcia      ttyprintk.c
generic_nvram.c max485_ctl.c  ppdev.c     uv_mmtimer.c
genrtc.c       max485_ctl.o  ps3flash.c  viotape.c
gps.c          mbc.s        ramoops.c  virtio_console.c
gps.h          mbc.s        random.c   xilinx_hwic平
gps.o          mem.c        random.o
```

如上图所示，这里要关注的文件只有框框中的三个 “itop4412\_leds.c” ，

“itop4412\_leds.o” ， “Makefile” 。

itop4412\_leds.c : 比较好理解，就是 led 驱动的源码

itop4412\_leds.o : 这个是生成最终 zImage 二进制的中间文件

Makefile : 就是 Make 命令所需要的文件

这里不关心 itop4412\_leds.c 中的代码，只需先知道 itop4412\_leds.c 的文件名即可。在源码目录中使用命令 “vim drivers/char/Makefile” ，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
agp          hangcheck-timer.c    misc.o        s3c_mem.c
apm-emulation.c hpet.c          mm timer.c   s3c_mem.h
applicom.c    hw_random         modules.builtin s3c_mem.o
applicom.h     i8k.c           modules.order   scc.h
bfin-otp.c    ipmi            msm_smd_pkt.c scx200_gpio.c
briq_panel.c  itop4412_adc.c   mspec.c       snsc.c
bsr.c         itop4412_adc.o   mwave          snsc_event.c
built-in.o    itop4412_buzzer.c nsc_gpio.c   snsc.h
dcc_tty.c     itop4412_leds.c  nvram.c      sonypi.c
ds1302.c      itop4412_relay.c nwbutton.c  tb0219.c
ds1620.c      itop4412_relay.o nwbutton.h  tlclk.c
dsp50k.c      Kconfig         nwflash.c    toshiba.c
dtlk.c        lp.c            pc8736x_gpio.c tpm
efirtc.c      Makefile        pcmcia        ttyprintk.c
exynos_mem.c  max485_ctl.c   ppdev.c     uv_mm timer.c
exynos_mem.o  max485_ctl.o   ps3flash.c  viotape.c
generic_nvram.c mbcs.c       ramoops.c   virtio_console.c
genrtc.c      mbcs.h        random.c    xilinx_hwicap
gps.c         mem.c          random.o   raw.c
gps.h         mem.o          rtc.c
gps.o         misc.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/char/Makefile
```

打开 “vim drivers/char/Makefile” 文件之后，找到和 itop4412\_leds.c 相关的文件，搜索 “itop4412\_leds” ,如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
obj-$(CONFIG_PCMCIA)          += pcmcia/
obj-$(CONFIG_IPMI_HANDLER)     += ipmi/

obj-$(CONFIG_HANGCHECK_TIMER)  += hangcheck-timer.o
obj-$(CONFIG_TCG_TPM)          += tpm/

obj-$(CONFIG_DCC_TTY)          += dcc_tty.o
obj-$(CONFIG_PS3_FLASH)        += ps3flash.o
obj-$(CONFIG_RAMOOPS)          += ramoops.o

obj-$(CONFIG_JS_RTC)          += js-rtc.o
js-rtc-y = rtc.o

obj-$(CONFIG_S3C_MEM)          += s3c_mem.o
obj-y += gps.o

obj-$(CONFIG_MAX485_CTL)
obj-$(CONFIG_LEDS_CTL)          += max485_ctl.o
                                += itop4412_leds.o
obj-$(CONFIG_BUZZER_CTL)
obj-$(CONFIG_ADC_CTL)
obj-$(CONFIG_RELAY_CTL)         += itop4412_buzzer.o
                                += itop4412_adc.o
                                += itop4412_relay.o

obj-$(CONFIG_EXYNOS_MEM)
/itop4412_leds
```

如上图方框所示，就是编译 itop4412\_leds.c 的脚本。这个脚本大家也只需要仿照写即可，严格的按照已有的例子来写即可。

```
| obj-$(CONFIG_MAX485_CTL)      += max485_ctl.o
| obj-$(CONFIG_LEDS_CTL)        += itop4412_leds.o
| obj-$(CONFIG_BUZZER_CTL)      += itop4412_buzzer.o
| obj-$(CONFIG_ADC_CTL)         += itop4412_adc.o
| obj-$(CONFIG_RELAY_CTL)       += itop4412_relay.o
|
```

如果想添加类似的字符驱动，就可以在这个目录下添加。

Kconfig、Makefile、menuconfig、“.config”文件大家就可以联系起来了。

#### 4.5.2 Makefile 脚本语法简介

本小节介绍的语法是对内核源码子目录中 Makefile 进行简单的介绍，这部分是经常会用到的，也是需要掌握的。

常用的强制编译写法，还是在“drivers/char/Makefile”中，在 Top 行，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
#
# Makefile for the kernel character device drivers.
#
obj-y                         += mem.o random.o
obj-$(CONFIG_TTY_PRINTK)        += ttysize.o
obj-y                         += misc.o
obj-$(CONFIG_ATARI_DSP50K)      += dsp50k.o
obj-$(CONFIG_VIRTIO_CONSOLE)    += virtio_console.o
obj-$(CONFIG_RAW_DRIVER)        += raw.o
obj-$(CONFIG_SGI_SN5C)          += sn5c.o sn5c_event.o
obj-$(CONFIG_MSM_SMD_PKT)       += msm_smd_pkt.o
obj-$(CONFIG_MSPEC)             += mspec.o
obj-$(CONFIG_MM_TIMER)          += mm_timer.o
obj-$(CONFIG_UV_MM_TIMER)        += uv_mm_timer.o
obj-$(CONFIG_VIOTAPE)           += viotape.o
obj-$(CONFIG_IBM_BSR)           += bsr.o
obj-$(CONFIG_SGI_MBCS)          += mbcbs.o
obj-$(CONFIG_BRIQ_PANEL)         += briq_panel.o
obj-$(CONFIG_BFIN_OTP)           += bfin-otp.o
obj-$(CONFIG_PRINTER)            += lp.o
```

1,1

Top

上图中的这一行 “obj-y

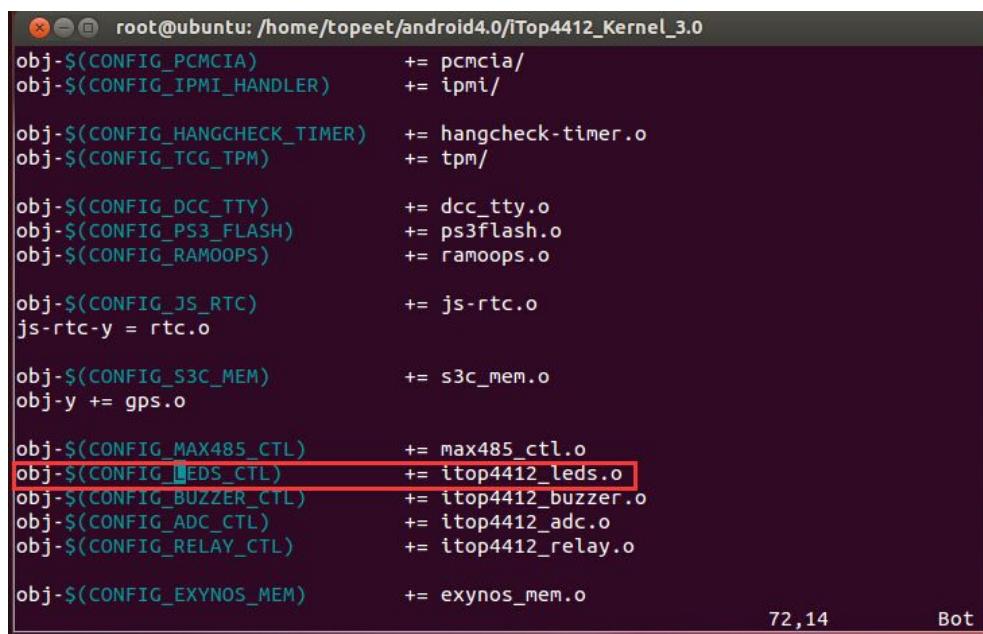
+ = misc.o” , 注意这里需要和内

核自带的代码对齐。

“加等号” 右边的 “misc.o” 表示要编译 Makefile 目录下的 “misc.c” 文件 ,

“加等号” 左边的 “obj-y ” 表示要编译 , 并链接进内核 ( 链接进内核是 linux 源码中自带的工具处理的 , 驱动工程师完全不用关心具体是怎么实现的 )

如下图所示 , 就是 LEDS\_CTL 的条件编译 , 也就是需要在 Kconfig 中定义 , 在 menuconfig 中配置之后 , 编译器运行的时候找到对应的宏变量 LEDS\_CTL 之后才会编译。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
obj-$(CONFIG_PCMCIA)      += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
obj-$(CONFIG_HANGCHECK_TIMER) += hangcheck-timer.o
obj-$(CONFIG_TCG_TPM)      += tpm/
obj-$(CONFIG_DCC_TTY)      += dcc_tty.o
obj-$(CONFIG_PS3_FLASH)    += ps3flash.o
obj-$(CONFIG_RAMOOPS)      += ramoops.o
obj-$(CONFIG_JS_RTC)       += js rtc.o
js-rtc-y = rtc.o
obj-$(CONFIG_S3C_MEM)      += s3c_mem.o
obj-y += gps.o
obj-$(CONFIG_MAX485_CTL)   += max485_ctl.o
obj-$(CONFIG_LEDS_CTL)     += itop4412_leds.o
obj-$(CONFIG_BUZZER_CTL)   += itop4412_buzzer.o
obj-$(CONFIG_ADC_CTL)      += itop4412_adc.o
obj-$(CONFIG_RELAY_CTL)    += itop4412_relay.o
obj-$(CONFIG_EXYNOS_MEM)   += exynos_mem.o
```

上图比较好理解 , 因为一直是以这个 LEDS\_CTL 为例来讲解的。这种方式在写驱动的时候会经常使用到。

最后介绍一下目录层次的迭代，如下图所示，使用命令“vim drivers/Makefile”打开“drivers/char”上一层目录的 Makefile 文件。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
bfin-otp.c      ipmi          msm_smd_pkt.c   scx200_gpio.c
briq_panel.c    itop4412_adc.c  mspec.c        snsc.c
bsr.c           itop4412_adc.o  mwave          snsc_event.c
built-in.o       itop4412_buzzer.c nsc_gpio.c   snsc.h
dcc_tty.c        itop4412_leds.c  nvram.c       sonypi.c
ds1302.c         itop4412_relay.c nwbutton.c   tb0219.c
ds1620.c         itop4412_relay.o nwbutton.h   tlclk.c
dsp56k.c         Kconfig        nwflash.c    toshiba.c
dtlk.c          lp.c          pc8736x_gpio.c tpm
efirtc.c         Makefile      pcmcia        ttyprintk.c
exynos_mem.c    max485_ctl.c   ppdev.c      uv_mmtimer.c
exynos_mem.o    max485_ctl.o   ps3flash.c   viotape.c
generic_nvram.c mbcs.c       ramoops.c   virtio_console.c
genrtc.c         mbcs.h       random.c    xilinx_hwicap
gps.c            mem.c        random.o    raw.c
gps.h            mem.o       rtc.c
gps.o            misc.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/char/Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/char/Makefile
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/Makefile
```

搜索关键词“char”，因为前面介绍到的字符变量在“char”目录下，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
obj-$(CONFIG_RAPIDIO)      += rapidio/
obj-y                      += video/
obj-y                      += idle/
obj-$(CONFIG_ACPI)         += acpi/
obj-$(CONFIG_SFI)          += sfi/
# PnP must come after ACPI since it will eventually need to check if acpi
# was used and do nothing if so
obj-$(CONFIG_PNP)          += pnp/
obj-$(CONFIG_ARM_AMBA)     += amba/
# Many drivers will want to use DMA so this has to be made available
# really early.
obj-$(CONFIG_DMA_ENGINE)   += dma/
obj-$(CONFIG_VIRTIO)        += virtio/
obj-$(CONFIG_XEN)          += xen/
# regulators early, since some subsystems rely on them to initialize
obj-$(CONFIG_REGULATOR)    += regulator/
# tty/ comes before char/ so that the VT console is the boot-time
# default.
obj-y                      += ttv/
obj-y                      += char/
/char
```

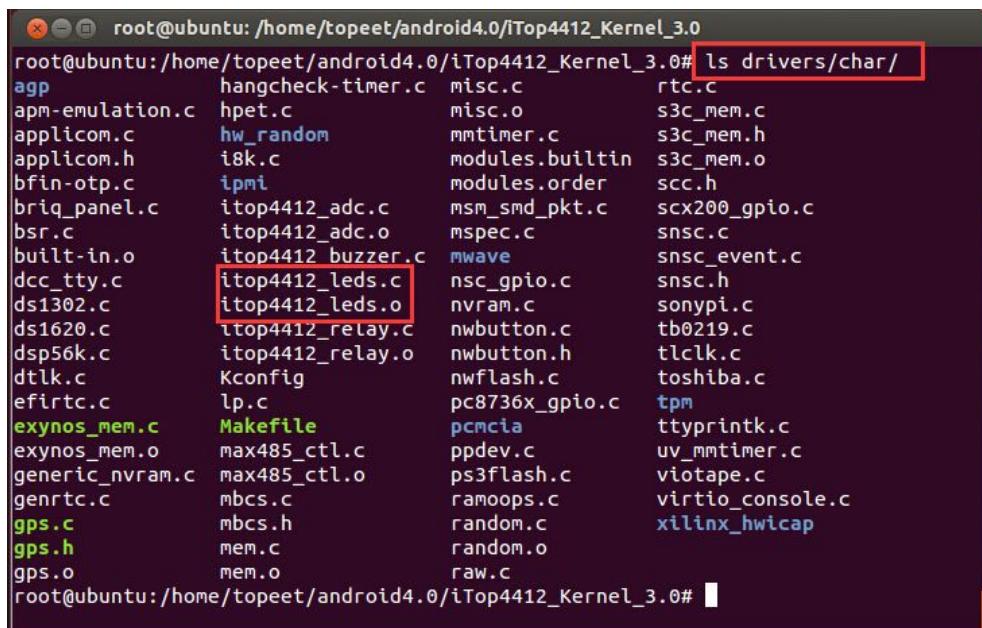
上图中的语法也是很简单，只不过“加等号”右边有文件变为了文件夹。这里表示强制编译当前目录“/drivers”的下一级目录“/char”。在执行编译命令执行到这一句的时候，就会先跳转到“/char”目录下的“Makefile”文件。

## 4.6 Makefile 测试

本小节给大家做个简单的小实验，通过配置 menuconfig 中的 LEDS，来将内核编译进内核或者不编译进内核。

### 4.6.1 将 LEDS 驱动不编译进内核

如下图所示，在给大家提供的源码中，解压之后默认状态就有将“itop4412\_leds.c”编译进内核的中间文件“itop4412\_leds.o”，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  misc.c      rtc.c
apm-emulation.c hpet.c        misc.o      s3c_mem.c
applicom.c    hw_random       mmtimer.c   s3c_mem.h
applicom.h    i8k.c         modules.builtin s3c_mem.o
bfin-otp.c    ipmi          modules.order  scc.h
briq_panel.c itop4412_adc.c msm_smd_pkt.c scx200_gpio.c
bsr.c         itop4412_adc.o mspec.c     snsc.c
built-in.o    itop4412_buzzer.c mwave       snsc_event.c
dcc_tty.c    itop4412_leds.c nsc_gpio.c  snsc.h
ds1302.c     itop4412_leds.o nvram.c    sonypi.c
ds1620.c     itop4412_relay.c nwbutton.c tb0219.c
dsp56k.c     itop4412_relay.o nwbutton.h tlclk.c
dtlk.c       Kconfig        nwflash.c   toshiba.c
efirtc.c     lp.c          pc8736x_gpio.c tpm
exynos_mem.c Makefile       pcmcia     ttyprintk.c
exynos_mem.o max485_ctl.c  ppdev.c    uv_mmtimer.c
generic_nvram.c max485_ctl.o ps3flash.c viotape.c
genrtc.c     mbcs.c        ramoops.c virtio_console.c
gps.c        mbcs.h       random.c   xilinx_hwicap
gps.h        mem.c        random.o
gps.o        mem.o        raw.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

如下图所示，使用命令“rm -rf drivers/char/itop4412\_leds.o”删除掉 LEDS 驱动的中间文件“itop4412\_leds.o”。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  misc.c      rtc.c
apm-emulation.c hpet.c        misc.o      s3c_mem.c
applicom.c    hw_random       mmtimer.c   s3c_mem.h
applicom.h    i8k.c         modules.builtin s3c_mem.o
bfin-otp.c    ipmi          modules.order scc.h
briq_panel.c  itop4412_adc.c  msm_smd_pkt.c scx200_gpio.c
bsr.c         itop4412_adc.o  mspec.c     snsc.c
built-in.o    itop4412_buzzer.c mwave       snsc_event.c
dcc_tty.c     itop4412_leds.c nsc_gpio.c  sonypi.c
ds1302.c      itop4412_leds.o nvram.c    tb0219.c
ds1620.c      itop4412_relay.c nwbutton.c tlclk.c
dsp56k.c      itop4412_relay.o nwbutton.h toshiba.c
dtlk.c        Kconfig        nwflash.c   tpm
efirtc.c      lp.c          pc8736x_gpio.c ttyprintk.c
exynos_mem.c  Makefile      ppdev.c     uv_mmtimer.c
exynos_mem.o  max485_ctl.c  ps3flash.c viotape.c
generic_nvram.c max485_ctl.o mbcsv.c    virtio_console.c
genrtc.c      mbcsv.c      random.c   xilinx_hwicap
gps.c         mbcsv.h      random.o
gps.h         mem.c        raw.c
gps.o         mem.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# rm -rf drivers/char/itop4412_leds.o
```

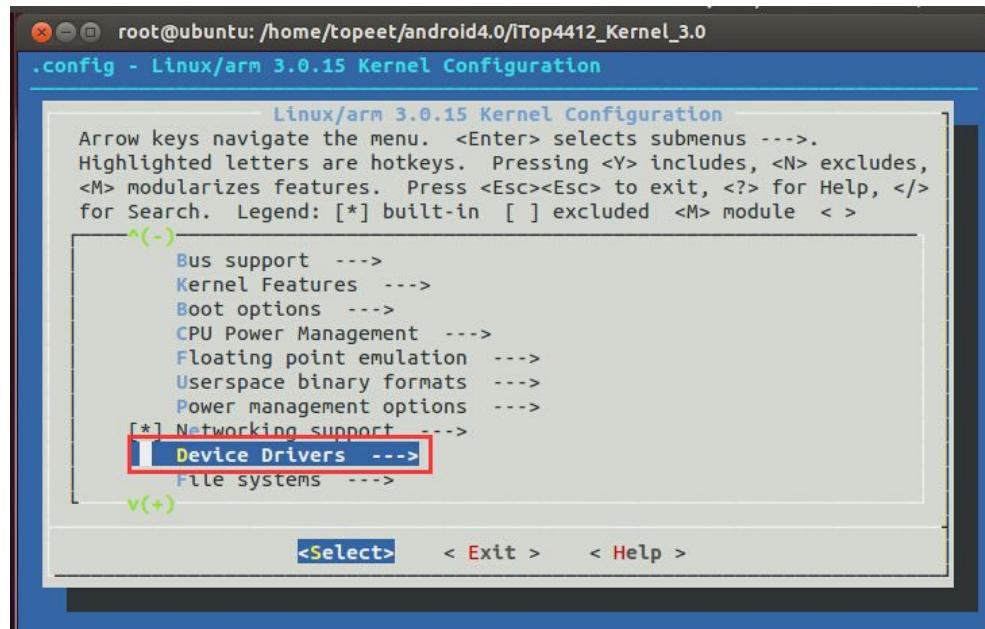
如下图所示，使用命令 ‘ls drivers/char/’ 查看一下，发现已经没有了 ‘itop4412\_leds.o’ 文件。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
p4412_leds.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  misc.o      s3c_mem.c
apm-emulation.c hpet.c        mm timer.c  s3c_mem.h
applicom.c    hw_random       modules.builtin s3c_mem.o
applicom.h    i8k.c         modules.order scc.h
bfin-otp.c    ipmi          modules.order scc.h
briq_panel.c  itop4412_adc.c  msm_smd_pkt.c scx200_gpio.c
bsr.c         itop4412_adc.o  mspec.c     snsc.c
built-in.o    itop4412_buzzer.c mwave       snsc_event.c
dcc_tty.c     itop4412_leds.c nsc_gpio.c  sonypi.c
ds1302.c      itop4412_relay.c nwbutton.c tb0219.c
ds1620.c      itop4412_relay.o nwbutton.h tlclk.c
dsp56k.c      Kconfig        nwflash.c   toshiba.c
dtlk.c        lp.c          pc8736x_gpio.c tpm
efirtc.c      Makefile      ppdev.c     uv_mm timer.c
exynos_mem.c  max485_ctl.c  ps3flash.c viotape.c
exynos_mem.o  max485_ctl.o  mbcsv.c    virtio_console.c
genrtc.c      mbcsv.c      random.c   xilinx_hwicap
gps.c         mem.c        random.o
gps.h         mem.o
gps.o         misc.c      rtc.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

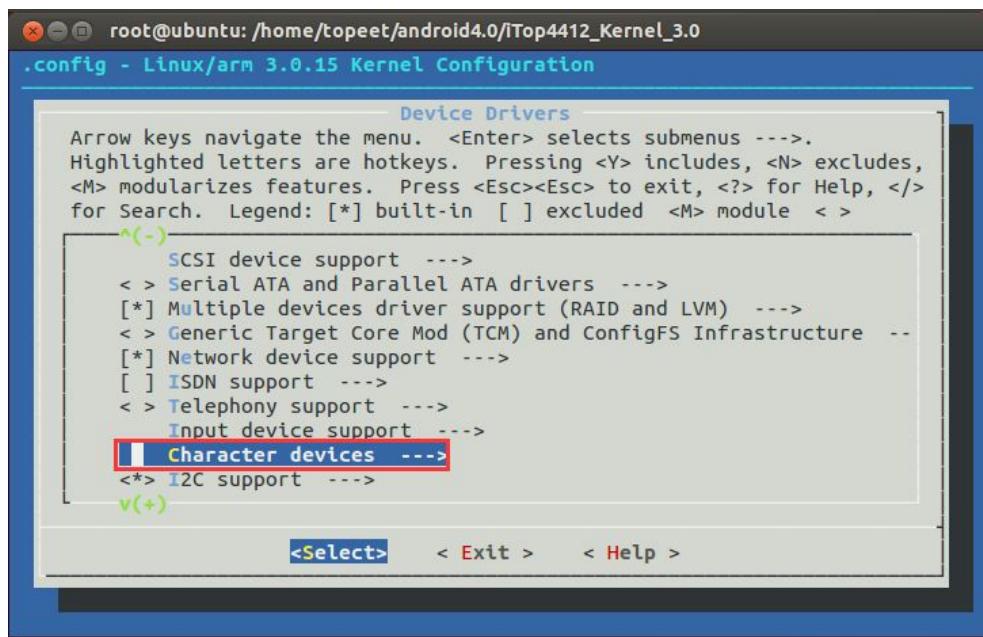
然后使用命令 “make menuconfig” ，打开配置工具，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
p4412_leds.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  misc.o      s3c_mem.c
apm-emulation.c hpet.c        mm timer.c  s3c_mem.h
applicom.c   hw_random       modules.builtin s3c_mem.o
applicom.h    i8k.c         modules.order   scc.h
bfin-otp.c   ipmi          msm_smd_pkt.c scx200_gpio.c
briq_panel.c itop4412_adc.c  mspec.c     sns c.
bsr.c        itop4412_adc.o  mwave        sns c_event.c
built-in.o    itop4412_buzzer.c nsc_gpio.c  sns c.h
dcc_tty.c    itop4412_leds.c nvram.c    sonypi.c
ds1302.c    itop4412_relay.c nwbutton.c tb0219.c
ds1620.c    itop4412_relay.o nwbutton.h tclk.c
dsp56k.c    Kconfig        nwflash.c   toshiba.c
dtlk.c      lp.c          pc8736x_gpio.c tpm
efirtc.c    Makefile       pcmcia      ttyprintk.c
exynos_mem.c max485_ctl.c  ppdev.c    uv_mm timer.c
exynos_mem.o max485_ctl.o  ps3flash.c viotape.c
generic_nvram.c mbcs.c     ramoops.c virtio_console.c
genrtc.c    mbcs.h        random.c   xilinx_hwicap
gps.c       mem.c         random.o
gps.h       mem.o         raw.c
gps.o       misc.c        rtc.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make menuconfig
```

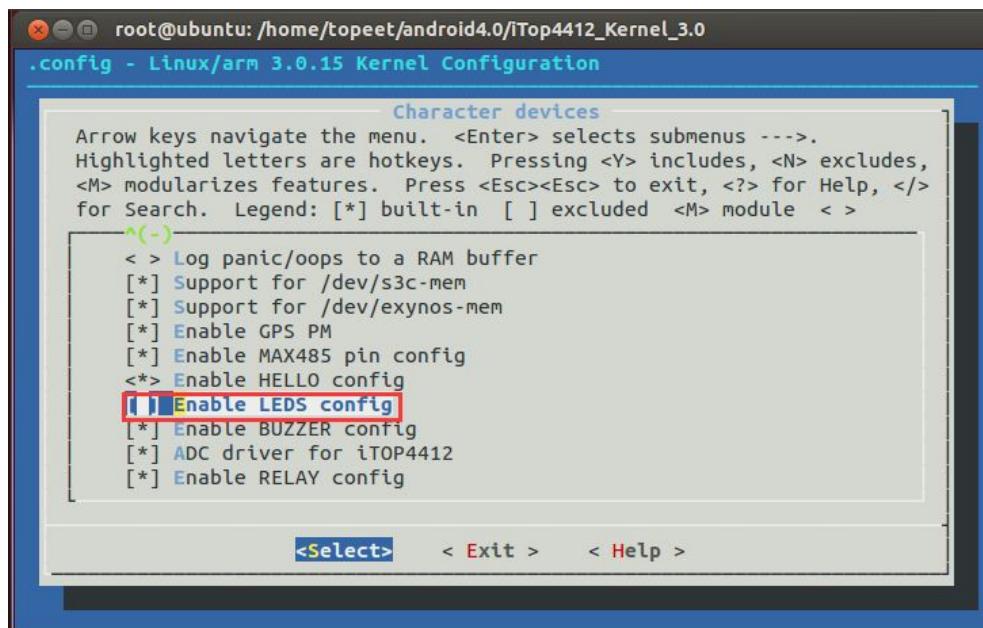
打开之后，如下图所示。找到“Device Drivers --->”选项。



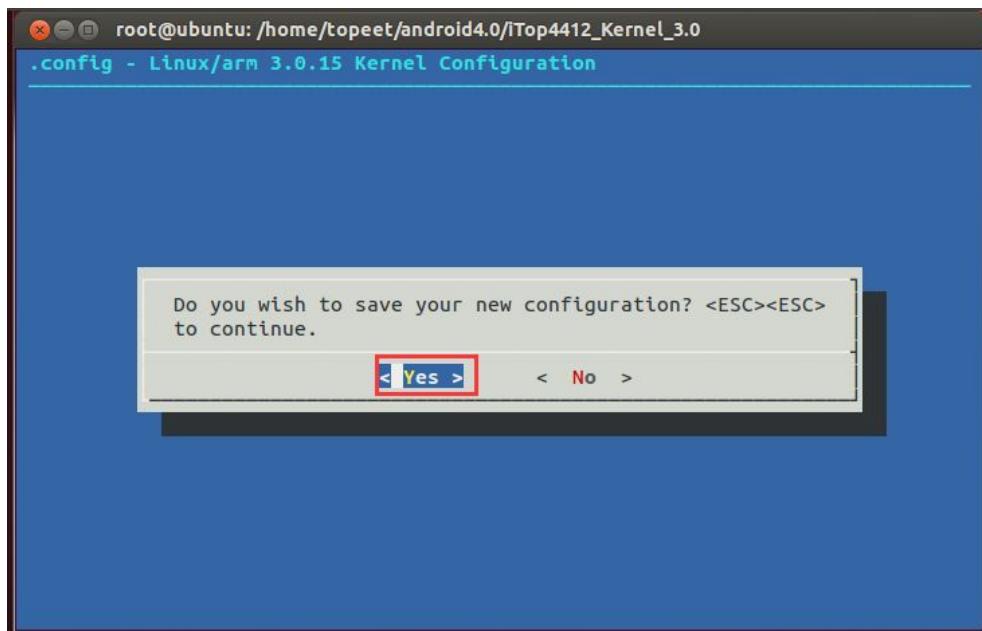
如下图所示，进入到“Device Drivers --->”选项，找到选项“Character devices --->”。



如下图所示，找到选项“Enable LEDS config”，将 LEDS 的配置关闭



退出，保存，生成新的“.config”文件。



保存退出后，打开 “.config” 文件，可以发现 “CONFIG\_LEDS\_CTL” 没有配置。

A screenshot of a terminal window titled ".config - Linux/arm 3.0.15 Kernel Configuration". The window displays the contents of the .config file. One specific line, "# CONFIG\_LEDS\_CTL is not set", is highlighted with a red box. Other visible lines include "# CONFIG\_HW\_RANDOM\_TIMERIOMEM is not set", "# CONFIG\_R3964 is not set", "# CONFIG\_RAW\_DRIVER is not set", "# CONFIG\_TCG\_TPM is not set", "# CONFIG\_DCC\_TTY is not set", "# CONFIG\_RAMOOPS is not set", "CONFIG\_S3C\_MEM=y", "CONFIG\_EXYNOS\_MEM=y", "CONFIG\_GPS\_PM=y", "CONFIG\_MAX485\_CTL=y", "CONFIG\_BUZZER\_CTL=y", "CONFIG\_ADC\_CTL=y", "CONFIG\_RELAY\_CTL=y", "CONFIG\_I2C=y", "CONFIG\_I2C\_BOARDINFO=y", "CONFIG\_I2C\_COMPAT=y", "CONFIG\_I2C\_CHARDEV=y", "# CONFIG\_I2C\_MUX is not set", "# CONFIG\_I2C\_HELPER\_AUTO is not set", and "# CONFIG\_I2C\_SMBUS is not set". The bottom right corner of the terminal window shows the coordinates "1507,10" and the battery level "54%".

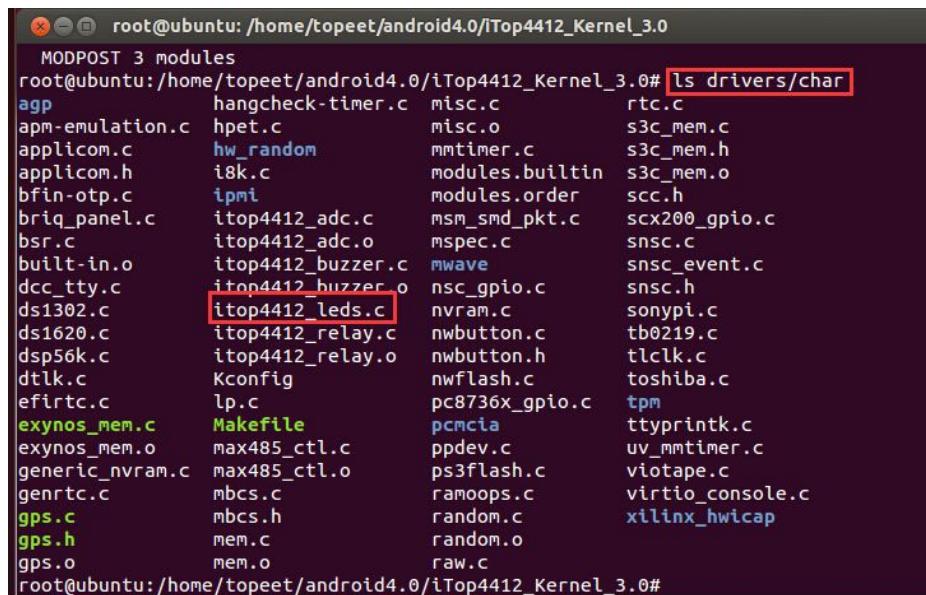
关闭 “.config” 文件，然后执行编译内核的命令 “make” ，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
dcc_tty.c      itop4412_leds.c    nvram.c        sonypi.c
ds1302.c       itop4412_relay.c   nwbutton.c    tb0219.c
ds1620.c       itop4412_relay.o   nwbutton.h    tclk.c
dsp56k.c       Kconfig          nwflash.c     toshiba.c
dtlk.c         lp.c             pc8736x_gpio.c tpm
efirtc.c       Makefile         pcmcia        ttyprintk.c
exynos_mem.c   max485_ctl.c    ppdev.c       uv_mmtimer.c
exynos_mem.o   max485_ctl.o    ps3flash.c   viotape.c
generic_nvram.c mbcsv.c       ramoops.c    virtio_console.c
genrtc.c       mbcsv.h        random.c     xilinx_hwicap
gps.c          mem.c           random.o
gps.h          mem.o           raw.c
gps.o          misc.c         rtc.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make menuconfig
scripts/kconfig/mconf Kconfig
#
# configuration written to .config
#
*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make
```

如下图所示，编译完成。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
UPD      include/generated/compile.h
CC       init/version.o
LD       init/built-in.o
LD       .tmp_vmlinux1
KSYM    .tmp_kallsyms1.S
AS       .tmp_kallsyms1.o
LD       .tmp_vmlinux2
KSYM    .tmp_kallsyms2.S
AS       .tmp_kallsyms2.o
LD       vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
GZIP    arch/arm/boot/compressed/piggy.gzip
AS      arch/arm/boot/compressed/piggy.gzip.o
SHIPPED arch/arm/boot/compressed/lib1funcs.S
AS      arch/arm/boot/compressed/lib1funcs.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
Building modules, stage 2.
MODPOST 3 modules
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

如下图所示，使用命令 “ls drivers/char” 查看 LEDS 驱动所在目录，发现并没有产生了中间文件。

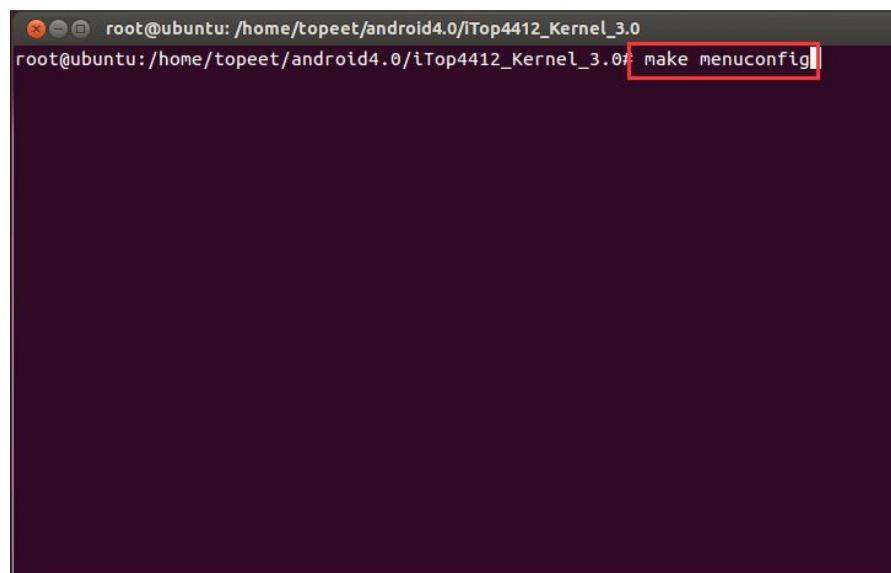


```
MODPOST 3 modules
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char
agp                  hangcheck-timer.c    misc.c          rtc.c
apm-emulation.c      hpet.c             misc.o          s3c_mem.c
applicom.c           hw_random         mmtimer.c     s3c_mem.h
applicom.h            i8k.c              modules.builtin s3c_mem.o
bfin-otp.c           ipmi               modules.order   scc.h
briq_panel.c          itop4412_adc.c    msm_smd_pkt.c scx200_gpio.c
bsr.c                itop4412_adc.o    mspec.c        sns.c
built-in.o            itop4412_buzzer.c mwave          sns_event.c
dcc_tty.c             itop4412_buzzer.o nsc_gpio.c    sns.h
ds1302.c              itop4412_leds.c   nvram.c       sonypi.c
ds1620.c              itop4412_relay.c  nwbutton.c   tb0219.c
dsp56k.c              itop4412_relay.o  nwbutton.h   tlclk.c
dtlk.c                Kconfig           nwflash.c    toshiba.c
efirtc.c              lp.c               pc8736x_gpio.c tpm
exynos_mem.c          Makefile          pcmcia        ttyprintk.c
exynos_mem.o          max485_ctl.c    ppdev.c       uv_mmtimer.c
generic_nvram.c       max485_ctl.o    ps3flash.c   viotape.c
genrtc.c              mbcs.c           ramoops.c    virtio_console.c
gps.c                mbcs.h           random.c    xilinx_hwicap
gps.h                mem.c             random.o    raw.c
gps.o                mem.o           raw.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

最后也可以将生成的文件“zImage”文件下载到开发板，会发现“/dev”目录中没有产生LEDS 驱动的设备节点“leds”。

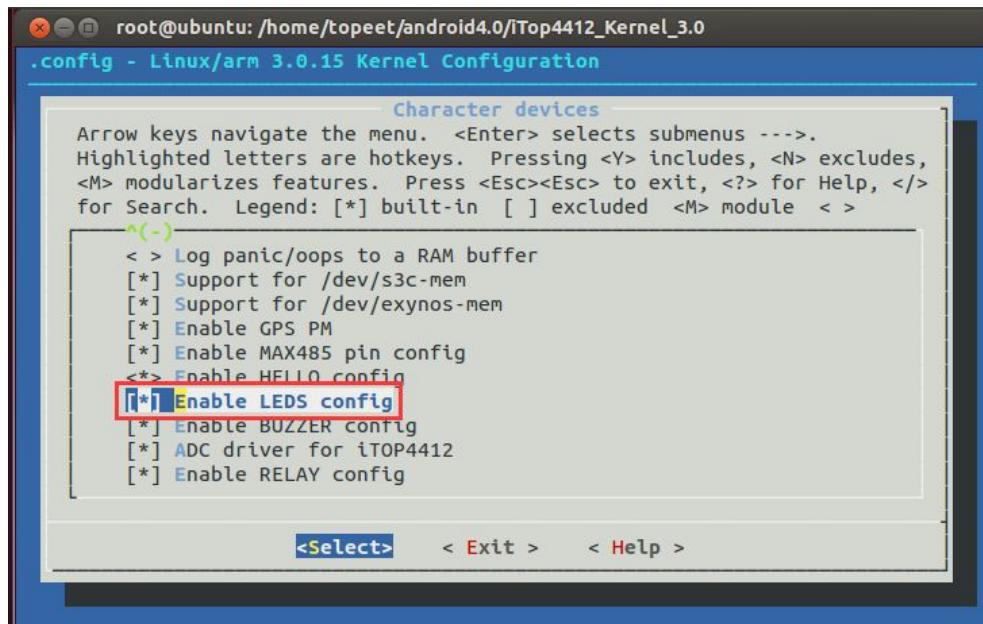
#### 4.6.2 将 LEDS 驱动编译进内核

接上一小节，如下图所示，使用命令“make menuconfig”，打开 menuconfig 配置工具。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# make menuconfig
```

进入“Device Drivers --->”选项，然后进入“Character devices --->”选项，然后如下图所示，配置上“Enable LEDS config”。



保存退出。打开“.config”文件，可以发现“CONFIG\_LEDS\_CTL=y”已经配置。

```
# CONFIG_HW_RANDOM_TIMERIOMEM is not set
# CONFIG_R3964 is not set
# CONFIG_RAW_DRIVER is not set
# CONFIG_TCG_TPM is not set
# CONFIG_DCC_TTY is not set
# CONFIG_RAMOOPS is not set
CONFIG_S3C_MEM=y
CONFIG_EXYNOS_MEM=y
CONFIG_GPS_PM=y
CONFIG_MAX485_CTL=y
CONFIG_HELLO_CTL=v
CONFIG_LEDCTL=y
CONFIG_BUZZER_CTL=y
CONFIG_ADC_CTL=y
CONFIG_RELAY_CTL=y
CONFIG_I2C=y
CONFIG_I2C_BOARDINFO=y
CONFIG_I2C_COMPAT=y
CONFIG_I2C_CHARDEV=y
# CONFIG_I2C_MUX is not set
# CONFIG_I2C_HELPER_AUTO is not set
# CONFIG_I2C_SMBUS is not set
```

执行编译命令“make”，如下图所示，编译完成之后，使用命令“ls drivers/char/”可以看到“itop4412\_leds.o”已经生成。

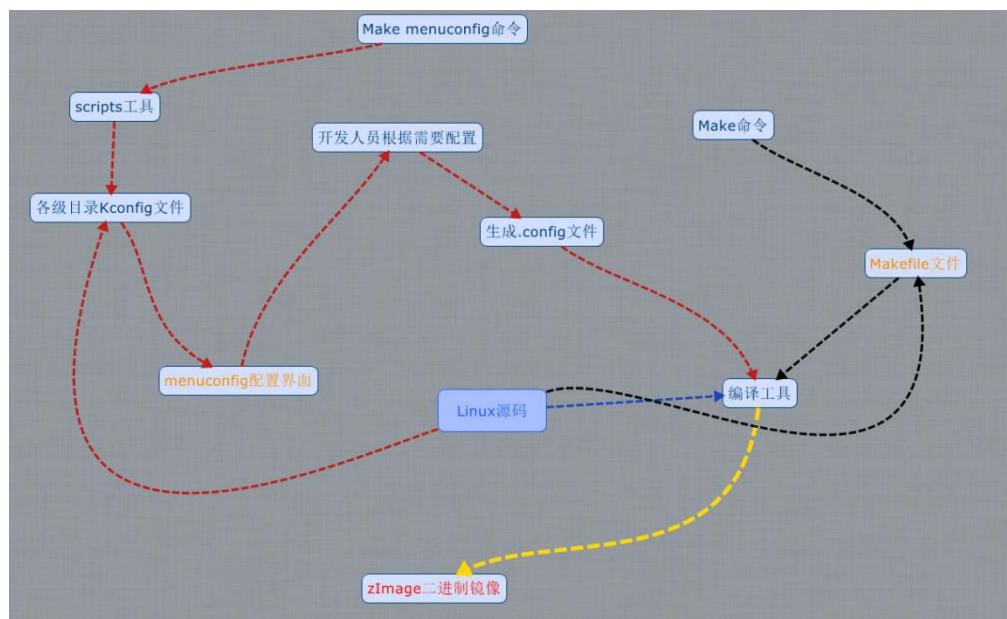
```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
MODPOST 3 modules
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c    mem.o        raw.c
apm-emulation.c hpet.c          misc.c       rtc.c
applicom.c    hw_random         misc.o       s3c_mem.c
applicom.h    i8k.c            mmtimer.c   s3c_mem.h
bfin-otp.c    ipmi             modules.builtin s3c_mem.o
briq_panel.c  itop4412_adc.c   modules.order scc.h
bsr.c         itop4412_adc.o    msm_smd_pkt.c scx200_gpio.c
built-in.o    itop4412_buzzer.c mspec.c     snsc.c
dcc_tty.c    itop4412_buzzer.o mwave        snsc_event.c
ds1302.c     itop4412_leds.c   nsc_gpio.c  snsc.h
ds1620.c     itop4412_leds.o   nvram.c     sonypi.c
dsp56k.c      itop4412_relay.c nwbutton.c  tb0219.c
dtlk.c        itop4412_relay.o nwbutton.h  tlcclk.c
efirtc.c      Kconfig           nwflash.c   toshiba.c
exynos_mem.c  Makefile          pc8736x_gpio.c tpm
exynos_mem.o  Makefile          pcmcia      ttyprintk.c
generic_nvram.c max485_ctl.c   ppdev.c    uv_mmtimer.c
genrtc.c      max485_ctl.o    ps3flash.c  viotape.c
gps.c         mbcs.c           ramoops.c  virtio_console.c
gps.h         mbcs.h           random.c  xilinx_hwicap
gps.o         mem.c            random.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

最后也可以将生成的文件“zImage”文件下载到开发板，会发现“/dev”目录中产生LEDs驱动的设备节点“leds”，LED灯的控制又恢复正常了。

## 4.7 编译流程图解

现在内核编译相关的知识已经全部介绍完，本节做一个小节。

下图已经将内核编译的全部过程包含。



如上图所示。

红色的线条表示配置文件 Kconfig 这一部分，在 Kconfig 中要定义针对具体驱动文件的宏变量。然后使用 menuconfig 工具生成新的“.config”文件。

黑色的线条表示编译文件 Makefile 这一部分，在 Makefile 中针对宏变量编译驱动文件。执行 make 命令之后，调用 “.config” 文件，配合各级目录中的 Makefile 文件编译具体的驱动源代码，将源代码编译成 “.o” 中间文件。

当中间文件全部编译完成之后，编译工具会生成一个非常精炼的 “zImage” 二进制文件。

# 实验 05 总线\_设备\_驱动注册流程详解

## 5.1 本章导读

在 Linux2.6 之后，Linux 设备驱动分为三个实体总线、设备、驱动，平台总线将设备和驱动匹配。在系统注册任意一个驱动的时候，都会寻找对应的设备；当系统注册设备的时候，系统也会寻找对应的驱动进行匹配。

本节实验通过一张框架图，从理论上给大家分析总线设备驱动三者的关系。

### 5.1.1 工具

#### 5.1.1.1 硬件工具

PC 机一台

#### 5.1.1.2 软件工具

虚拟机 Ubuntu 系统

Linux 源码 “iTop4412\_Kernel\_3.0\_xxx”（在光盘目录 “/Android 源码” 文件夹下,xxx 表示日期）

### 5.1.2 预备课程

Linux 常用命令视频教程 “01-烧写、编译以及基础知识视频” → “实验 07-Linux 常用命令”

使用手册 3.3.3 小节 Linux 常用 shell 命令

### 5.1.3 视频资源

本节配套视频为“视频 05\_总线\_设备\_驱动注册流程详解”

## 5.2 学习目标

本章需要学习以下内容：

了解 Linux 总线的概念

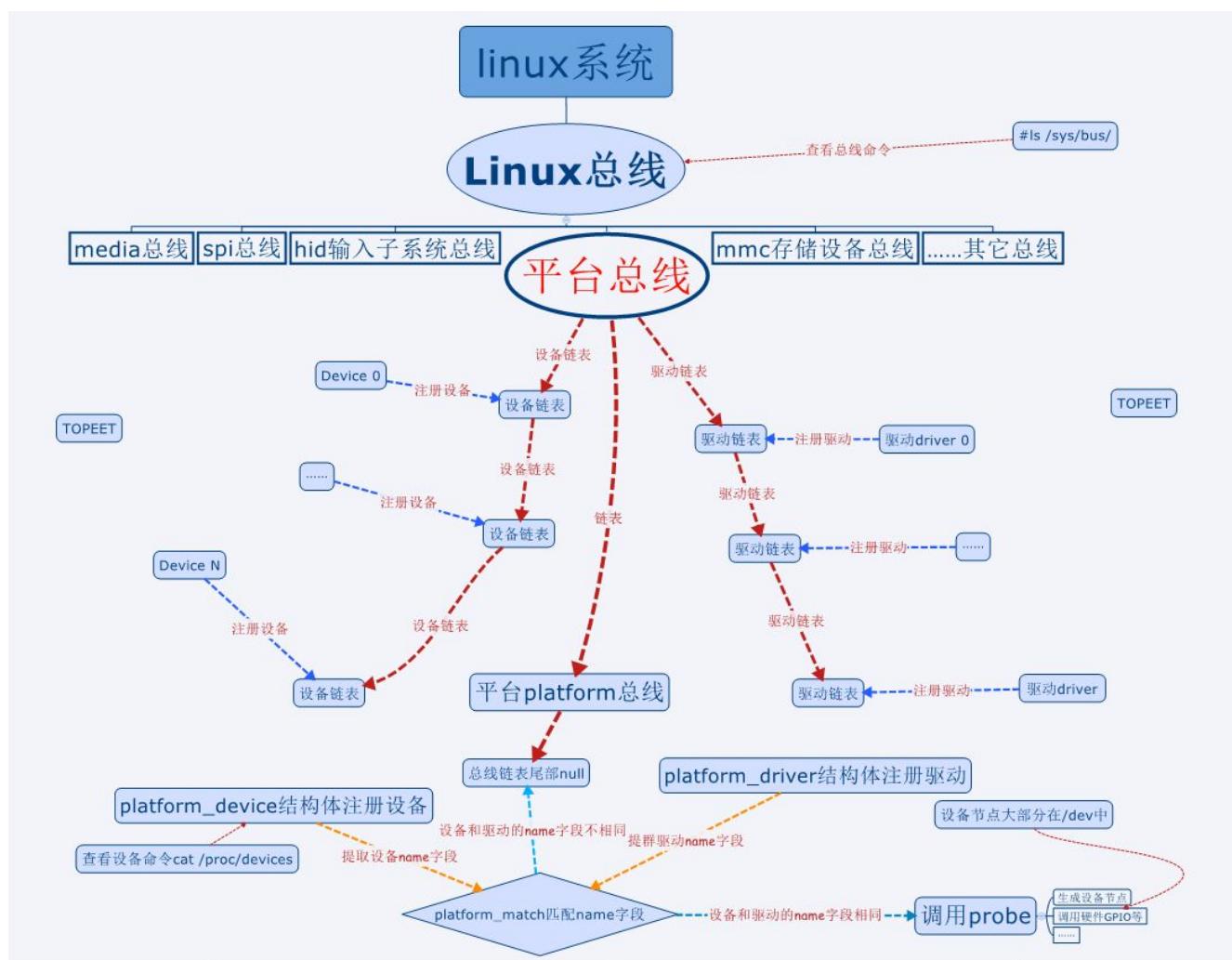
理解平台总线 platform

理解 Linux 设备的概念

掌握 Linux 驱动注册流程

## 5.3 总线、设备、驱动框架图分析

如下图所示，这是总线、设备、驱动的框架图，下面就来分析它们之间的关系。



### 5.3.1 总线和平台总线

这里再次强调一个观点，Linux 系统中大部分内容不需要关心，哪怕一些编译进内核的源代码，绝大部分都不需要去阅读。

在 Linux 系统中，任何一个 Linux 设备和 Linux 驱动都是需要挂载到一种总线中。有一些常规的大家容易理解的总线，例如 media 总线、spi 总线、hid 输入子系统总线、eMMC 存储

设备总线等等。假如说设备本身就是一个总线设备，那么挂载到对应的总线上，那是容易理解的。

在任意一个 soc 系统中，都有一些集成的总线，例如在 4412 中就集成了 i2c，spi、usb 等等。针对这些总线设备，它们注册驱动的时候，都会调用对应的总线设备，一个驱动对应一个设备，这个概念很好理解。

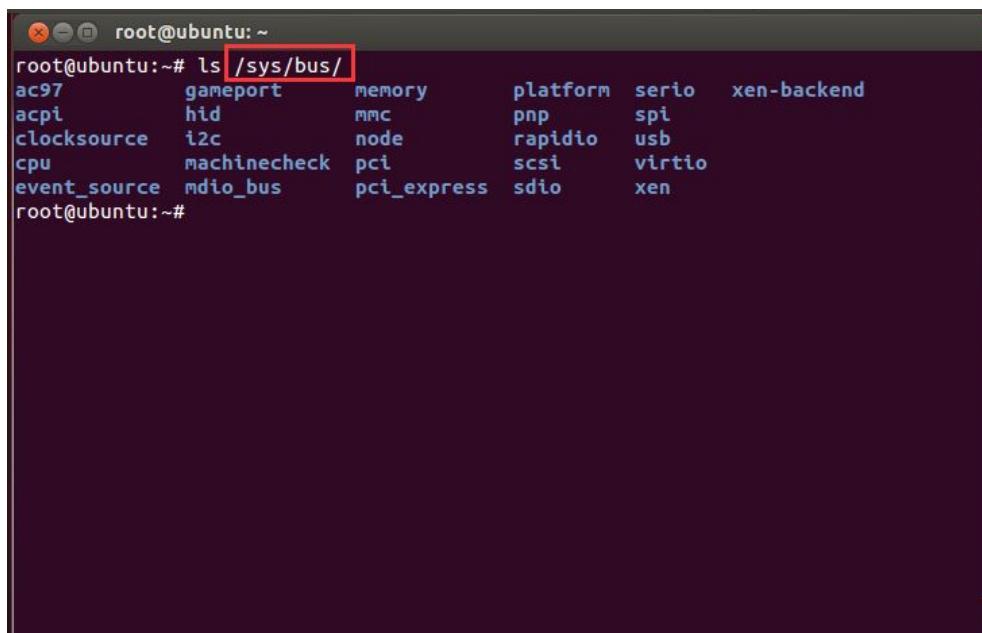
但是还有一些例如 led、蜂鸣器等等一些设备，都不是从字面上理解的总线设备。针对这种情况，Linux 创立了一种虚拟总线，也叫平台总线或者 platform 总线，这个总线也有对应的设备 platform\_device, 对应的驱动叫 platform\_driver.

这里介绍的平台总线，不能够直接和常规的总线对应，只是 Linux 系统提供的一种附加手段，防止 linux 驱动的碎片化，降低 Linux 的使用难度。

另外这里的设备 platform\_device 和驱动 platform\_driver 也不是和常规的字符设备、块设备、网络设备并列的概念，它只是一种附加的手段。

具体的这个虚拟平台怎么使用，是后续实验要逐渐介绍。

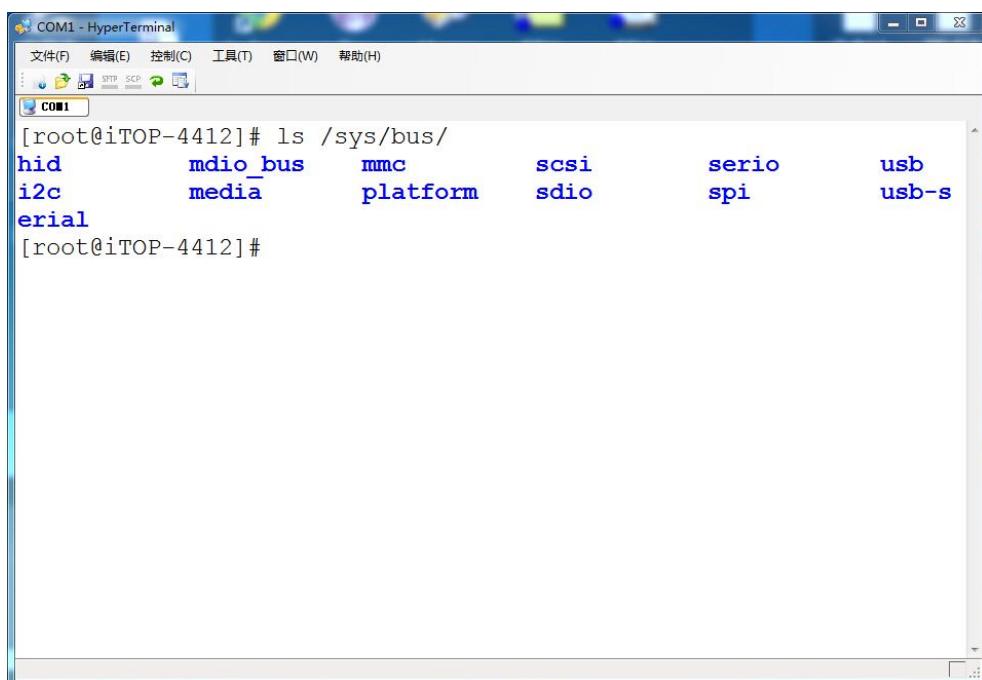
如下图所示，可以通过命令 “ls /sys/bus/” 查看总线，这是在 Ubuntu 系统下使用这个命令，Ubuntu 系统也是属于 Linux 系统，Ubuntu 是属于 Linux 庞大分支中一个拥有图形界面的操作系统。



```
root@ubuntu:~# ls /sys/bus/
ac97      gameport   memory      platform  serio    xen-backend
acpi       hid        mmc        pnp       spi
clocksource i2c       node       rapidio   usb
cpu        machinecheck pci       scsi      virtio
event_source mdio_bus  pci_express sdio     xen
root@ubuntu:~#
```

上图中显示的是 x86 PC 机上的 Ubuntu 总线。

如下图所示，在 iTOP-4412 开发板上运行最小 Linux 系统，使用命令 “ls /sys/bus” ，可以查看开发板带有的总线。其中的 platform 总线是和其它总线在同一个目录下。



```
COM1 - HyperTerminal
文件(F) 编辑(E) 控制(C) 工具(T) 窗口(W) 帮助(H)
... SFTP SCP ...
COM1
[root@iTOP-4412]# ls /sys/bus/
hid      mdio_bus   mmc      scsi      serio      usb
i2c      media      platform  sdio      spi       usb-s
serial
[root@iTOP-4412]#
```

### 5.3.2 Linux 设备

硬件总类繁多，千变万化，一个 USB 接口就可以接无数种键盘、鼠标、存储设备等等。Linux 将设备分为了三大类：字符设备、块设备、网络设备。

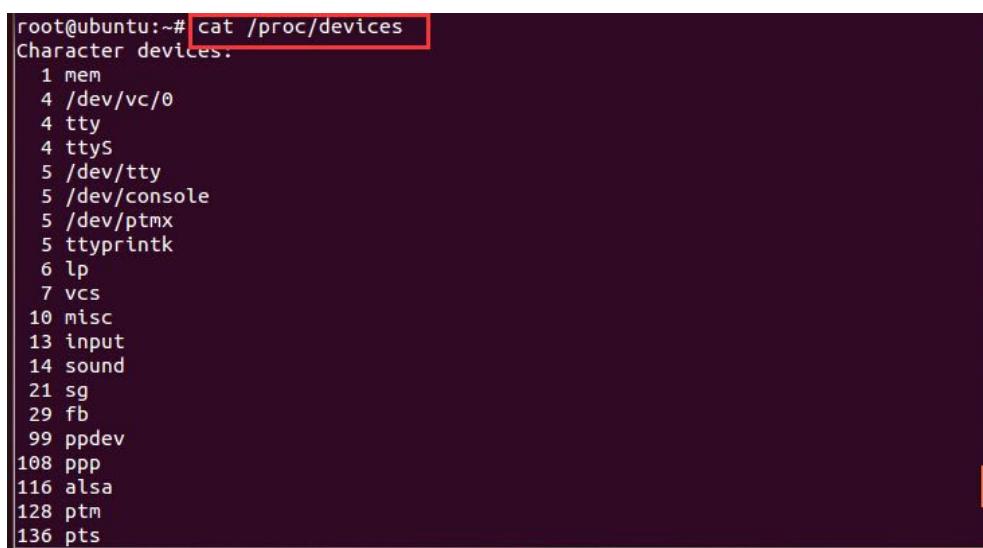
字符设备，字符设备是能够像字节流一样被访问的设备。一般说来对硬件设备 IO 的操作可以归结为字符设备。常见的字符设备有 led、蜂鸣器、串口、键盘等等。

块设备，块设备是通过内存缓冲区访问，可以随机存取的设备，一般性的理解就是存储介质类的设备。常见的字符设备有 U 盘、TF 卡、eMMC、电脑硬盘、光盘等等

网络设备，可以和其它主机交换数据的设备。常见的以太网设备、WIFI、蓝牙等。

虽然它们之间有这种官方的分类，但是也没有严格的界限，只是一个比较模糊的划分。

设备种类繁多，不同的设备对应不同的设备名和设备号，在 Ubuntu 虚拟机中使用命令“cat /proc/devices”，如下图所示，可以看到不同的设备都有了编号。



```
root@ubuntu:~# cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttys
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 5 ttprintk
 6 lp
 7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
```

启动开发板，在超级终端中输入命令“cat /proc/devices”，可以看到不同的设备。

```
[root@iTOP-4412]# cat /proc/devices
Character devices:
 1 mem
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
10 misc
13 input
21 sg
29 fb
81 video4linux
89 i2c
108 ppp
116 alsa
128 ptm
136 pts
150 ...
```

设备号肯定是有限的，一共就只有 256 个主设备号，这里引入了从设备号的概念，理论上就有  $256 \times 256$  个设备号，这种数量级目前是可以接受的。

### 5.3.3 Linux 驱动

驱动程序一般以一个模块初始化函数作为入口，例如实验手册“实验二”中最简单的驱动。在作者看来，驱动的学习分为两大类，一类是需要会写的，主要是字符设备，一类就是移植。

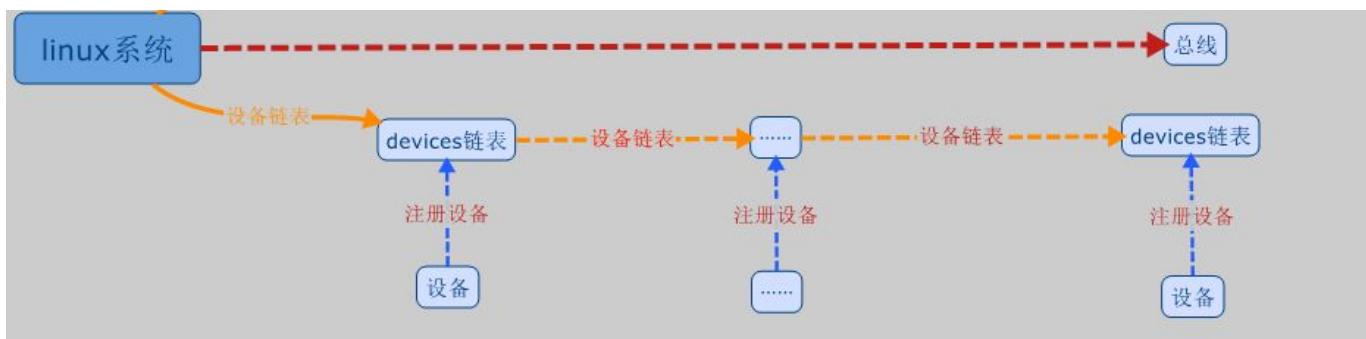
需要会写的就是字符设备，像块设备和网络设备要么是 soc 原厂已经集成，要么是外围器件芯片厂商提供源代码，主要工作在管脚配置和调试。

实验手册主要就是围绕驱动展开的，通过后续的学习大家就可以理解驱动是怎么回事，驱动怎么编写和移植。

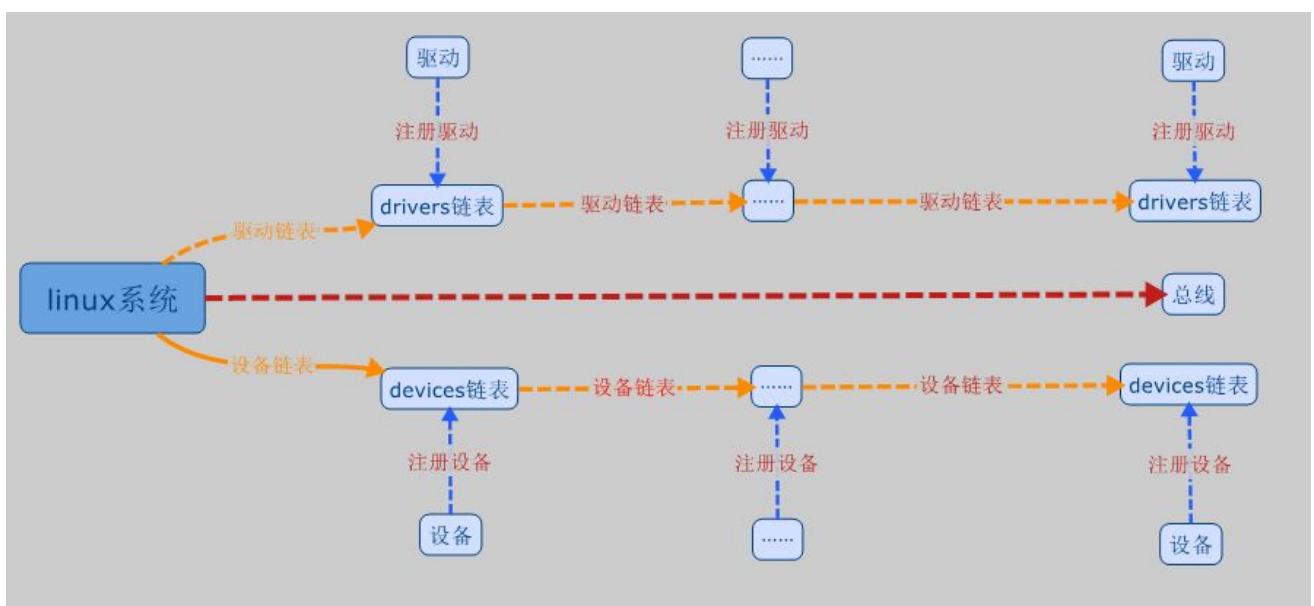
### 5.3.4 Linux 驱动和设备的注册过程

Linux 内核会要求每出现一个设备就要像总线汇报，或者说注册，出现一个驱动，也要向总线汇报，或者叫注册。

在系统初始化的时候，会扫描连接了哪些设备，并为每一个设备建立一个 struct\_device 的变量，然后将设备的变量插入到 devices 链表，如下图所示。



系统初始化任意一个驱动程序的时候，就要准备一个 struct device\_driver 结构的变量，然后将驱动的变量插入到 drivers 链表，如下图所示。



Linux 总线是为了将设备和驱动绑定，方便管理。在系统每注册一个设备的时候，会寻找与之匹配的驱动（后面的热拔插设备会用到这个注册流程）；相反，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

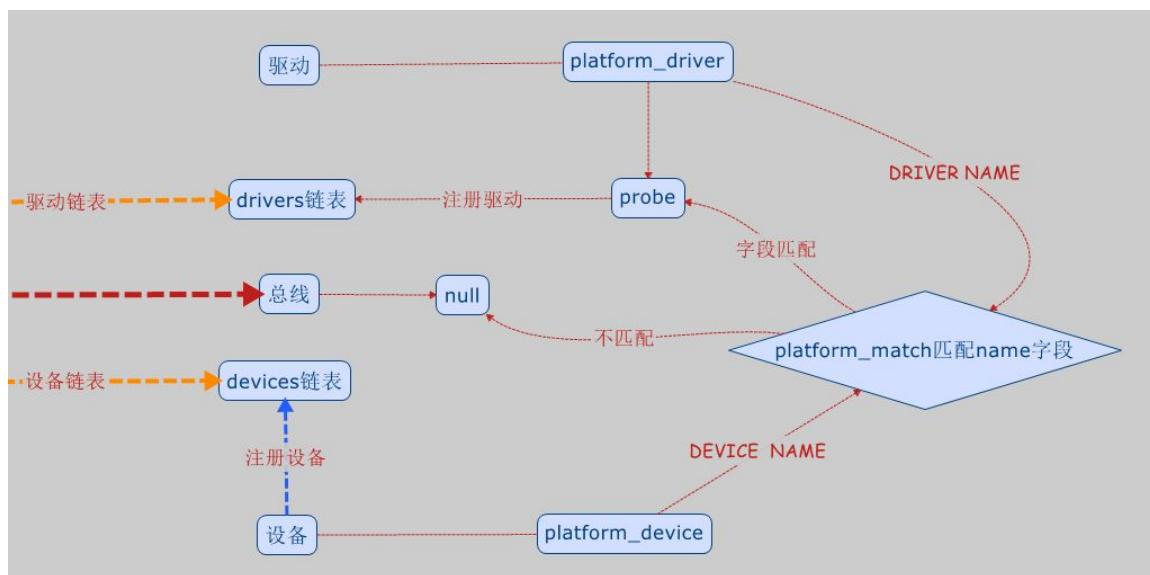
这里只讨论先注册设备再注册驱动的情况。

在注册驱动的时候，系统会通过 `platform_match` 函数匹配设备和驱动。

注册设备的结构体为 `platform_device`，注册驱动的结构体为 `platform_driver`。设备和驱动结构体的成员 `name` 字段，相同则匹配。

如果匹配了则会调用 `platform_driver` 中的 `probe` 函数，注册驱动。

也就是在上图注册驱动的时候要添加一个判断，如下图所示。



大家知道现在很多设备都是支持热拔插的。在 Linux 中，一般情况都是先注册设备，然后再注册驱动，但是有了热拔插设备之后，情况就不一样了。在热拔插设备中，是有了设备 `devices` 接入之后，内核会去 `driver` 链表中寻找驱动。这种情况在后面的实验再讨论。

### 5.3.5 设备节点简介

在 Linux 系统中，有一个概念叫“一切皆文件”，上层应用使用设备节点访问对应的设备。

设备节点一般是放在“/dev”目录下，在开发板下输入命令“ls /dev”，如下图所示。

```
[root@iTOP-4412] # ls /dev/
AGPS           input          ram0          tty3
HED            ion            ram1          tty4
adc            keychord       ram10         ttyGS0
alarm          kmem           ram11         ttyGS1
android_adb   kmsq           ram12         ttyGS2
ashmem          leds           ram13         ttyGS3
bus             log            ram14         ttyS0
buzzer_ctl    loop0          ram15         ttyS1
console        loop1          ram2          ttyS2
cpu_dma_latency loop2          ram3          ttyS3
exynos-mem    loop3          ram4          ttySAC0
fb0             loop4          ram5          ttySAC1
fb1             loop5          ram6          ttySAC2
fb10            loop6          ram7          ttySAC3
fb11            loop7          ram8          uinput
fb2             mali            ram9          ump
fb3             mapper          random        urandom
fb4             max485_ctl_pin rc522        usb_accessory
fb5             mem             root          usbdev1.1
fb6             mmcblk0        rtc0          usbdev1.2
fb7             mmcblk0p1       rtc1          usbdev1.3
fb8             mmcblk0p2       s3c-mem      usbdev1.4
fb9             mmcblk0p3       s3c-mfc      video0
fimg2d          mmcblk0p4       sda           video1
full            mtp_usb         sda1          video11
fuse            network_latency sg0           video12
gps             network_throughput shm          video16
i2c-0           null            snd           video2
i2c-1           pmem            srp           video20
i2c-3           pmem_gpu1      srp_ctrl     video3
i2c-4           ppp             tty           watchdog
i2c-5           ptmx           tty1          xt_qtaguid
i2c-7           pts             tty2          zero
[root@iTOP-4412] #
```

上层应用有一套标准的接口文件和函数用来和底层通信。

Linux 将所有对设备的操作全部都抽象为对文件的操作，常用的文件操作函数有 open、close、write、read、write 等。

后面会详细介绍这些函数如何使用，即使是驱动工程师，也要使用这些函数编写简单的驱动测试函数，这些函数的使用必须掌握的。

# 实验 06 设备注册

## 6.1 本章导读

在 Linux2.6 之后，引入了平台总线的概念，所以注册设备变得更加简单。为了使大家更快的入门，先介绍如何在平台总线中注册设备。

### 6.1.1 工具

#### 6.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 6.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) Ubuntu 系统下解压生成的 Linux 源码

### 6.1.2 预备课程

内核的编译以及烧写等相关课程

Menuconfig、Kconfig 的使用等相关课程

### 6.1.3 视频资源

本节配套视频为“视频 06\_设备注册”

## 6.2 学习目标

本章需要学习以下内容：

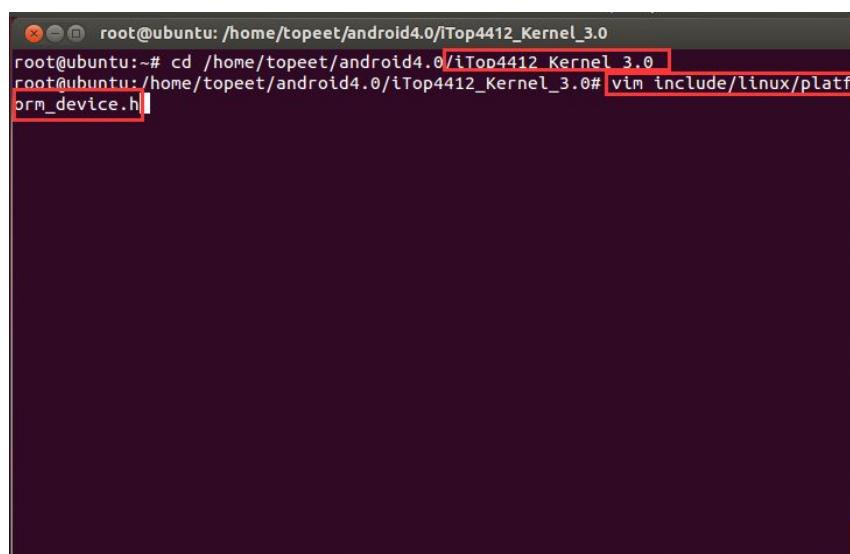
掌握在平台文件中注册设备的方法

## 6.3 在虚拟总线上注册设备

在实验 5 中介绍了注册设备的结构体“platform\_device”，并没有介绍“platform\_device”结构体的内部结构。

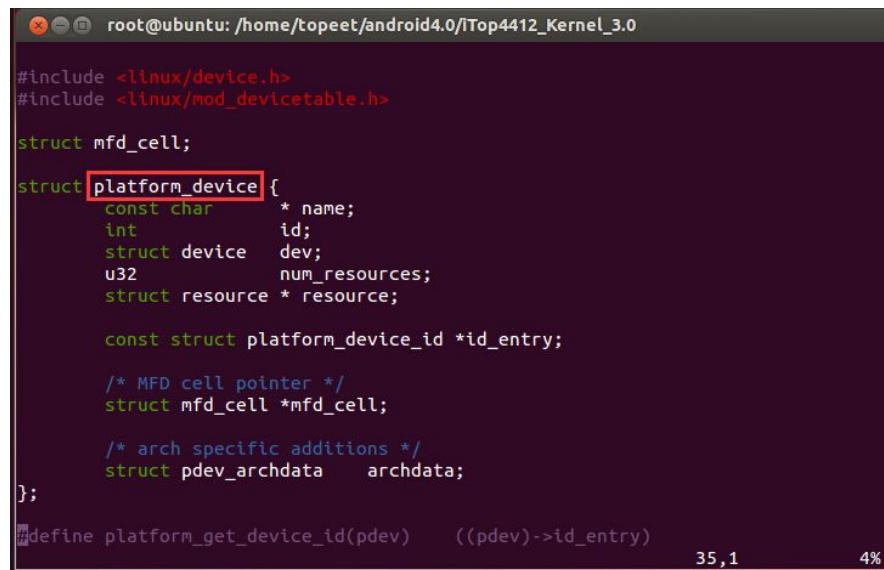
如下图所示，进入解压之后的内核文件夹“iT0p4412\_Kernel\_3.0”，使用命令

“vim include/linux/platform\_device.h” 打开“platform\_device”所在文件。



A screenshot of a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The window shows a command-line interface with the following text:  
root@ubuntu:~# cd /home/topeet/android4.0/iTop4412\_Kernel\_3.0  
root@ubuntu:/home/topeet/android4.0/iTop4412\_Kernel\_3.0# vim include/linux/platform\_device.h

如下图所示，就在第一页中，就可以看到结构体“platform\_device”。



```
#include <linux/device.h>
#include <linux/mod_devicetable.h>

struct mfd_cell;

struct platform_device {
    const char * name;
    int id;
    struct device dev;
    u32 num_resources;
    struct resource * resource;

    const struct platform_device_id *id_entry;

    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;

    /* arch specific additions */
    struct pdev_archdata archdata;
};

#define platform_get_device_id(pdev) ((pdev)->id_entry)
```

The screenshot shows a terminal window with the command "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". It displays the kernel source code for the "platform\_device" structure. The structure definition is shown, along with a macro "#define platform\_get\_device\_id(pdev) ((pdev)->id\_entry)". The terminal window has a dark background with light-colored text. The status bar at the bottom right shows "35,1" and "4%".

如下图所示，给结构体“platform\_device”添加了注释。

```
struct platform_device {
    const char * name;//设备名称，在sys/devices会显示
    int id;//设备id，用于插入总线并且具有相同name的设备编号，  
如果只有一个设备那么-1
    struct device dev;//结构体中内嵌的device结构体
    u32 num_resources;//设备使用资源的数量
    struct resource * resource;//设备使用的资源数组

    const struct platform_device_id *id_entry;

    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;

    /* arch specific additions */
    struct pdev_archdata archdata;
```

在结构体“platform\_device”中

第一个参数 “name” , 是一个字符指针 , 驱动初始化前需要和注册驱动的 “name” 字段匹配的参数 ;

第二个参数 “id” , 表示子设备编号 , 一个设备如果有多个子设备号 , 则需写入子设备号数量 , 如果只有一个则用-1 表示 ;

第三个参数 “device” , 表示结构体内嵌的设备结构体 ;

第四个参数 num\_resource , 表示设备使用的资源数组 ;

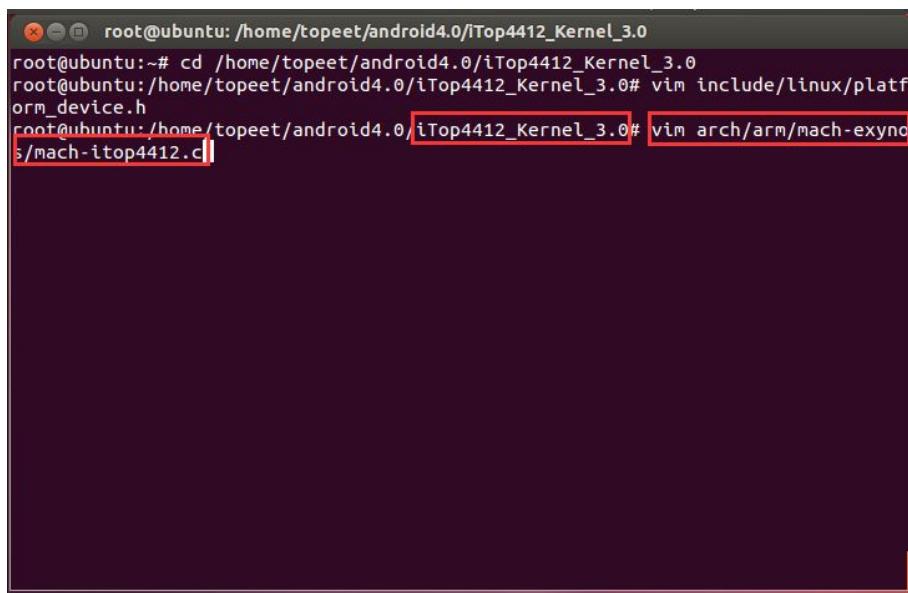
这些参数不一定全部使用 , 在大多数驱动中 , 需要写的只有设备名 \*name 和设备编号 id , 常用的还有资源数组 \*resource , 后面的参数在用到的时候再介绍。

## 6.4 添加设备到平台总线

在实验 5 中介绍了注册设备的流程 , 并没有详细介绍如何操作 , 像平台总线中添加 Linux 设备的方法很简单 , 下面就详细介绍一下基本步骤。

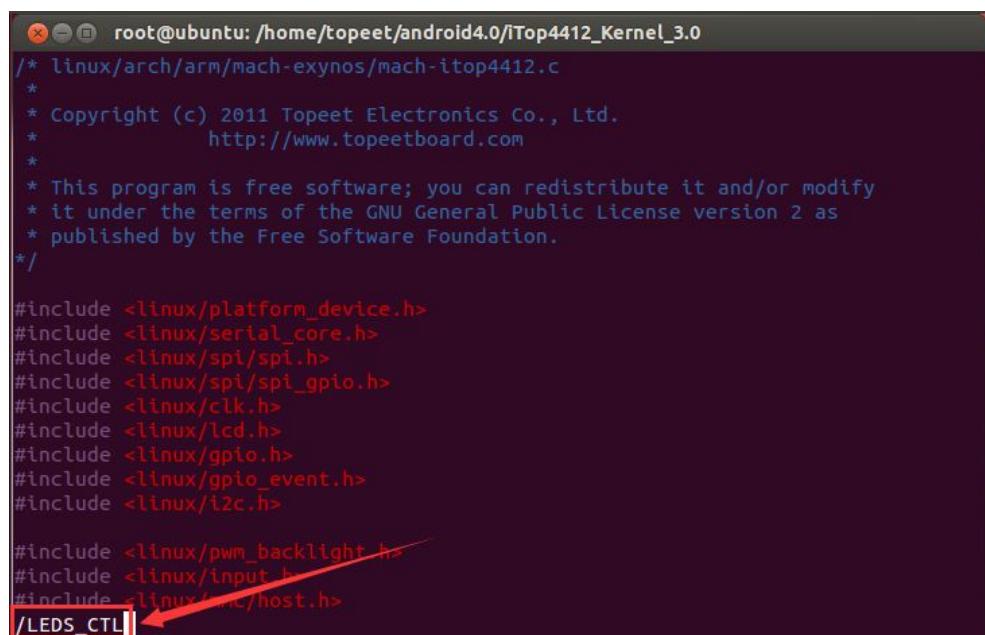
前面给大家介绍的平台文件 “arch/arm/mach-exynos/mach-itop4412.c” , 并没有介绍这个文件和注册平台设备的关系。

如下图所示 , 使用命令 “vim arch/arm/mach-exynos/mach-itop4412.c” , 打开平台文件。



```
root@ubuntu:~# cd /home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim include/linux/platform_device.h
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim arch/arm/mach-exynos/mach-itop4412.c
```

现在介绍如何添加最简单的设备，led 的驱动相对来说很简单，在里面查找宏定义“LEDS\_CTL”，如下图所示。



```
/* linux/arch/arm/mach-exynos/mach-itop4412.c
 *
 * Copyright (c) 2011 Topeet Electronics Co., Ltd.
 * http://www.topeetboard.com
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
#include <linux/platform_device.h>
#include <linux/serial_core.h>
#include <linux/spi/spi.h>
#include <linux/spi/spi_gpio.h>
#include <linux/clk.h>
#include <linux/lcd.h>
#include <linux/gpio.h>
#include <linux/gpio_event.h>
#include <linux/i2c.h>

#include <linux/pwm_backlight.h>
#include <linux/input.h>
#include <linux/firmware/host.h>
/LEDS_CTL
```

如下图所示，找到 leds 注册设备的代码。

```
#ifdef CONFIG_LEDS_CTL
struct platform_device s3c_device_leds_ctl = {
    .name    = "leds",
    .id      = -1,
};

#endif
#ifndef CONFIG_BUZZER_CTL
struct platform_device s3c_device_buzzer_ctl = {
    .name    = "buzzer_ctl",
    .id      = -1,
};
#endif

struct platform_device s3c_device_keyboard = {
    .name    = "samsung_keypad",
    .id      = "-1",
};

struct platform_device s3c_device_irq_test = {
    .name    = "irq_test",
    .id      = -1,
};
```

2521,15 53%

在上图红色方框中，注册平台设备结构体 “platform\_device” 中，只调用了两个参数

“\*name” 和 “id” 。

如下图所示，仿照着这段代码在它前面添加一个设备 “hello\_ctl” 。

```
.name    = "max485_ctl",
.id      = -1,
};

#endif

#ifndef CONFIG_HELLO_CTL
struct platform_device s3c_device_hello_ctl = {
    .name = "hello_ctl",
    .id   = -1,
};
#endif

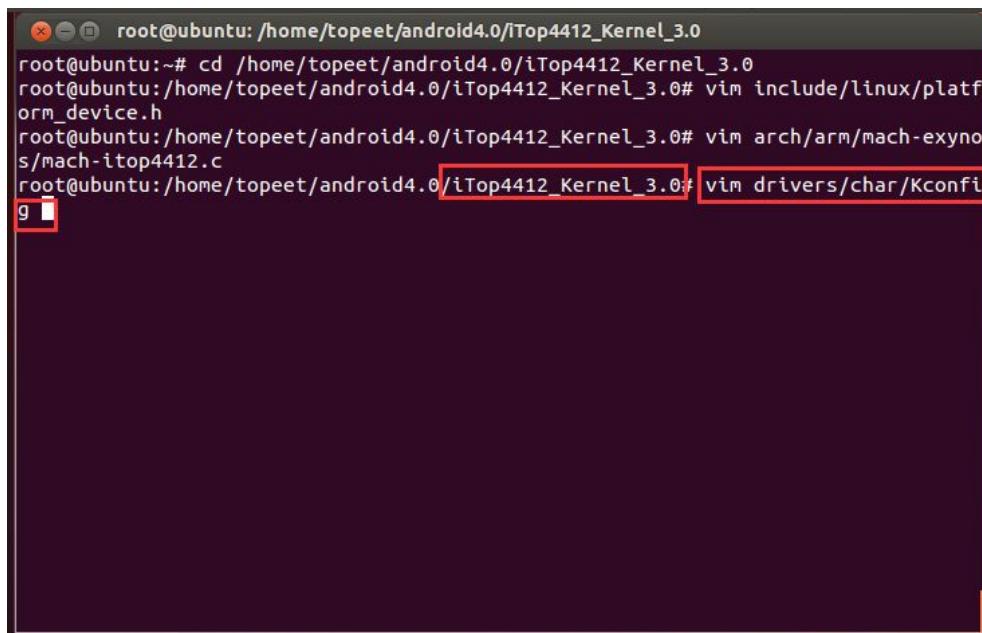
#endif
#ifndef CONFIG_LEDS_CTL
struct platform_device s3c_device_leds_ctl = {
    .name    = "leds",
    .id      = -1,
};
#endif

#ifndef CONFIG_BUZZER_CTL
struct platform_device s3c_device_buzzer_ctl = {
    .name    = "buzzer_ctl",
    .id      = -1,
};
```

2509,15 53%

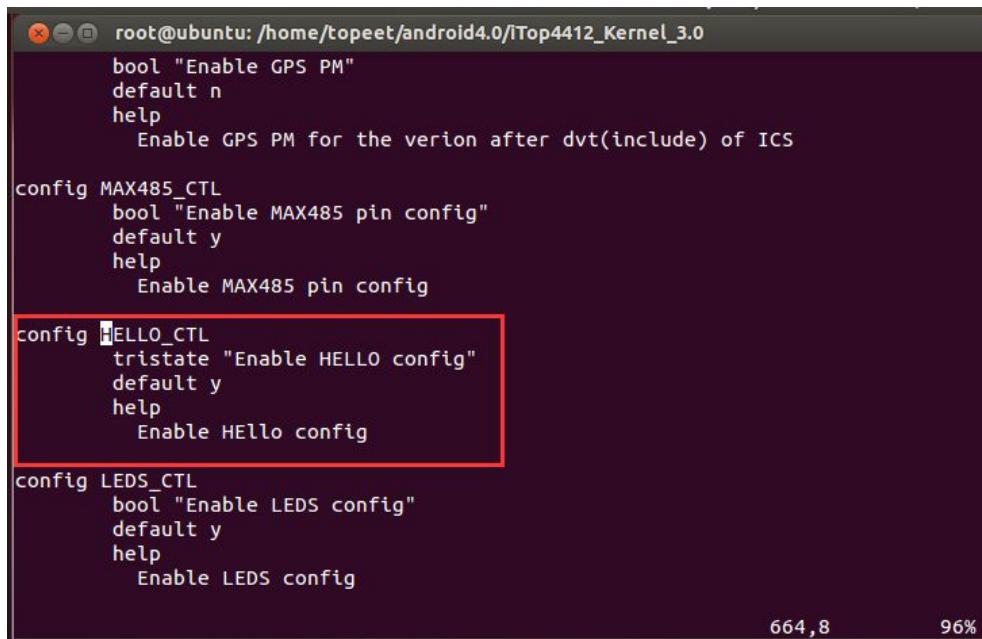
这里还需要确认一下，是否有 “HELLO\_CTL” 宏定义，只有定义了这个宏，在生成内核的时候才会将其编译到内核。

在前面关于 Kconfig 实验中，已经添加了 “HELLO\_CTL” 宏，如下图所示，使用命令 “vim drivers/char/Kconfig ” 打开前面定义过 “HELLO\_CTL” 的配置文件。



```
root@ubuntu:~# cd /home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim include/linux/platform_device.h
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim arch/arm/mach-exynos/mach-itop4412.c
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim drivers/char/Kconfig
```

如下图所示，已经定义。



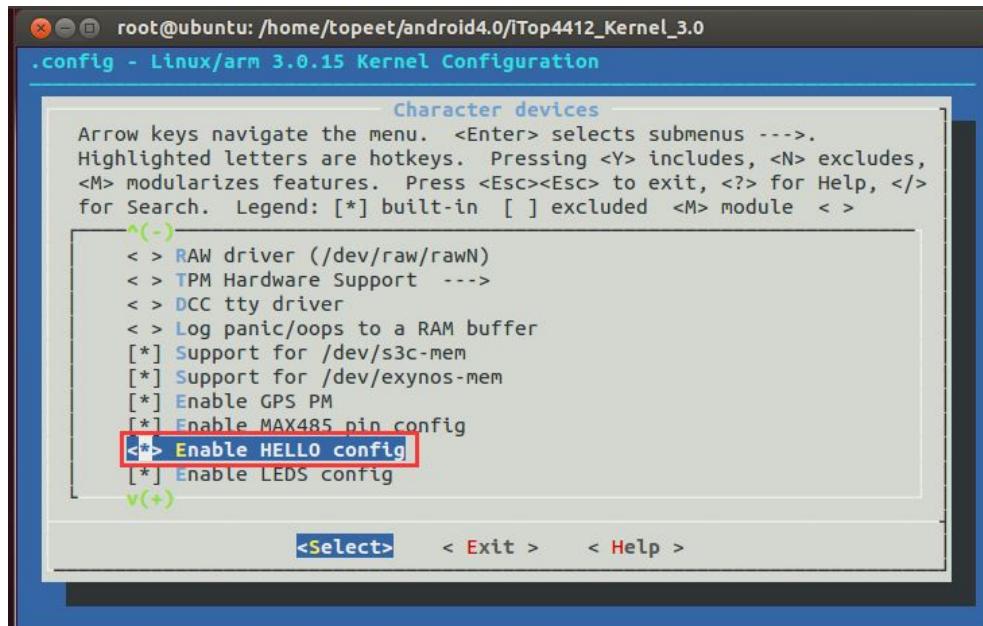
```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
bool "Enable GPS PM"
default n
help
    Enable GPS PM for the verion after dvt(include) of ICS

config MAX485_CTL
    bool "Enable MAX485 pin config"
    default y
    help
        Enable MAX485 pin config

config HELLO_CTL
    tristate "Enable HELLO config"
    default y
    help
        Enable HELLO config

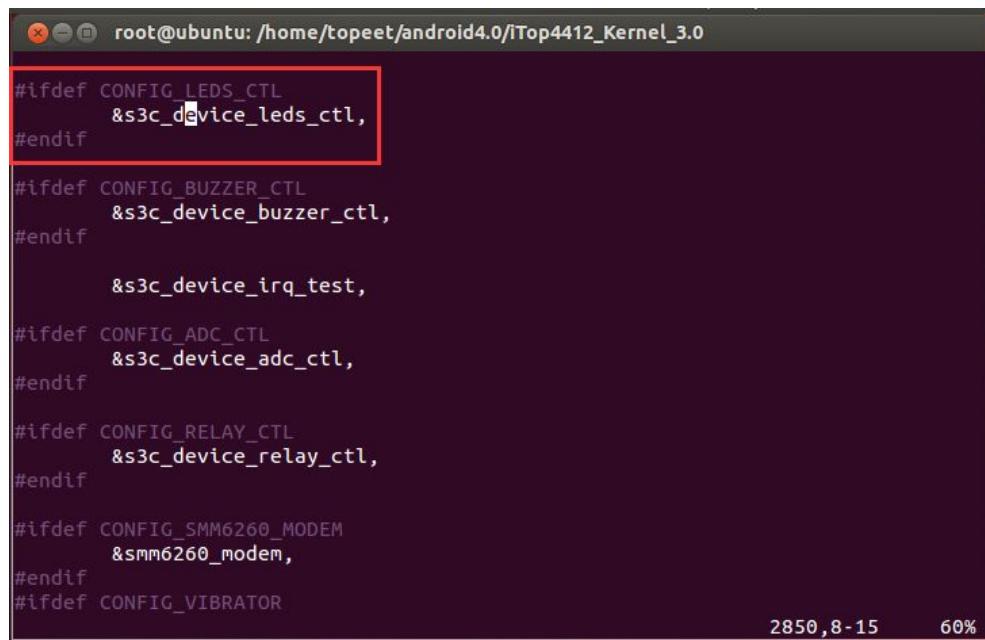
config LEDS_CTL
    bool "Enable LEDS config"
    default y
    help
        Enable LEDS config
```

接着到 menuconfig 中将其配置上 , 使用命令 “make menuconfig” , 进入 Device Drivers ---> ” → “Character devices ---> ” → “Enable HELLO config” , 如下图所示 , 配置上宏定义 “HELLO\_CTL” 。



配置后保存退出。这样就确认了宏定义 “HELLO\_CTL” 已经出现。

接着再次打开 “arch/arm/mach-exynos/mach-itop4412.c” 平台文件 , 再搜索 “LEDS\_CTL” 。如下图所示 , 查找到设备初始化的代码 , 这一段比较简单 , 仿照着写即可。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
#endif CONFIG_LEDs_CTL
    &s3c_device_leds_ctl,
#endif

#ifndef CONFIG_BUZZER_CTL
    &s3c_device_buzzer_ctl,
#endif

    &s3c_device_irq_test,

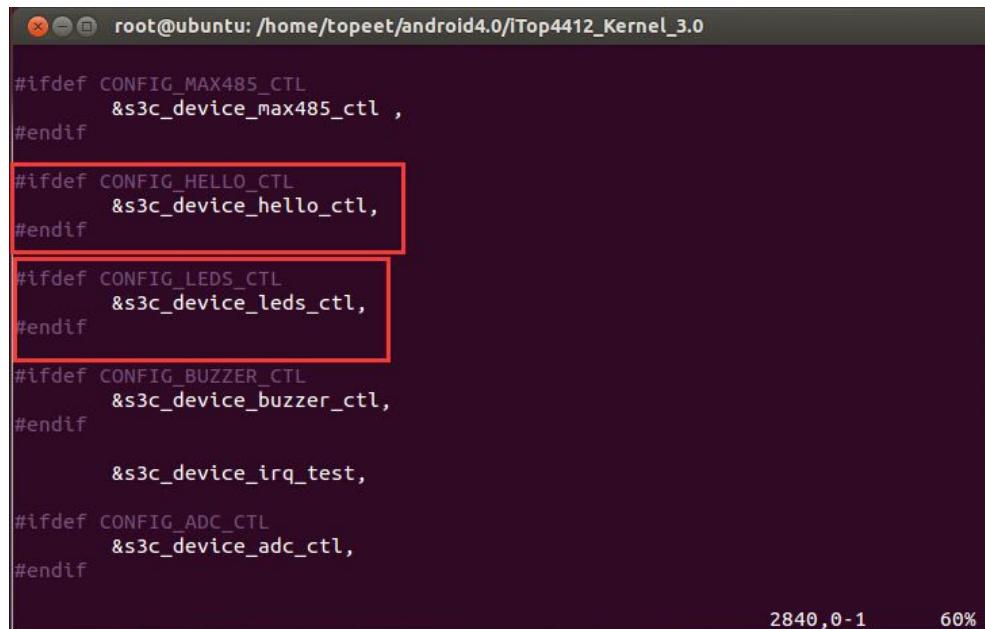
#ifndef CONFIG_ADC_CTL
    &s3c_device_adc_ctl,
#endif

#ifndef CONFIG_RELAY_CTL
    &s3c_device_relay_ctl,
#endif

#ifndef CONFIG_SMM6260_MODEM
    &smm6260_modem,
#endif
#ifndef CONFIG_VIBRATOR
```

2850,8-15 60%

如下图所示，添加 hello 设备的代码。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
#endif CONFIG_MAX485_CTL
    &s3c_device_max485_ctl ,
#endif

#ifndef CONFIG_HELLO_CTL
    &s3c_device_hello_ctl,
#endif

#ifndef CONFIG_LEDs_CTL
    &s3c_device_leds_ctl,
#endif

#ifndef CONFIG_BUZZER_CTL
    &s3c_device_buzzer_ctl,
#endif

    &s3c_device_irq_test,

#ifndef CONFIG_ADC_CTL
    &s3c_device_adc_ctl,
#endif
```

2840,0-1 60%

保存退出，重新编译内核，烧写到开发板。

开发板启动之后，使用命令 “ls /sys/devices/platform/” 可以查看到新注册的 hello 设备，如下图所示。

```
[root@iTOP-4412]# ls /sys/devices/platform/
adc_ctl                  relay_ctl          samsung-audio
alarm                   s3c-p1330.1        samsung-audio-idma
android_pmem.0           s3c-p1330.2        samsung-i2s.0
android_pmem.1           s3c-sdhci.2       samsung-i2s.4
arm-pmu.0                s3c-sdhci.3       samsung-keypad
bt-sysfs                 s3c-usbgadget    samsung-kmsg
buzzer_ctl               s3c2410-wdt      samsung-pd.0
dw_mmc                   s3c2440-i2c.1     samsung-pd.1
exynos-busfreq           s3c2440-i2c.3     samsung-pd.2
exynos-usb-switch        s3c2440-i2c.4     samsung-pd.4
exynos4412-adc          s3c2440-i2c.5     samsung-pd.5
gpio-keys                s3c2440-i2c.7     samsung-pd.6
hello_ctl                s3c24xx-pwm.1    samsung-pd.7
i2c-gpio.0               s3c64xx-rtc      samsung-rp
ion-exynos               s3c64xx-spi.2   serial8250
irq_test                 s5p-ehci          si_gps
leds                     s5p-pmic          snd-soc-dummy
max485_ctl               s5p-sysmmu.15   soc-audio
power                    s5pv210-uart.0  switch-gpio.0
power.0                  s5pv210-uart.1  tc4-regulator-consumer
reg-dummy                s5pv210-uart.2  uevent
regulatory.0             s5pv210-uart.3  wlan_ar6000_pm_dev.1
[root@iTOP-4412]#
```

# 实验 07 驱动注册

## 7.1 本章导读

前面的实验介绍过注册驱动的流程，注册的时候需要和设备匹配，简单介绍了将驱动注册到平台设备的结构体“platform\_driver\_register”，并没有介绍怎么使用。本章就给大家详细介绍如何使用这个结构体来注册驱动。

### 7.1.1 工具

#### 7.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 7.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 实验配套源码文件夹“probe\_linux\_module”

## 7.1.2 预备课程

### 7.1.3 视频和代码资源

本节配套视频为“视频 07\_驱动注册”

源码为“probe\_linux\_module”

## 7.2 学习目标

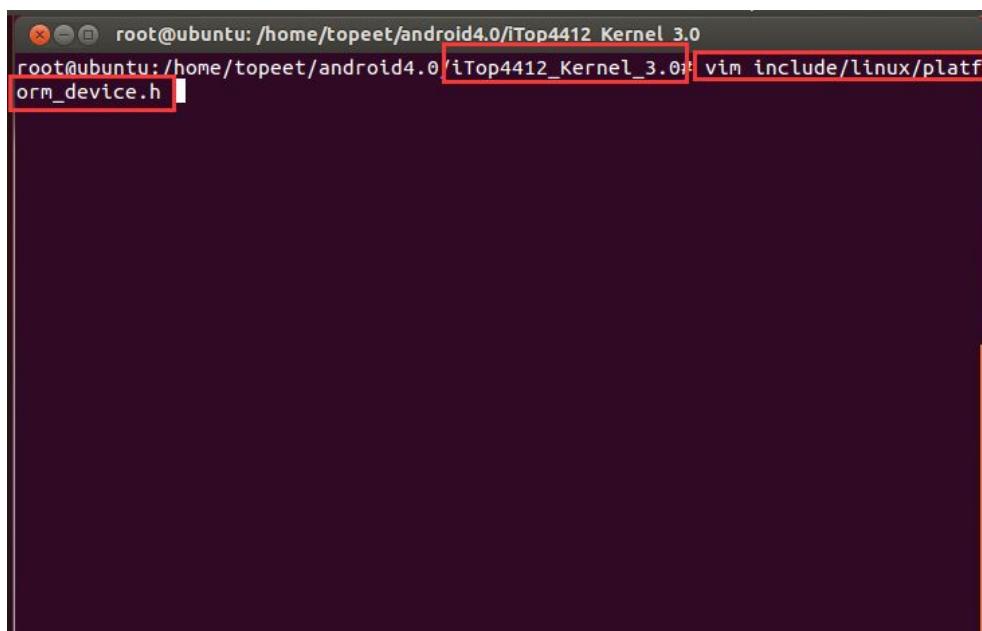
本章需要学习以下内容：

掌握将驱动注册到虚拟平台总线的方法

## 7.3 platform\_driver\_register 和 platform\_driver\_unregister 函数

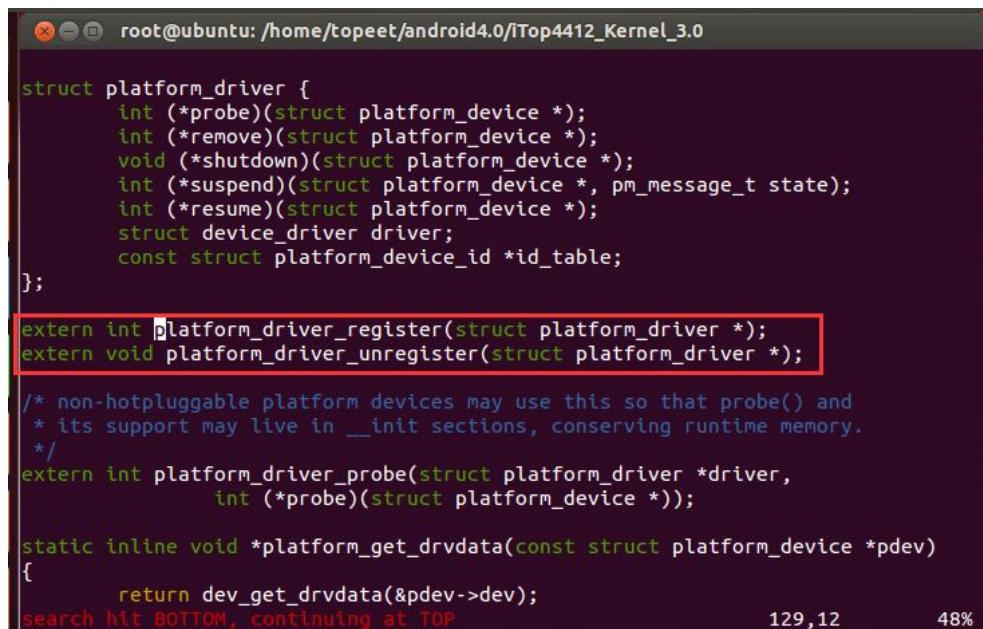
platform\_driver\_register 函数和 platform\_driver\_unregister 函数用于注册和卸载驱动。

如下图所示，在 Linux 源码目录下，使用命令“vim include/linux/platform\_device.h”。



A screenshot of a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The user has typed the command "vim include/linux/platform\_device.h" and is in the process of pressing the Enter key. The terminal window has a dark background and light-colored text.

查找一下“platform\_driver\_register”，如下图所示。



The screenshot shows a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The code displayed is part of the Linux kernel's platform driver interface. It defines a struct platform\_driver with various callbacks and pointers to device drivers and device IDs. Two external functions are declared: platform\_driver\_register and platform\_driver\_unregister. A note about non-hotpluggable devices is present. The terminal window also shows some search history at the bottom.

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};

extern int platform_driver_register(struct platform_driver *);
extern void platform_driver_unregister(struct platform_driver *);

/* non-hotpluggable platform devices may use this so that probe() and
 * its support may live in __init sections, conserving runtime memory.
 */
extern int platform_driver_probe(struct platform_driver *driver,
                                 int (*probe)(struct platform_device *));

static inline void *platform_get_drvdata(const struct platform_device *pdev)
{
    return dev_get_drvdata(&pdev->dev);
}

search hit BOTTOM, continuing at TOP
```

如上图所示。

注册驱动的函数：

```
extern int platform_driver_register(struct platform_driver *)
```

卸载驱动的函数：

```
extern void platform_driver_unregister(struct platform_driver *)
```

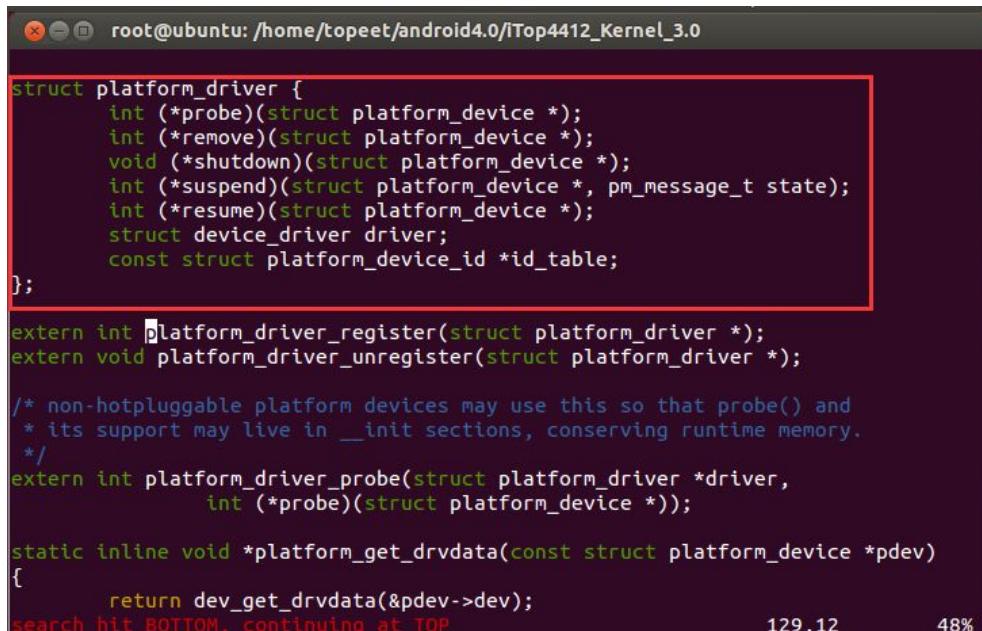
具体这两个函数是怎么实现的，大家其实不用去关心，都是前人做好的，只要掌握如何调用即可。

这两个函数都会调用一个 platform\_driver 类型的结构体。

## 7.4 platform\_driver 结构体

前一小节介绍到注册和卸载驱动的函数需要调用 platform\_driver 类型结构体。这个结构体非常重要，大家一定要掌握它的使用方法。

如下图所示，这个结构体也是定义在 “include/linux/platform\_device.h” 头文件中。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};

extern int platform_driver_register(struct platform_driver *);
extern void platform_driver_unregister(struct platform_driver *);

/* non-hotpluggable platform devices may use this so that probe() and
 * its support may live in __init sections, conserving runtime memory.
 */
extern int platform_driver_probe(struct platform_driver *driver,
                                 int (*probe)(struct platform_device *));

static inline void *platform_get_drvdata(const struct platform_device *pdev)
{
    return dev_get_drvdata(&pdev->dev);
}
search hit BOTTOM, continuing at TOP
129,12          48%
```

这个结构体需要好好分析，这个结构体提非常非常的重要，几乎所有的驱动都会用到。

该结构中包含了一组操作函数和一个 struct device\_driver 的对像。在驱动中首先要做的就是定义 platform\_driver 中的函数，并创建这个结构的一个对象实例，然后在 init() 函数中调用 platform\_driver\_register() 向系统注册驱动。

函数 int (\*probe)(struct platform\_device \*);

主要是进行设备的探测和初始化。例如想调用一个 GPIO，那么首先需要探测这个 GPIO 是否被占用了，如果被占用了那么初始化失败，驱动注册也就失败了；如果没有被占用，那么就申明要占用它。

该函数中一般还会添加生成设备节点的函数，如果初始化成功，那么就会需要添加设备节点。设备节点的知识在下一节介绍。

函数 `int (*remove)(struct platform_device *);`

移除驱动，该函数中一般用于去掉设备节点或者释放软硬件资源。

接着的三个函数：

`void (*shutdown)(struct platform_device *);`

`int (*suspend)(struct platform_device *, pm_message_t state);`

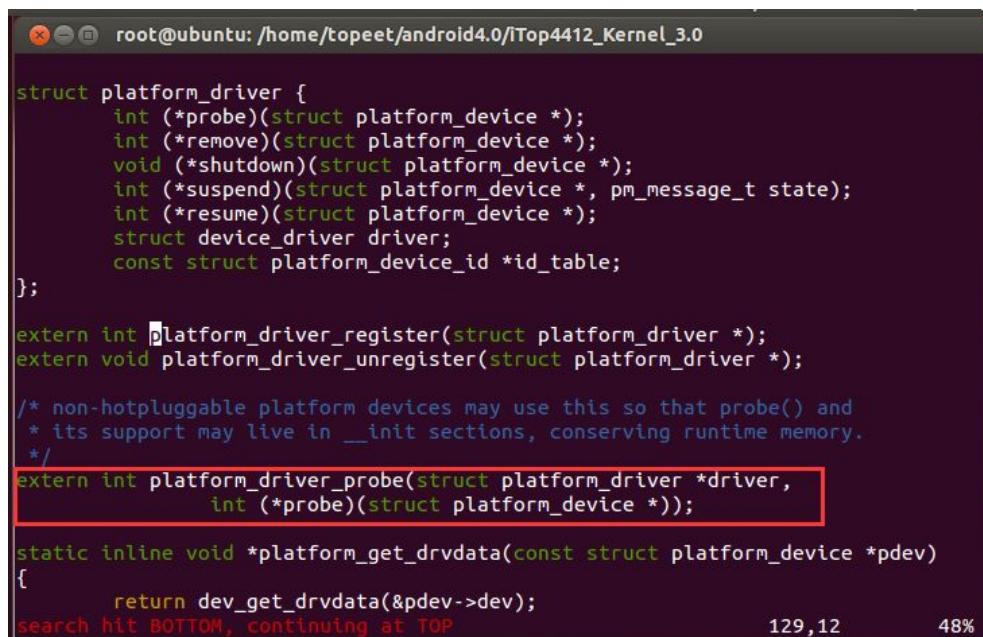
`int (*resume)(struct platform_device *);`

从字面上就很好理解了，关闭驱动，悬挂（休眠）驱动以及恢复的时候该驱动要做什么。

接着的结构体 `struct device_driver driver;`

主要包含两个参数，一个是 `name` 参数，驱动名称（需要和设备驱动结构体中的 `name` 参数一样）；一个是 `owner`，一般是 `THIS_MODULE`。

下面接着给大家介绍一个小知识点，以 `platform_driver` 结构体中的参数 `probe` 为例，这个参数指向 `platform_driver_probe` 函数，如下图所示。



```

struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};

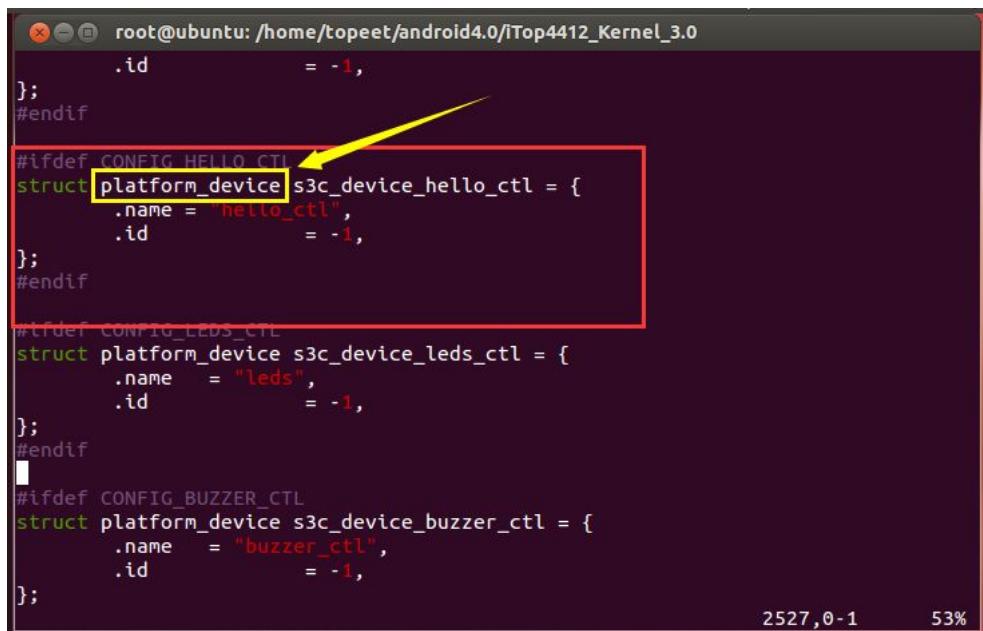
extern int platform_driver_register(struct platform_driver *);
extern void platform_driver_unregister(struct platform_driver *);

/* non-hotpluggable platform devices may use this so that probe() and
 * its support may live in __init sections, conserving runtime memory.
 */
extern int platform_driver_probe(struct platform_driver *driver,
                                 int (*probe)(struct platform_device *));
static inline void *platform_get_drvdata(const struct platform_device *pdev)
{
    return dev_get_drvdata(&pdev->dev);
}

```

在视频中没有介绍这个参数 platform\_driver 是从哪里传来的。

大家打开平台文件，看一下注册设备的代码，如下图所示，是 HELLO\_CTL 的结构体。



```

        .id      = -1,
};

#endif

#ifndef CONFIG_HELLO_CTL
struct platform_device s3c_device_hello_ctl = {
    .name = "hello_ctl",
    .id   = -1,
};
#endif

#ifndef CONFIG_LEDS_CTL
struct platform_device s3c_device_leds_ctl = {
    .name  = "leds",
    .id    = -1,
};
#endif

#ifndef CONFIG_BUZZER_CTL
struct platform_device s3c_device_buzzer_ctl = {
    .name  = "buzzer_ctl",
    .id    = -1,
};

```

上图红色的结构体和 platform\_driver\_probe 函数的第一个参数类型一样，如果你注册一个 HELLO\_CTL 驱动，那么你的初始化函数 platform\_driver\_probe 就会调用这个结构体。

虽然这个结构体在这里很简单，但是在后面专门的移植课程中，你会发现移植中有部分工作是解决这个结构体中的问题，另一部分工作是调试 bug。在具体使用的时候再详细介绍相关知识，这里只要知道注册驱动的时候会调用平台文件中对应结构体即可。

## 7.5 实验操作

在前面“实验 2mini\_linux\_module”的基础上添加注册驱动的代码。

如下图所示，是“mini\_linux\_module.c”的代码。

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3
4 MODULE_LICENSE("Dual BSD/GPL");
5 MODULE_AUTHOR("TOPEET");
6
7 static int hello_init(void)
8 {
9     printk(KERN_EMERG "HELLO WORLD enter!\n");
10    return 0;
11 }
12
13 static void hello_exit(void)
14 {
15     printk(KERN_EMERG "HELLO WORLD exit!\n");
16 }
17
18 module_init(hello_init);
19 module_exit(hello_exit);
20
21
```

先将这个文件名改为 probe\_linux\_module.c。

然后在 probe\_linux\_module.c 文件中开始修改和添加带代码。

如下图所示，首先需要添加头文件 “#include <linux/platform\_device.h>” ，然后定义一个宏变量 DRIVER\_NAME , 定义为 “hello\_ctl” ，需要和前面注册的 hello 设备的名称相同。

```
#include <linux/init.h>
#include <linux/module.h>

/*驱动注册的头文件，包含驱动的结构体和注册和卸载的函数*/
#include <linux/platform_device.h>

#define DRIVER_NAME "hello_ctl"

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("TOPEET");
```

然后在模块入口和出口调用函数 platform\_driver\_register 和 platform\_driver\_unregister , 如下图所示 , 先将参数名定义为 “&hello\_driver” 。另外注册驱动的时候 , 会返回数值 , 将其打印出来判断是否注册成功。

```
static int hello_init(void)
{
    int DriverState;

    printk(KERN_EMERG "HELLO WORLD enter!\n");
    DriverState = platform_driver_register(&hello_driver);

    printk(KERN_EMERG "\tDriverState is %d\n",DriverState);
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_EMERG "HELLO WORLD exit!\n");

    platform_driver_unregister(&hello_driver);
}

module_init(hello_init);
module_exit(hello_exit);
```

如下图所示 , 定义结构体 “hello\_driver” 。

driver 中的 name 参数就是驱动名称 , 这里将前面定义的宏变量 DRIVER\_NAME 赋给它 ; 另外一个参数 owner 一般默认为 THIS\_MODULE 。

```
struct platform_driver hello_driver = {
    .probe = hello_probe,
    .remove = hello_remove,
    .shutdown = hello_shutdown,
    .suspend = hello_suspend,
    .resume = hello_resume,
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
    }
};
```

然后定义函数 hello\_probe、hello\_remove、hello\_shutdown、hello\_suspend、

hello\_resume。

```
static int hello_probe(struct platform_device *pdv) {
    printk(KERN_EMERG "\tinitialized\n");
    return 0;
}

static int hello_remove(struct platform_device *pdv) {
    return 0;
}

static void hello_shutdown(struct platform_device *pdv) {
    ;
}

static int hello_suspend(struct platform_device *pdv) {
    return 0;
}

static int hello_resume(struct platform_device *pdv) {
    return 0;
}
```

如上图所示，这里在 hello\_probe 函数中添加打印信息 “printk(KERN\_EMERG

"\tinitialized\n");”

如果设备和驱动匹配成功就会进入函数 hello\_probe 打印 “initialized” 。

接着需要修改一下 Makefile 文件，将“ mini\_linux\_module.o ” 改为 “probe\_linux\_module.o” ，注释部分用户可以自己修改，如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件iTop4412_hello.o
obj-m += probe_linux_module.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

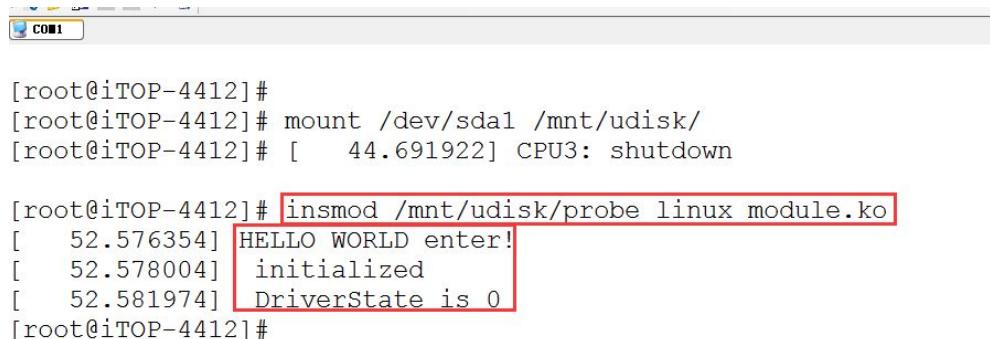
在 Ubuntu 中的目录 “/home/topeet” 中新建目录 “probe\_linux\_module” ，拷贝驱动文件 “probe\_linux\_module.c” 和编译文件 “Makefile” 到新建中，如下图所示。

```
root@ubuntu:~# cd /home/topeet/
root@ubuntu:/home/topeet# mkdir probe_linux_module/
root@ubuntu:/home/topeet# ls probe_linux_module/
Makefile  probe_linux_module.c
root@ubuntu:/home/topeet#
```

进入 “probe\_linux\_module” 目录，使用命令 “make” 编译 “probe\_linux\_module.c” ，  
如下图所示，生成模块文件 “probe\_linux\_module.ko” 。

```
root@ubuntu:~/home/topeet/probe_linux_module
root@ubuntu:~# cd /home/topeet/
root@ubuntu:/home/topeet# mkdir probe_linux_module/
root@ubuntu:/home/topeet# ls probe_linux_module/
Makefile  probe_linux_module.c
root@ubuntu:/home/topeet# cd probe_linux_module/
root@ubuntu:/home/topeet/probe_linux_module# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/probe_linux_m
odule modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M]  /home/topeet/probe_linux_module/probe_linux_module.o
/home/topeet/probe_linux_module/probe_linux_module.c:43: warning: initialization
from incompatible pointer type
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/topeet/probe_linux_module/probe_linux_module.mod.o
  LD [M]  /home/topeet/probe_linux_module/probe_linux_module.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/probe_linux_module# ls
Makefile      probe_linux_module.c      probe_linux_module.mod.o
modules.order  probe_linux_module.ko   probe_linux_module.o
Module.symvers probe_linux_module.mod.c
root@ubuntu:/home/topeet/probe_linux_module#
```

启动开发板，拷贝 “probe\_linux\_module.ko” 到 U 盘，将 U 盘插入开发板，加载驱动文件 “probe\_linux\_module.ko” ，如下图所示，可以看到打印出了 “initialized” ，表明进入了 probe 函数。



```
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# [ 44.691922] CPU3: shutdown

[root@iTOP-4412]# insmod /mnt/udisk/probe_linux_module.ko
[ 52.576354] HELLO WORLD enter!
[ 52.578004] initialized
[ 52.581974] DriverState is 0
[root@iTOP-4412]#
```

使用命令 “rmmod probe\_linux\_module” 卸载驱动，可以看到打印 “HELLO WORLD exit!” ，表明卸载成功。

```
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# [ 44.691922] CPU3: shutdown

[root@iTOP-4412]# insmod /mnt/udisk/probe_linux_module.ko
[ 52.576354] HELLO WORLD enter!
[ 52.578004] initialized
[ 52.581974] DriverState is 0
[root@iTOP-4412]# lsmod
probe_linux_module 1244 0 - Live 0xbff00000
[root@iTOP-4412]# rmmod probe_linux_
[ 116.633719] HELLO WORLD exit!
[root@iTOP-4412]#
```

# 实验 08\_生成设备节点

## 8.1 本章导读

一部分驱动要和上层通信，都需要生成设备节点，上层应用通过一套标准的接口函数调用设备节点就可以控制底层以及和底层通信。

本章就给大家介绍最简单易用的杂项设备节点如何生成。

### 8.1.1 工具

#### 8.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 8.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 实验配套源码文件夹 “devicenode\_linux\_module”

### 8.1.2 预备课程

实验 06\_设备注册

实验 07\_驱动注册

### 8.1.3 视频资源

本节配套视频为“视频 08\_生成设备节点”

## 8.2 学习目标

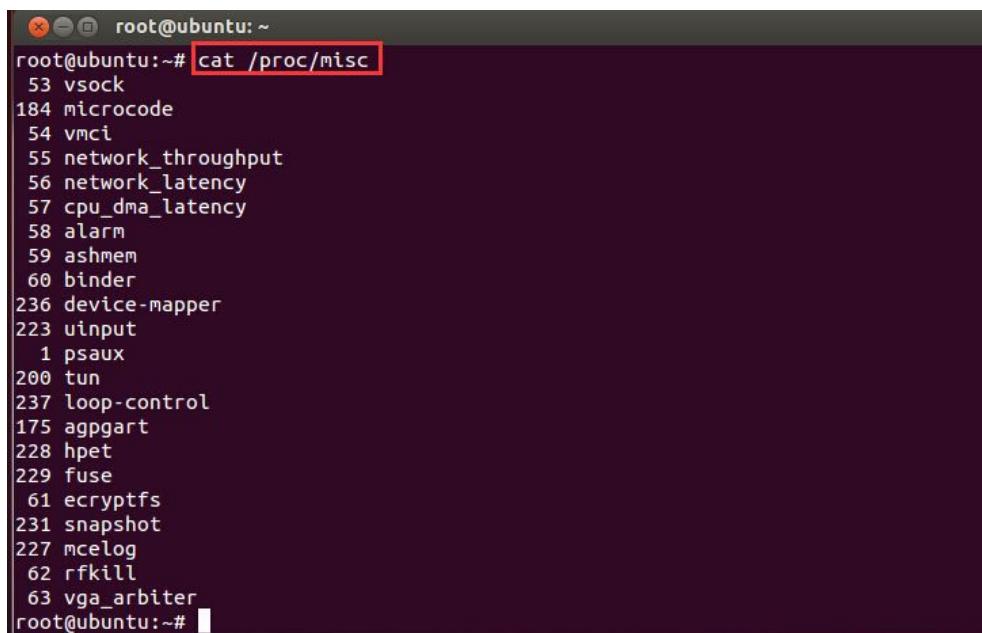
本章需要学习以下内容：

理解什么是杂项设备

生成杂项设备的设备节点

## 8.3 为什么引入杂项设备

在虚拟机的 Ubuntu 系统上，如下图所示，使用命令“cat /proc/misc”，可以查看到 PC 机 Ubuntu 系统的杂项设备。



```
root@ubuntu:~# cat /proc/misc
53 vsock
184 microcode
54 vmci
55 network_throughput
56 network_latency
57 cpu_dma_latency
58 alarm
59 ashmem
60 binder
236 device-mapper
223 uinput
  1 psaux
200 tun
237 loop-control
175 agpgart
228 hpet
229 fuse
  61 ecryptfs
231 snapshot
227 mcelog
  62 rfkill
  63 vga_arbiter
root@ubuntu:~#
```

启动开发板，在超级终端中输入命令“cat /proc/misc”也可以查看对应的杂项设备。

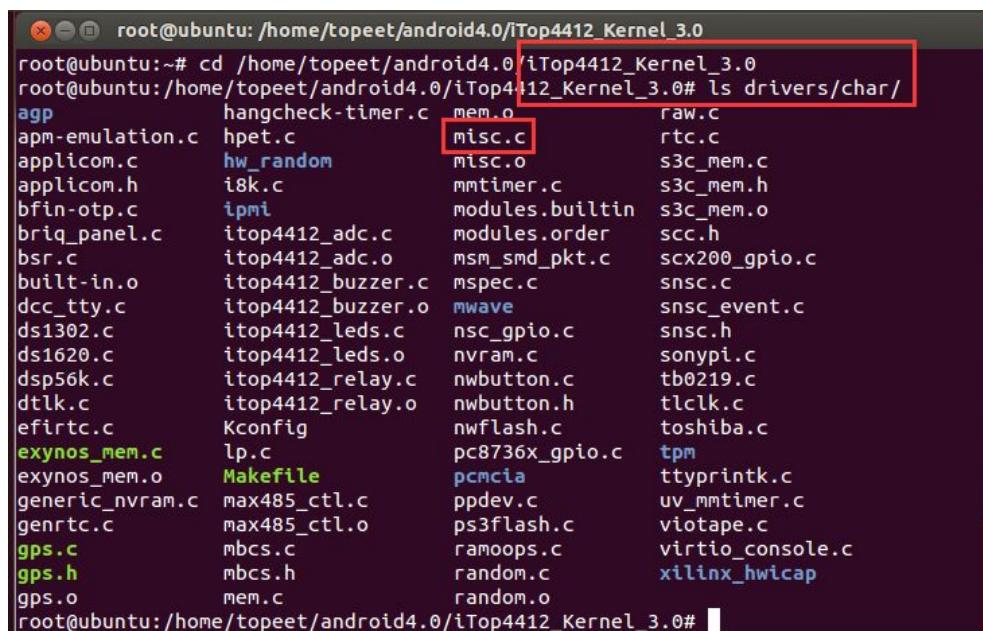
```
[root@iTOP-4412]# cat /proc/misc
47 network_throughput
48 network_latency
49 cpu_dma_latency
50 xt_qtaguid
251 srp_ctrl
253 srp
236 device-mapper
130 watchdog
51 alarm
223 uinput
52 keychord
53 usb_accessory
54 mtp_usb
55 android_adb
1 pmem_gpu1
0 pmem
56 relay_ctl
57 adc
58 buzzer_ctl
59 leds
60 max485_ctl_pin
61 AGPS
229 fuse
62 ashmem
63 ion
[root@iTOP-4412]#
```

前面介绍过主设备号只有 256 个，设备又非常多，所以引入了子设备号。

其中杂项设备的主设备号是 10，在任何 Linux 系统中它都是固定的。

一般将 Linux 驱动分为字符设备、块设备、网络设备，但是这个分类不能包含所有的设备，所以将无法归类的设备统称为杂项设备，杂项设备可能用到字符设备、快设备、网络设备中的一项或者多项设备。

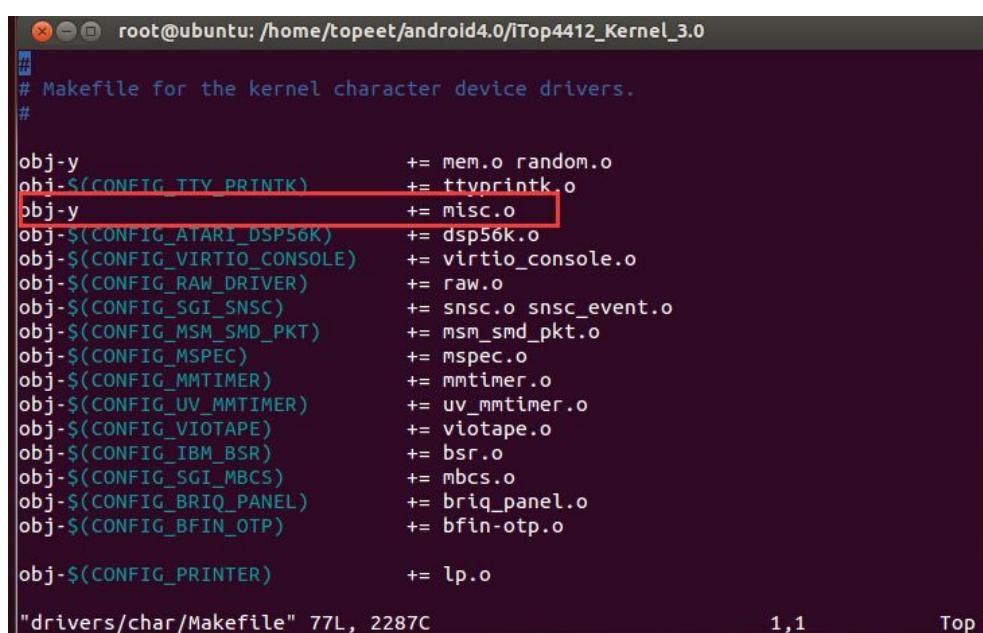
如下图所示，进入源码文件夹，使用命令 “ls drivers/char/” ，可以查看到杂项设备的文件 “misc.c” 。



```
root@ubuntu:~/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:~# cd /home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# ls drivers/char/
agp          hangcheck-timer.c  mem.o      raw.c
apm-emulation.c hpet.c        misc.c     rtc.c
applicom.c    hw_random       misc.o     s3c_mem.c
applicom.h    i8k.c          mmtimer.c  s3c_mem.h
bfin-otp.c    ipmi           modules.builtin s3c_mem.o
briq_panel.c  itop4412_adc.c  modules.order scc.h
bsr.c         itop4412_adc.o   msm_smd_pkt.c scx200_gpio.c
built-in.o    itop4412_buzzer.c mspec.c   snsc.c
dcc_tty.c    itop4412_buzzer.o  mwave     snsc_event.c
ds1302.c     itop4412_leds.c   nsc_gpio.c snsc.h
ds1620.c     itop4412_leds.o   nvram.c   sonypi.c
dsp56k.c     itop4412_relay.c  nwbutton.c tb0219.c
dtlk.c       itop4412_relay.o   nwbutton.h tlclk.c
efirtc.c     Kconfig          nwflash.c toshiba.c
exynos_mem.c lp.c            pc8736x_gpio.c tpm
exynos_mem.o Makefile        pcmcia     ttyprintk.c
generic_nvram.c max485_ctl.c  ppdev.c   uv_mmtimer.c
genrtc.c     max485_ctl.o    ps3flash.c viotape.c
gps.c        mbcs.c          ramoops.c virtio_console.c
gps.h        mbcs.h          random.c  xilinx_hwicap
gps.o        mem.c           random.o
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0#
```

如上图所示，可以看到它被编译为“misc.o”，也就是被编译进了内核 zImage 文件。

使用命令“vim drivers/char/Makefile”打开杂项设备文件的编译文件。如下图所示，可以看到，这个文件相当于被强制编译的。



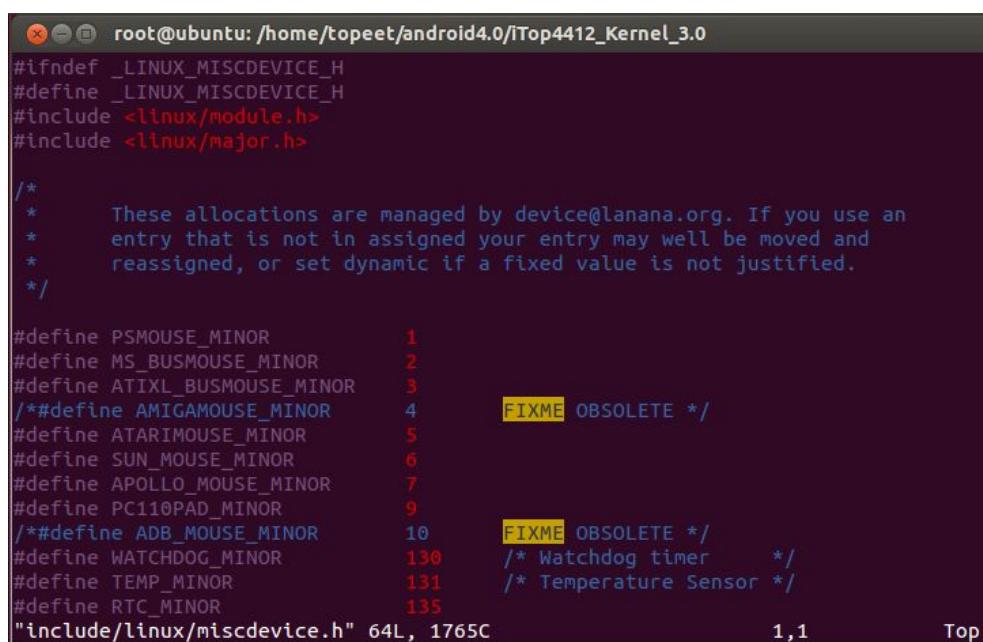
```
root@ubuntu:~/home/topeet/android4.0/iTop4412_Kernel_3.0
#
# Makefile for the kernel character device drivers.
#
obj-y          += mem.o random.o
obj-$(CONFIG_TTY_PRINTRK) += ttyprintk.o
obj-y          += misc.o
obj-$(CONFIG_ATARI_DSP56K) += dsp56k.o
obj-$(CONFIG_VIRTIO_CONSOLE) += virtio_console.o
obj-$(CONFIG_RAW_DRIVER)    += raw.o
obj-$(CONFIG_SGI_SNDC)      += snsc.o snsc_event.o
obj-$(CONFIG_MSM_SMD_PKT)   += msm_smd_pkt.o
obj-$(CONFIG_MSPEC)         += mspec.o
obj-$(CONFIG_MMTIMER)       += mmtimer.o
obj-$(CONFIG_UV_MMTIMER)    += uv_mmtimer.o
obj-$(CONFIG_VIOTAPE)       += viotape.o
obj-$(CONFIG_IBM_BSR)       += bsr.o
obj-$(CONFIG_SGI_MBDS)      += mbcs.o
obj-$(CONFIG_BRIQ_PANEL)    += briq_panel.o
obj-$(CONFIG_BFIN_OTP)       += bfin-otp.o
obj-$(CONFIG_PRINTER)       += lp.o
"drivers/char/Makefile" 77L, 2287C
```

杂项设备部分完全制作好了，只需要添加子设备，非常方便，在后面实验操作中大家就可以感受到。

这样杂项设备的引入即解决了设备号数量少的问题，又降低了使用难度，还能防止碎片化，一举多得。

## 8.4 杂项设备注册函数以及结构体

杂项设备的头文件在“include/linux/miscdevice.h”，有两个需要掌握的函数和一个结构体，如下图所示，在源码目录下使用命令“vim include/linux/miscdevice.h”。



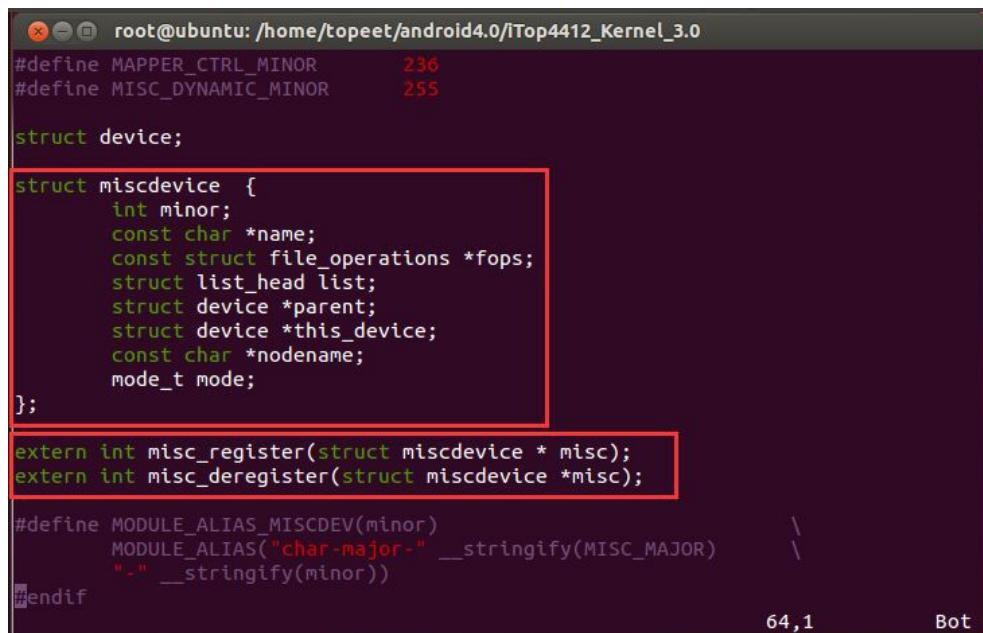
```
#ifndef _LINUX_MISCDEVICE_H
#define _LINUX_MISCDEVICE_H
#include <linux/module.h>
#include <linux/major.h>

/*
 *      These allocations are managed by device@lanana.org. If you use an
 *      entry that is not in assigned your entry may well be moved and
 *      reassigned, or set dynamic if a fixed value is not justified.
 */

#define PSMOUSE_MINOR          1
#define MS_BUSMOUSE_MINOR       2
#define ATIXL_BUSMOUSE_MINOR    3
/*#define AMIGAMOUSE_MINOR      4      FIXME OBSOLETE */
#define ATARIMOUSE_MINOR        5
#define SUN_MOUSE_MINOR         6
#define APOLLO_MOUSE_MINOR     7
#define PC110PAD_MINOR          9
/*#define ADB_MOUSE_MINOR       10      FIXME OBSOLETE */
#define WATCHDOG_MINOR          130     /* Watchdog timer */
#define TEMP_MINOR               131     /* Temperature Sensor */
#define RTC_MINOR                135

#include</linux/miscdevice.h> 64L, 1765C
```

如下图所示，到最底行。



```
#define MAPPER_CTRL_MINOR      236
#define MISC_DYNAMIC_MINOR      255

struct device;

struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    mode_t mode;
};

extern int misc_register(struct miscdevice * misc);
extern int misc_deregister(struct miscdevice *misc);

#define MODULE_ALIAS_MISCDEV(minor)
    MODULE_ALIAS("char-major-"
        __ stringify(MISC_MAJOR) \
        "-" __ stringify(minor))
#endif
```

如上图所示，红色框中有两个函数。

```
extern int misc_register(struct miscdevice * misc);
```

杂项设备注册函数；一般在 probe 中调用，参数是 miscdevice

```
extern int misc_deregister(struct miscdevice *misc);
```

杂项设备卸载函数；一般是在 hello\_remove 中用于卸载驱动。

结构体 miscdevice 中参数很多，下面几个是常用的。

int .minor；设备号，赋值为 MISC\_DYNAMIC\_MINOR，这个宏定义可以查到为 10

const char \*name;设备名称

const struct file\_operations \*fops；file\_operations 结构体，在下一小节专门介绍。

## 8.5 file\_operations 结构体

file\_operations 结构体的成员函数属于驱动设计的主体内容，里面的函数和 Linux 系统给应用程序提供系统接口一一对应。

file\_operations 结构体在头文件 “include/linux/fs.h” 中，如下图所示，使用命令 “vim include/linux/fs.h” 打开头文件。

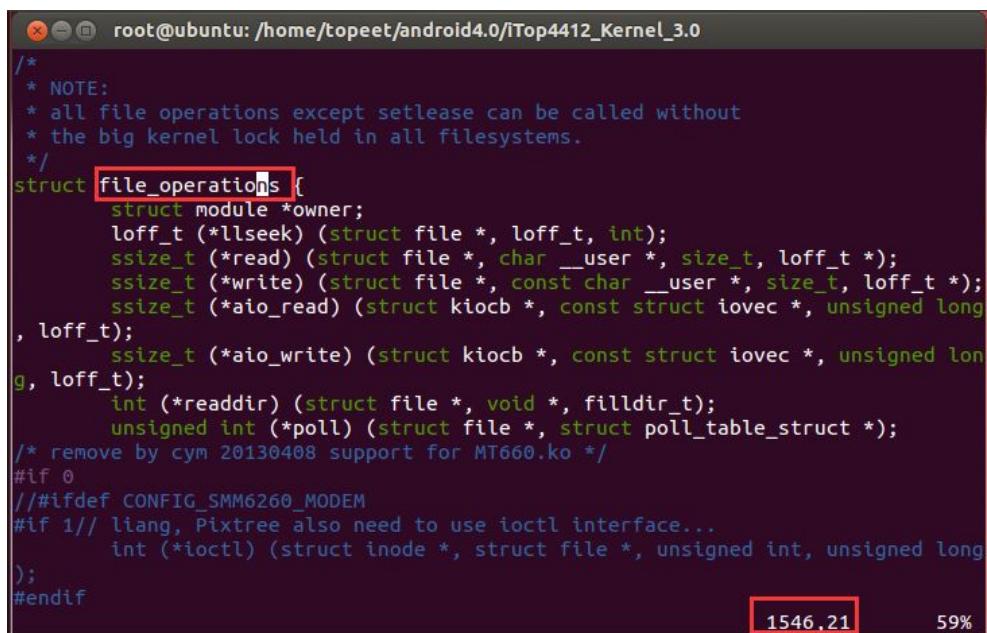
```
#ifndef _LINUX_FS_H
#define _LINUX_FS_H

/*
 * This file has definitions for some important file table
 * structures etc.
 */

#include <linux/limits.h>
#include <linux/ioctl.h>
#include <linux/blk_types.h>
#include <linux/types.h>

/*
 * It's silly to have NR_OPEN bigger than NR_FILE, but you can change
 * the file limit at runtime and only root can increase the per-process
 * nr_file rlimit, so it's safe to set up a ridiculously high absolute
 * upper limit on files-per-process.
 *
 * Some programs (notably those using select()) may have to be
 * recompiled to take full advantage of the new limits..
 */
#include/linux/fs.h" 2616L, 89651C           1,1           Top
```

查找 “file\_operations” ，如下图所示，在 1546 行可以找到其定义。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
/*
 * NOTE:
 * all file operations except setlease can be called without
 * the big kernel lock held in all filesystems.
 */
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long
, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long
, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
/* remove by cym 20130408 support for MT660.ko */
#endif
#ifndef CONFIG_SMM6260_MODEM
#if 1// liang, Pixtree also need to use ioctl interface...
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long
);
#endif

```

如上图所示，可以看到结构体中包含的参数非常多。

struct module \*owner;一般是 THIS\_MODULE。

int (\*open) (struct inode \*, struct file \*);对应上层的 open 函数，打开文件。

int (\*release) (struct inode \*, struct file \*);对应上层的 close 函数，打开文件操作之后一般需要关闭。

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);读函数，上层应用从底层读取函数。

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);写函数，上层应用向底层传输数据。

long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);这个函数功能和写函数稍微有点重合，但是这个函数占用的内存非常小，主要针对 IO 口的控制。

其它结构体中的参数，具体用到再介绍。

## 8.6 实验操作

下面来具体介绍如何写一个杂项设备。

将上一期中的 “probe\_linux\_module.c” 改写为 “devicenode\_linux\_module.c” ,然后添加头文件具体代码等。

如下图所示，将头文件 “linux/platform\_device.h” 、 “linux/miscdevice.h” 以及 “linux/fs.h” 添加到文件中。然后定义一个 DEVICE\_NAME , 将其赋值为"hello\_ctl123" , 帮助大家理解这个设备节点名称和前面介绍的注册设备名称是不同的。

```
#include <linux/init.h>
#include <linux/module.h>

/*驱动注册的头文件，包含驱动的结构体和注册和卸载的函数*/
#include <linux/platform_device.h>
/*注册杂项设备头文件*/
#include <linux/miscdevice.h>
/*注册设备节点的文件结构体*/
#include <linux/fs.h>

#define DRIVER_NAME "hello_ctl"
#define DEVICE_NAME "hello_ctl123"
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("TOPEET");
```

如下图所示 ,向添加 hello\_probe 中添加注册杂项设备的函数 misc\_register ,如下图所示 ,将 miscdevice 参数定义为 hello\_dev。

```
static int hello_probe(struct platform_device *pdev) {
    printk(KERN_EMERG "\tinitialized\n");
    misc_register(&hello_dev);

    return 0;
}

static int hello_remove(struct platform_device *pdev) {
    printk(KERN_EMERG "\tremove\n");
    misc_deregister(&hello_dev);
    return 0;
}
```

如下图所示，定义了 miscdevice hello\_dev。

参数 minor 为 MISC\_DYNAMIC\_MINOR，也就是 10

参数 name 为 DEVICE\_NAME，也就是 hello\_ctl123

参数 fops 为 “hello\_ops”

```
static struct miscdevice hello_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &hello_ops,
};
```

如下图所示，定义了 miscdevice hello\_dev，其中有四个参数。

参数 owner 为 THIS\_MODULE，

参数 open 为 hello\_open，

参数 release 为 hello\_release，

参数 unlocked\_ioctl 为 hello\_ioctl，

```
static struct file_operations hello_ops = {  
    .owner = THIS_MODULE,  
    .open = hello_open,  
    .release = hello_release,  
    .unlocked_ioctl = hello_ioctl,  
};
```

如下图所示，对 hello\_ops 结构体中的函数挨个定义。

```
static long hello_ioctl(struct file *file, unsigned int cmd, unsigned long arg){  
  
    printk("cmd is %d,arg is %d\n",cmd,arg);  
    return 0;  
}  
  
static int hello_release(struct inode *inode, struct file *file){  
    printk(KERN_EMERG "hello release\n");  
    return 0;  
}  
  
static int hello_open(struct inode, struct file *file){  
    printk(KERN_EMERG "hello open\n");  
    return 0;  
}  
  
static struct file_operations hello_ops = {  
    .owner = THIS_MODULE,  
    .open = hello_open,  
    .release = hello_release,  
    .unlocked_ioctl = hello_ioctl,  
};
```

如上图所示，函数中只做简单的打印。

在调用 hello\_ioctl 的时候打印 “cmd is %d,arg is %d” ，

在调用 hello\_release 的时候打印 “hello release” ，

在调用 hello\_open 的时候打印 “hello open” 。

将前一章实验的 Makefile 文件简单修改一下，将 “probe\_linux\_module.o” 改为

“devicenode\_linux\_module.o” , 注释部分用户可以自己修改 , 如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件itop4412_hello.o
obj-m += devicenode_linux_module.o

#源码目录变量, 这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)} Linux源码目录, 作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)} 当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules
    ...

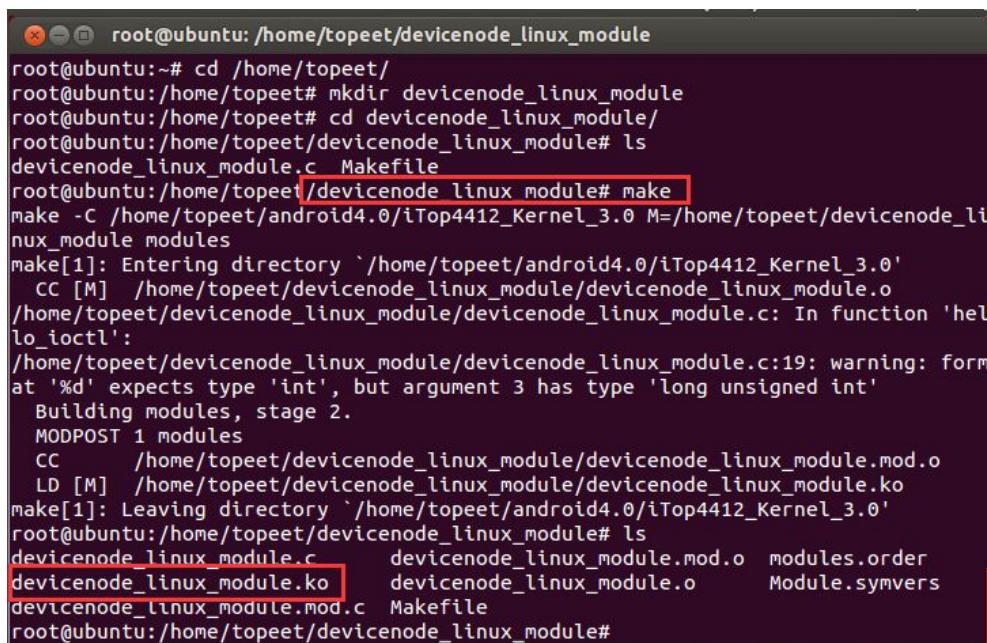
#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

在 Ubuntu 中的目录 “/home/topeet” 下新建 “devicenode\_linux\_module” 目录 , 拷贝驱动文件 “devicenode\_linux\_module.c” 和编译文件 “Makefile” 到新建目录中 , 如下图所示。

The screenshot shows a terminal window with a dark background and light-colored text. The command history is as follows:

```
root@ubuntu:~# cd /home/topeet/
root@ubuntu:/home/topeet# mkdir devicenode_linux_module
root@ubuntu:/home/topeet# cd devicenode_linux_module/
root@ubuntu:/home/topeet/devicenode_linux_module# ls
devicenode_linux_module.c Makefile
root@ubuntu:/home/topeet/devicenode_linux_module#
```

如下图所示，使用命令“make”，编译生成模块“devicenode\_linux\_module.ko”文件。



```
root@ubuntu:~# cd /home/topeet/devicenode_linux_module
root@ubuntu:/home/topeet# mkdir devicenode_linux_module
root@ubuntu:/home/topeet# cd devicenode_linux_module/
root@ubuntu:/home/topeet/devicenode_linux_module# ls
devicenode_linux_module.c  Makefile
root@ubuntu:/home/topeet/devicenode_linux_module# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/devicenode_linux_module modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M]  /home/topeet/devicenode_linux_module/devicenode_linux_module.o
  /home/topeet/devicenode_linux_module/devicenode_linux_module.c: In function 'hello_ioctl':
  /home/topeet/devicenode_linux_module/devicenode_linux_module.c:19: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
    Building modules, stage 2.
    MODPOST 1 modules
    CC      /home/topeet/devicenode_linux_module/devicenode_linux_module.mod.o
    LD [M]  /home/topeet/devicenode_linux_module/devicenode_linux_module.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/devicenode_linux_module# ls
devicenode_linux_module.c      devicenode_linux_module.mod.o  modules.order
devicenode_linux_module.ko     devicenode_linux_module.o       Module.symvers
devicenode_linux_module.mod.c  Makefile
root@ubuntu:/home/topeet/devicenode_linux_module#
```

启动开发板，拷贝“devicenode\_linux\_module.ko”到U盘，将U盘插入开发板，加载驱动文件“devicenode\_linux\_module.ko”，如下图所示。



```
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/devicenode_linux_module.ko
[ 4112.512749] HELLO WORLD enter!
[ 4112.514620] initialized
[ 4112.530912] DriverState is 0
[root@iTOP-4412]#
```

如下图所示，加载之后使用命令“ls /dev”，可以看到新生成了设备节点hello\_ctl123，也就是设备节点和驱动名以及设备名没有关系，不过最好设备节点的命名便于识别。

```
[root@iTOP-4412]# insmod /mnt/udisk/devicenode_linux_module.ko
[ 4112.512749] HELLO WORLD enter!
[ 4112.514620] initialized
[ 4112.530912] DriverState is 0
[root@iTOP-4412]# ls /dev/
AGPS          leds           ram10          tty1
adc           log            ram11          tty2
alarm         loop0          ram12          tty3
android_adb   loop1          ram13          tty4
ashmem        loop2          ram14          ttyGS0
bus           loop3          ram15          ttyGS1
buzzer_ctl   loop4          ram2           ttyGS2
console       loop5          ram3           ttyGS3
cpu_dma_latency loop6         ram4           ttyS0
exynos-mem   loop7         ram5           ttyS1
fb0           mapper        ram6           ttyS2
fb1           max485_ctl_pin ram7           ttyS3
fb2           mem            ram8           ttySAC0
fb3           mmcblk0        ram9           ttySAC1
fb4           mmcblk0p1      random         ttySAC2
full          mmcblk0p2      rc522          ttySAC3
fuse          mmcblk0p3      relay_ctl     uinput
hello_ctl123 mmcblk0p4      root           urandom
i2c-0          mtp_usb       rtc0           usb_accessory
i2c-1          network_latency  rtc1          usbdev1.1
i2c-3          network_throughput s3c-mem    usbdev1.2
i2c-4          null           sda            usbdev1.3
i2c-5          pmem           sda1          usbdev1.4
i2c-7          pmem_gpu1     sg0            watchdog
input          ppp            shm           xt_qtaguid
ion            ptmx          snd            zero
keychord      pts            srp
kmem          ram0          srp_ctrl
kmsg          ram1          tty
```

[root@iTOP-4412]#

如下图所示，使用命令“rmmod devicenode\_linux\_module”卸载驱动。

```
-----+
[root@iTOP-4412]# lsmod
devicenode_linux_module 1873 0 - Live 0xbff00000
[root@iTOP-4412]# rmmod devicenode_linux_module
[ 4312.465228] HELLO WORLD exit!
[ 4312.466824] remove
[root@iTOP-4412]#
```

如下图所示，使用命令“ls /dev”，发现设备节点 hello\_ctl123 已经去掉了。

```
[root@iTOP-4412]# ls /dev/
AGPS          leds           ram1           srp_ctrl
adc           log            ram10          tty
alarm         loop0          ram11          tty1
android_adb   loop1          ram12          tty2
ashmem        loop2          ram13          tty3
bus           loop3          ram14          tty4
buzzer_ctl   loop4          ram15          ttyGS0
console       loop5          ram2           ttyGS1
cpu_dma_latency loop6          ram3           ttyGS2
exynos-mem   loop7          ram4           ttyGS3
fb0           mapper         ram5           ttyS0
fb1           max485_ctl_pin ram6           ttyS1
fb2           mem            ram7           ttyS2
fb3           mmcblk0        ram8           ttyS3
fb4           mmcblk0p1      ram9           ttySAC0
full          mmcblk0p2      random         ttySAC1
fuse          mmcblk0p3      rc522          ttySAC2
i2c-0         mmcblk0p4      relay_ctl     ttySAC3
i2c-1         mtp_usb        root           uinput
i2c-3         network_latency rtc0           urandom
i2c-4         network_throughput rtc1          usb_accessory
i2c-5         null           s3c-mem       usbdev1.1
i2c-7         pmem           sda            usbdev1.2
input          pmem_gpu1     sda1          usbdev1.3
ion           ppp            sg0            usbdev1.4
keychord      ptmx          shm           watchdog
kmem          pts            snd            xt_qtaguid
kmsg          ram0           srp           zero
```

# 实验 09 编写简单应用调用驱动

## 9.1 本章导读

本期实验比较简单，就是写一个简单的应用程序调用前面写的驱动。

### 9.1.1 工具

#### 9.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 9.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端 ( 串口助手 )
- 4 ) 实验配套源码文件夹 “invoke\_hello”

### 9.1.2 预备课程

#### 实验 08\_生成设备节点

### 9.1.3 视频资源

本节配套视频为“视频 09-编写简单应用调用驱动”

## 9.2 学习目标

本章需要学习以下内容：

学会调用设备节点

## 9.3 实验操作

本期实验很简单，在前面 Linux 应用中就已经学习过设备节点的调用。

需要用到函数 `extern void printf(const char *format,...);` 定义在标准 C 语言头文件 `stdio.h` 中。

下面几个头文件在应用中一般一起调用。

头文件 `#include <sys/types.h>` 包含基本系统数据类型。系统的基本数据类型在 32 编译环境中保持为 32 位值，并会在 64 编译环境中增长为 64 位值。

头文件 `<sys/stat.h>` 包含系统调用文件的函数。可以调用普通文件、目录、管道、socket、字符、块的属性。

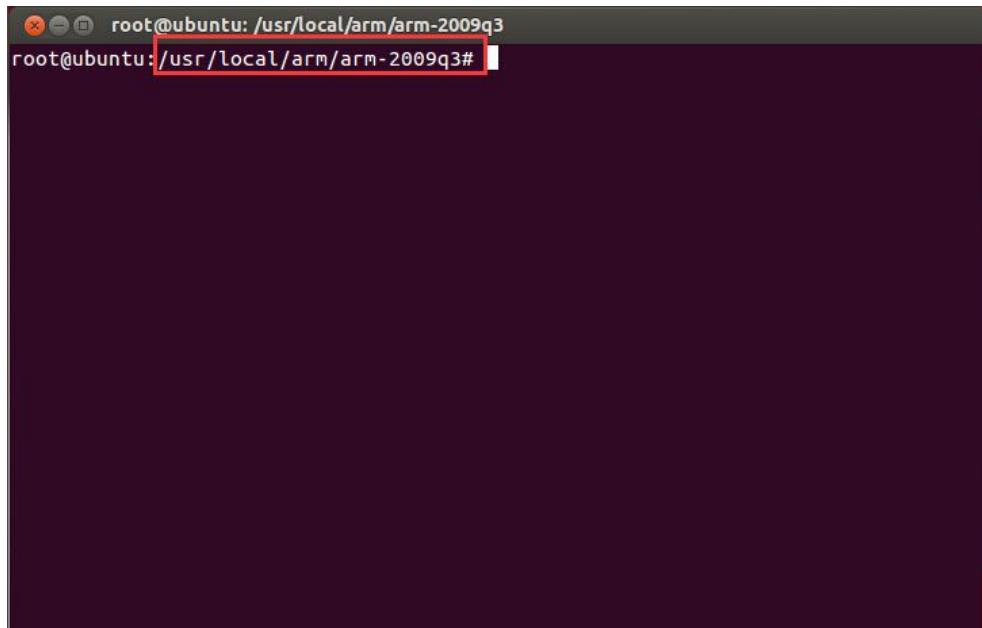
`<fcntl.h>` 定义了 `open` 函数

`<unistd.h>` 定义了 `close` 函数

`<sys/ioctl.h>` 定义了 `ioctl` 函数

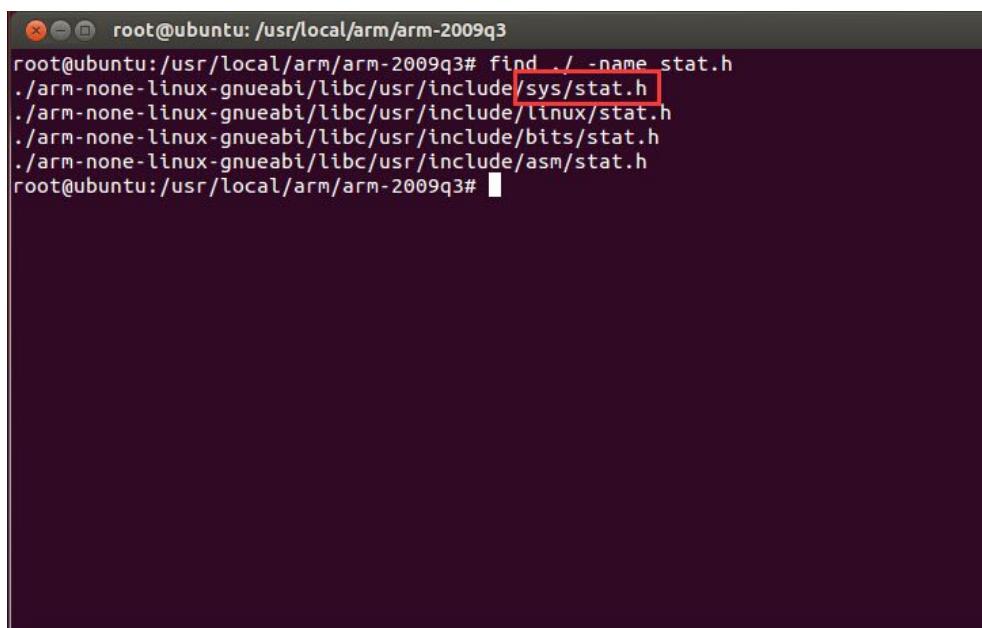
另外提醒一下，这些头文件是和编译器在一起。

这里使用，如下图所示，进入目录 “/usr/local/arm/arm-2009q3” 。



```
root@ubuntu:/usr/local/arm/arm-2009q3#
```

使用查找命令 “find ./ -name stat.h” ,如下图所示，使用的头文件是目录 “/arm-none-linux-gnueabi/libc/usr/include/sys/stat.h” 中的<sys/stat.h>.



```
root@ubuntu:/usr/local/arm/arm-2009q3# find ./ -name stat.h
./arm-none-linux-gnueabi/libc/usr/include/sys/stat.h
./arm-none-linux-gnueabi/libc/usr/include/linux/stat.h
./arm-none-linux-gnueabi/libc/usr/include/bits/stat.h
./arm-none-linux-gnueabi/libc/usr/include/asm/stat.h
root@ubuntu:/usr/local/arm/arm-2009q3#
```

其它几个头文件可以采用类似的方法查找，这里给大家提醒这一点，因为有时候拿到源码之后，可能编译器版本和源码不完全对应，这个时候就有可能需要修改和处理一下头文件。不过这种问题一般都可以通过网络查找错误提示的方法一个一个解决。

如下图所示，是一个简单的调用程序。

```
#include <stdio.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

main() {
    int fd;
    char *hello_node = "/dev/hello_ctl123";

    /*O_RDWR只读打开,O_NDELAY非阻塞方式*/
    if((fd = open(hello_node,O_RDWR|O_NDELAY))<0) {
        printf("APP open %s failed",hello_node);
    }
    else{
        printf("APP open %s success",hello_node);
        ioctl(fd,1,6);
    }

    close(fd);
}
```

新建“invoke\_hello”文件夹，将上图的中的文件拷贝进入，进入新建的“invoke\_hello”目录，使用编译命令

“arm-none-linux-gnueabi-gcc -o invoke\_hello invoke\_hello.c -static”

编译，如下图所示。

```
root@ubuntu:~/home/topeet/invoke_hello
root@ubuntu:~# cd /home/topeet/
root@ubuntu:/home/topeet# mkdir invoke_hello
root@ubuntu:/home/topeet# cd invoke_hello/
root@ubuntu:/home/topeet/invoke_hello# ls
invoke_hello.c
root@ubuntu:/home/topeet/invoke_hello# arm-none-linux-gnueabi-gcc -o invoke_hello invoke_hello.c -static
root@ubuntu:/home/topeet/invoke_hello# ls
invoke_hello invoke_hello.c
root@ubuntu:/home/topeet/invoke_hello#
```

将 “invoke\_hello” 拷贝到 U 盘 ,启动开发板 ,加载前一期的 “devicenode\_linux\_module” 驱动 ,如下图所示 ,使用 invoke\_hello 调用设备节点 “/dev/hello\_ctl123” 。

先使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘 ;

使用命令 “insmod /mnt/udisk/devicenode\_linux\_module.ko” 加载驱动 ;

使用命令 “./mnt/udisk/invoke\_hello” ,运行 invoke\_hello。

```
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/devicenode_linux_module.ko
[    32.363393] HELLO WORLD enter!
[    32.365480] initialized
[    32.380682] DriverState is 0
[root@iTOP-4412]# [    33.206858] CPU3: shutdown
[root@iTOP-4412]# ./mnt/udisk/in
int_helloworld      invoke_char_gpios  invoke_hello
invoke_char_driver  invoke_qpios       invoke_leds
[root@iTOP-4412]# ./mnt/udisk/invoke_hello
[    49.050442] hello open
[    49.056632] cmd is 1,arg is 6
[    49.058159] hello release
APP open /dev/hello_ctl123 success[root@iTOP-4412]#
```

如上图所示 ,运行 “invoke\_hello” 之后 ,会打印以下内容”

hello open

cmd is 1,arg is 6

hello release

如下图所示，设备节点 open、 close、 ioctl 分别对应打印信息

```
 printk(KERN_EMERG "hello open\n");
```

```
 printk(KERN_EMERG "hello release\n");
```

printf("cmd is %d,arg is %d\n",cmd,arg);ioctl 会打印第二个和第三个参数。

```
 static long hello_ioctl( struct file *files, unsigned int cmd, unsigned long arg) {  
     printk("cmd is %d,arg is %d\n",cmd,arg);  
     return 0;  
 }  
  
 static int hello_release(struct inode *inode, struct file *file){  
     printk(KERN_EMERG "hello release\n");  
     return 0;  
 }  
  
 static int hello_open(struct inode *inode, struct file *file){  
     printk(KERN_EMERG "hello open\n");  
     return 0;  
 }
```

通过前面的分析，可以看到上层应用对设备节点 open、 close、 ioctl 分别对应驱动层的 open、 release、 unlocked\_ioctl。

# 实验 10-11 原理图的使用

## 11.1 本章导读

驱动工程师的知识需要的杂一些，本节实验要介绍的就是硬件的知识，这部分内容并没有想象的那么难，只要学习了原理图的整个框架，学会如何使用这个文档，剩下的硬件知识就可以在实际应用中一步一步积累，主要是掌握框架和方法。

### 11.1.1 工具

#### 11.1.1.1 硬件工具

PC 机一台

#### 11.1.1.2 软件工具

Adobe 或者其他 pdf 阅读软件

### 11.1.2 预备课程

无

### 11.1.3 视频资源

本节配套视频为“视频 10\_驱动工程师硬件知识\_基础概念”

本节配套视频为“视频 11\_驱动工程师硬件知识\_原理图的使用”

## 11.2 学习目标

本章需要学习以下内容：

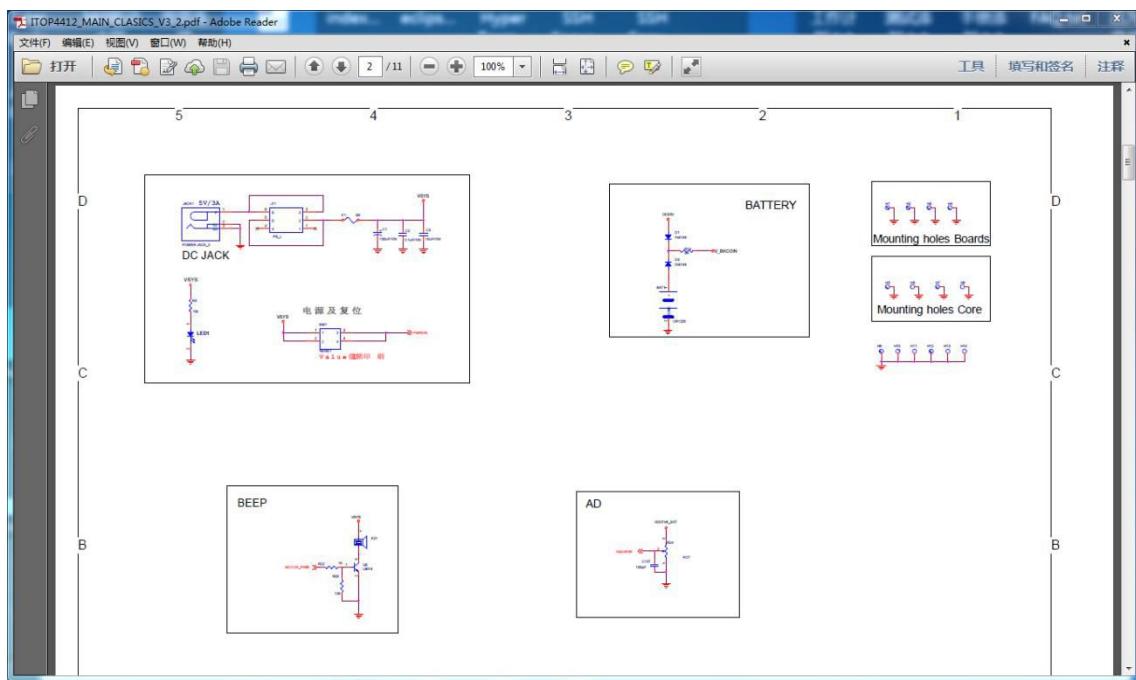
原理图的使用

## 11.3 原理图 PDF 的操作简介

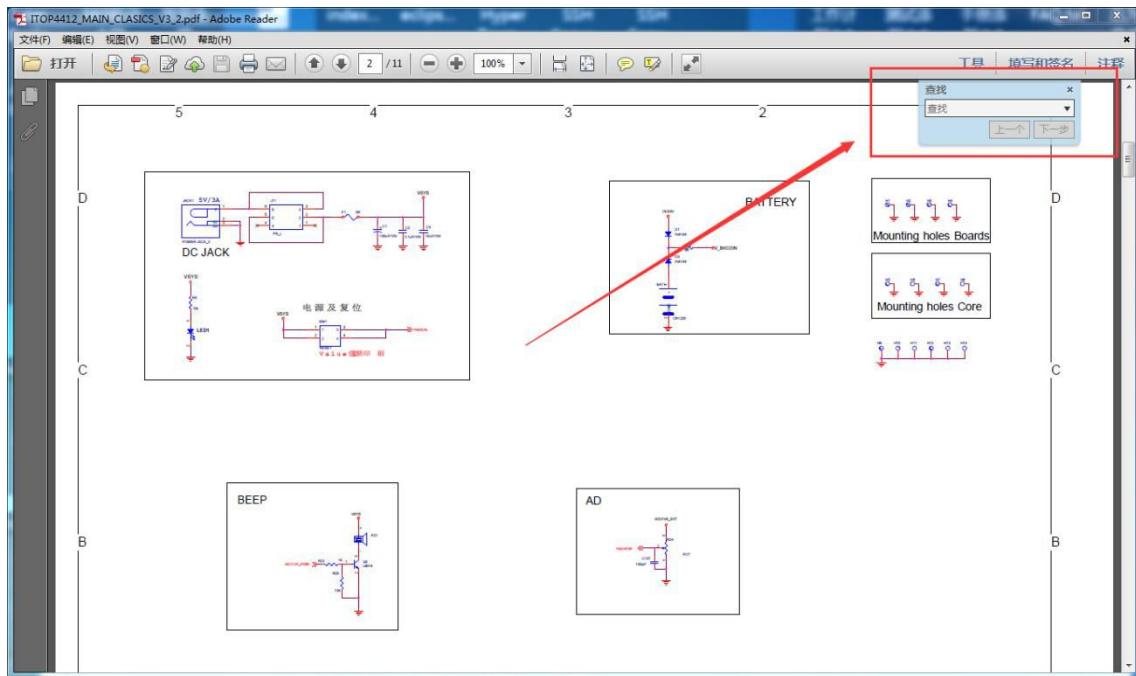
如下图所示，在光盘目录文件夹“原理图”提供了底板的原理图和 PCB，原理图有 pdf、Orcad 格式、AD09 格式的。



这里以 pdf 的为例介绍，驱动工程师并不需要掌握原理图和 PCB 软件。打开 pdf 格式原理图，如下图所示。



如上图所示，最常用的操作就是查找，键盘“Ctrl+f”可以启动查找功能，如下图所示。



如上图所示，可以在红色框中输入需要查找的内容，例如模块、元件标号、网络标号。

其它格式的原理图需要经过其它专业的训练，无论是 AD 还是 Orcad，都需要额外的学习，这个也超出了驱动学习的范围，如果感兴趣可以找相关的教程学习一下。

在网盘中给大家提供了于博士的 orcad 的教程。在“嵌入式学习推荐书籍及软件( 第三方 )”中有“于博士 Allegro 视频”，这是很经典的学习教程。

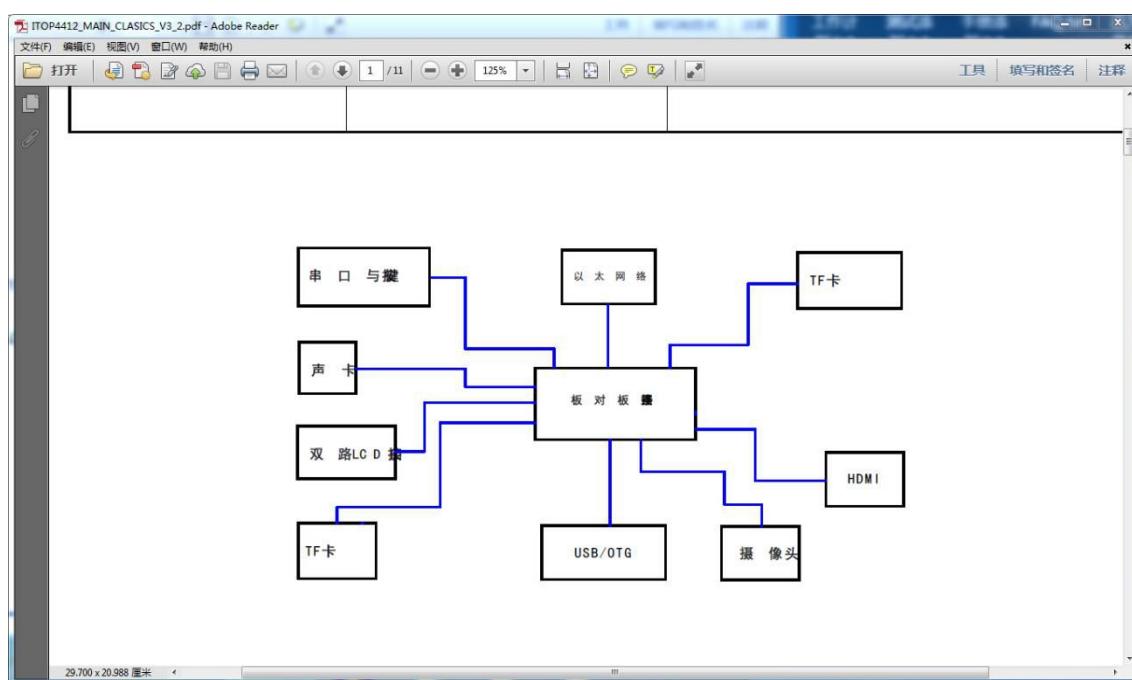
如果学习 AD，就简单一些，在网上买一本介绍操作的书来看一下就可以。

## 11.4 原理图——模块

拿到任何一个 PCB 的原理图之后，都需要对 PCB 的整体有个了解。

原理图工程师在设计前也是需要进行很长时间的准备工作，例如选型、评估、采购联系供应商等等。

经过反复讨论研究之后，原理图在出来之前就会有一个基本的功能框架，这些基本的功能框架就是模块，这些如下图所示。



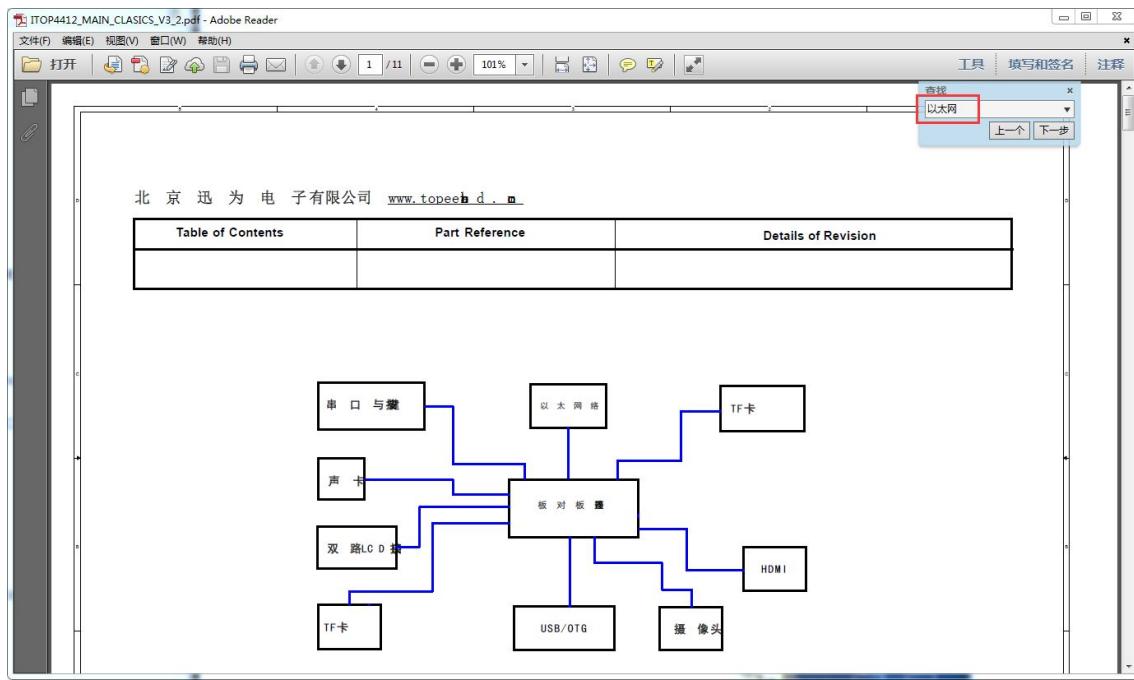
这里提醒一下，现在很多功能都会集成在一个外围芯片上，这种芯片也可以叫做模块，不过从原理图的角度分析，外围芯片加上其配套电路才能称为原理图的一个模块。

如上图所示，一般第一页就是的主要功能就是显示这些模块框图。

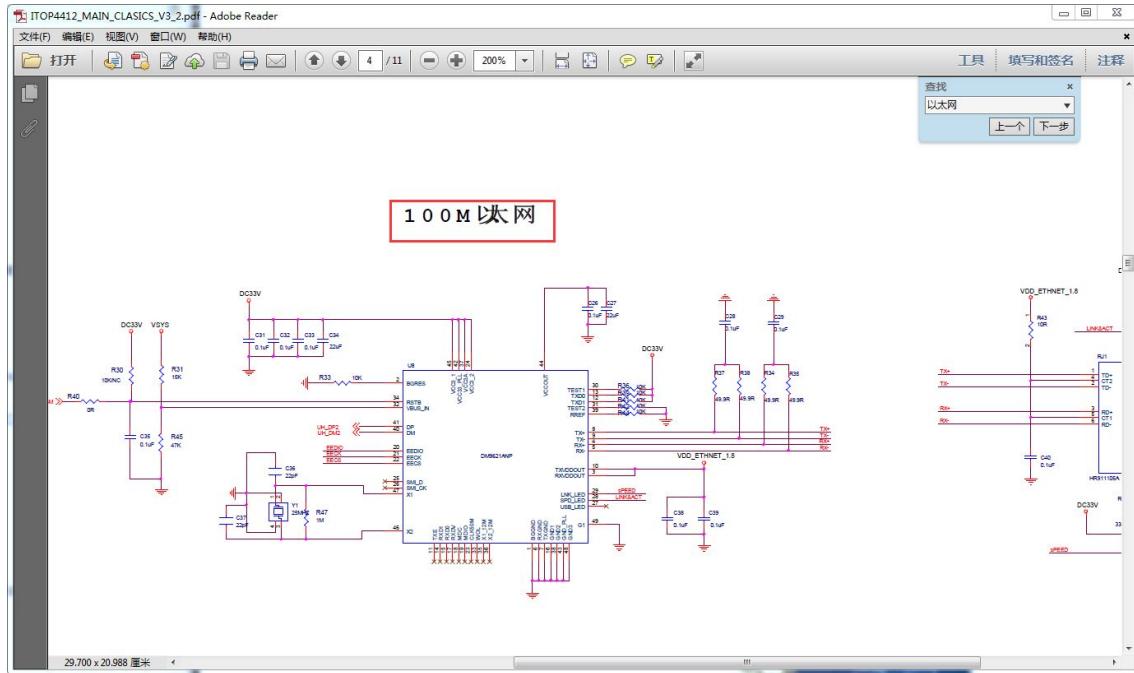
如下图所示，在对应的开发板使用手册中 1.1.2 小节中有接口说明。



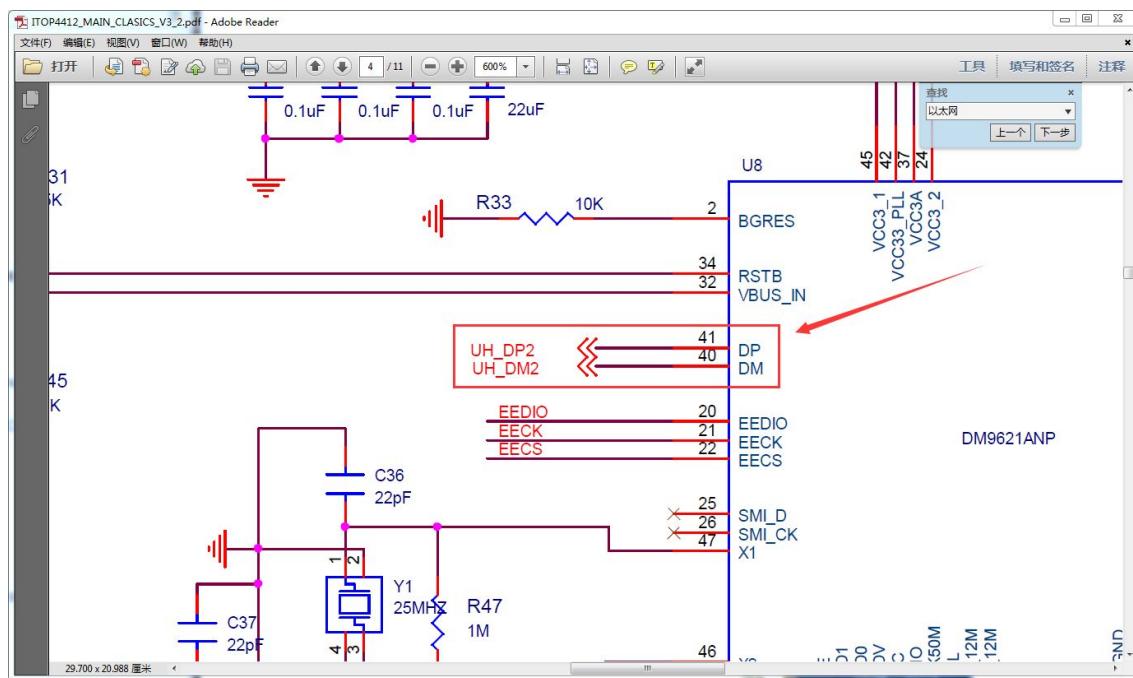
对模块有个整体的了解之后，例如想查找以太网部分。



如下图所示，就可以很快查找到。



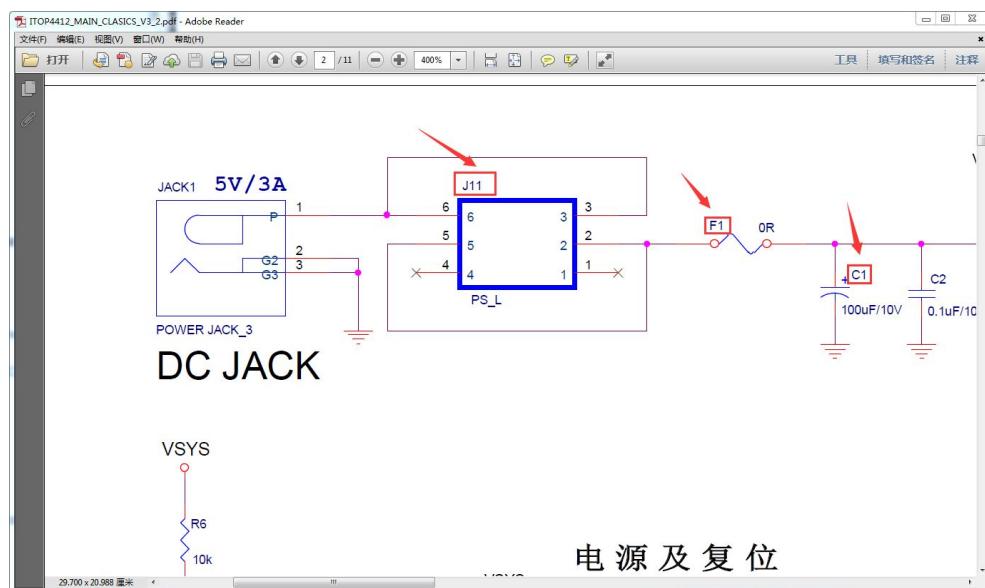
如下图所示，可以看到以太网和 4412 使用的通信接口是 USB。



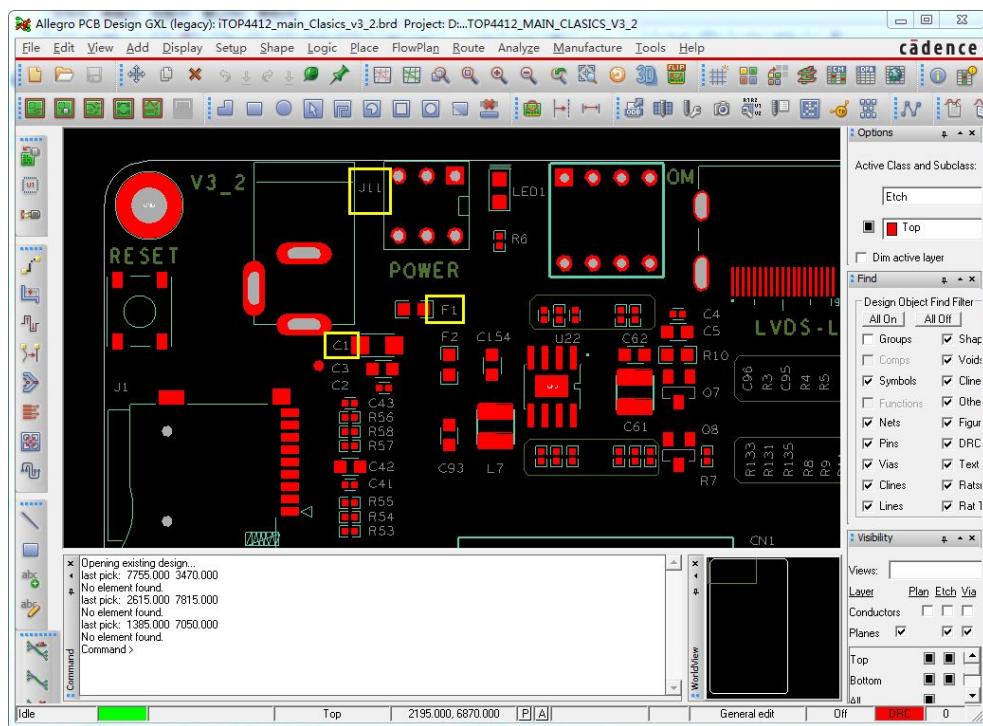
这一部分靠平时积累，掌握了方法之后，需要使用或者调试哪部分就可以去迅速查看对应的内容。

## 11.5 原理图——元件标号

如下图所示，电源输入部分，每一个元件都有具体的标号。

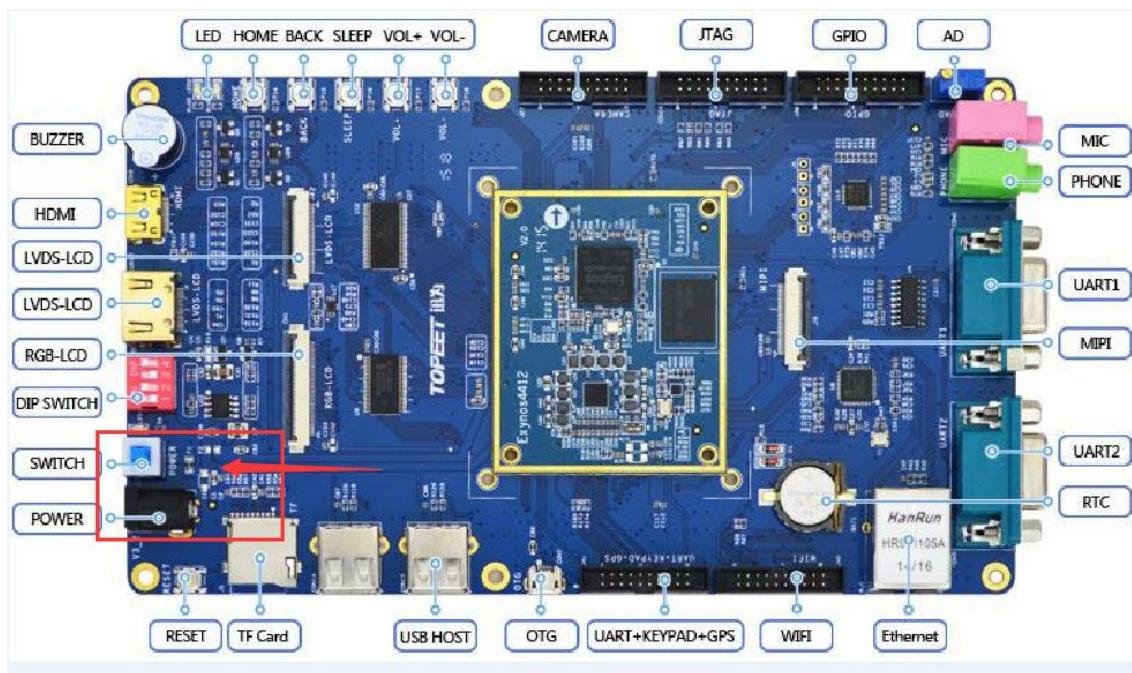


这一部分和 PCB 的标号一一对应。



如上图所示，如果会简单的使用 layout 软件，可以直接通过 PCB 查找对应的元件在 PCB 板上位置，如果不知道也没关系。

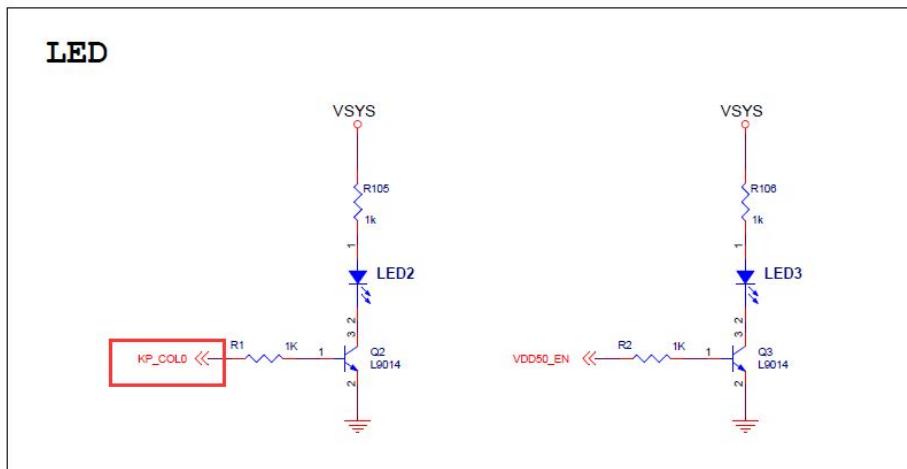
如下图所示，可以直接在 PCB 板上找到对应的元件，红色框中就是上面对应的电源输入部分。如果需要用到其它部分，也可以使用类似的方法找到实际的元器件。



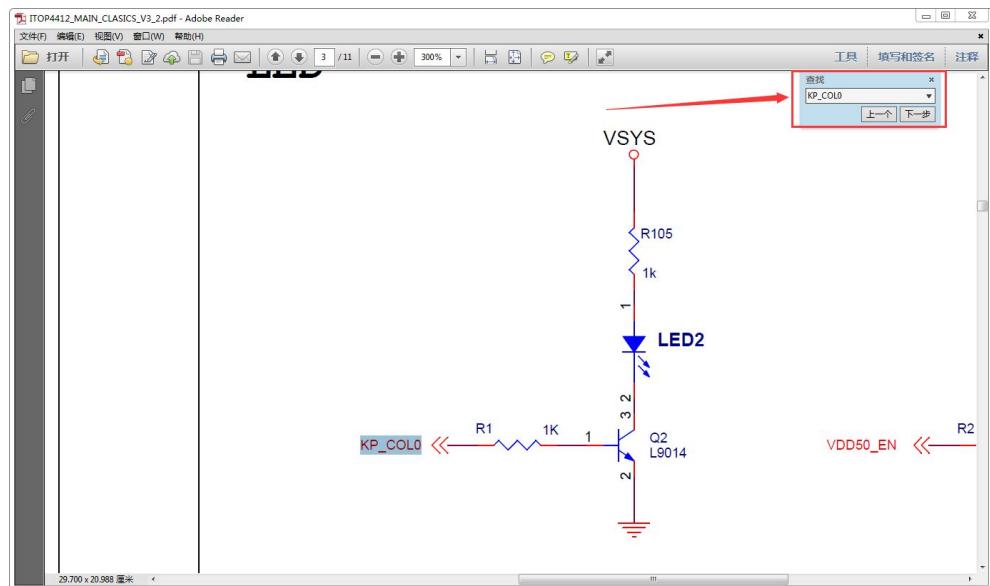
## 11.6 原理图——网络标号

网络标号这部分是最重要的，在学习驱动过程中会经常用到。

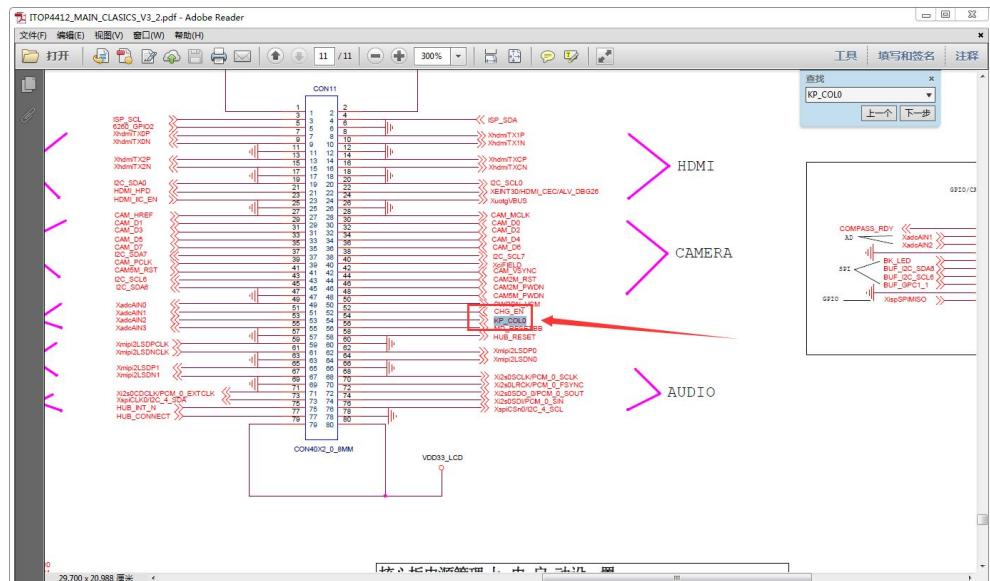
如下图所示，以 LED 为例，控制网络为 KP\_COL0。



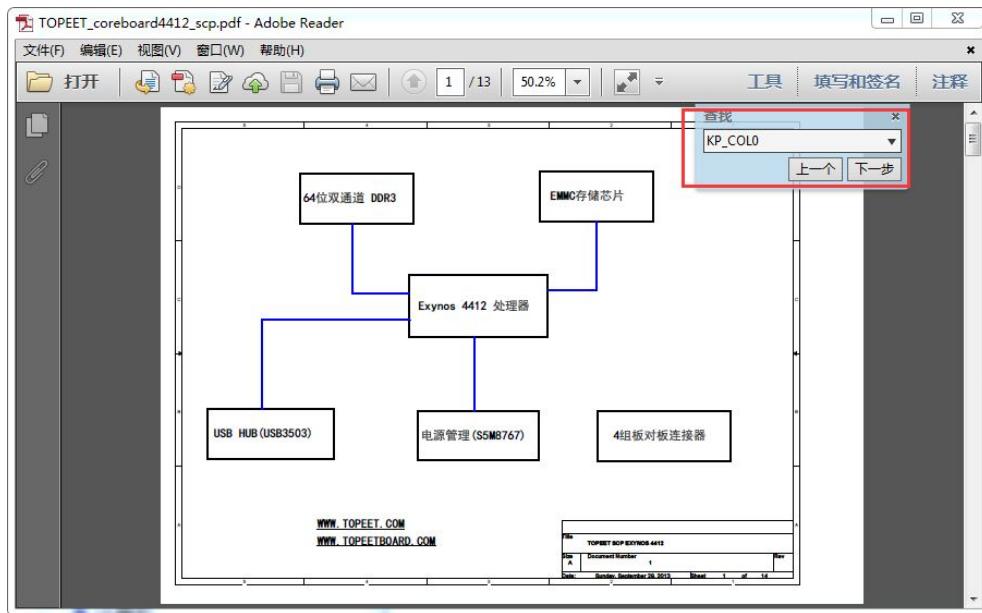
如下图所示，查找 KP\_COL0。



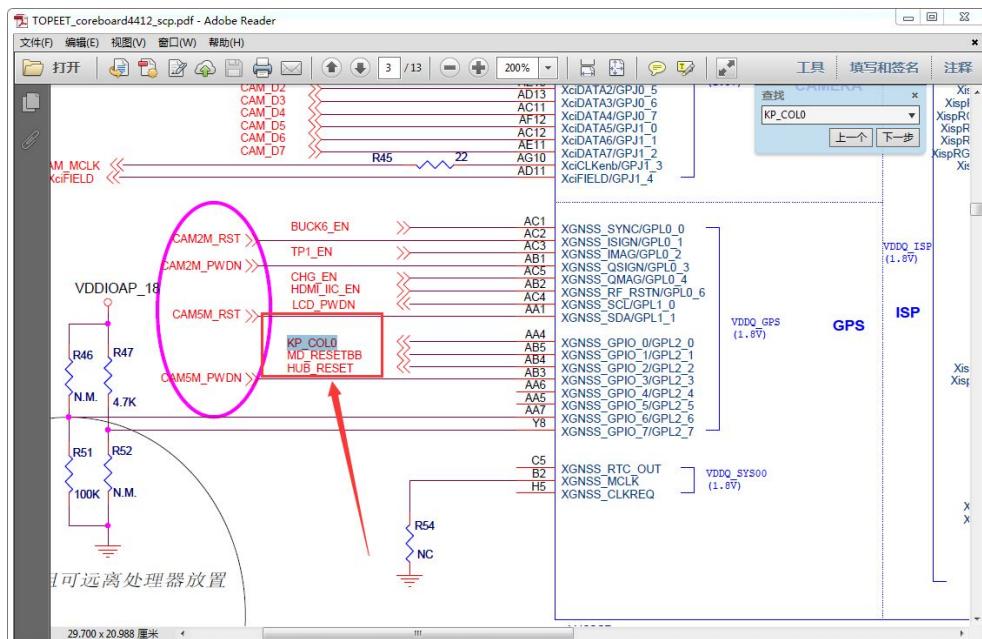
如下图所示，查找到核心板连接器上对应的网络，底板通过连接器和核心板连接。



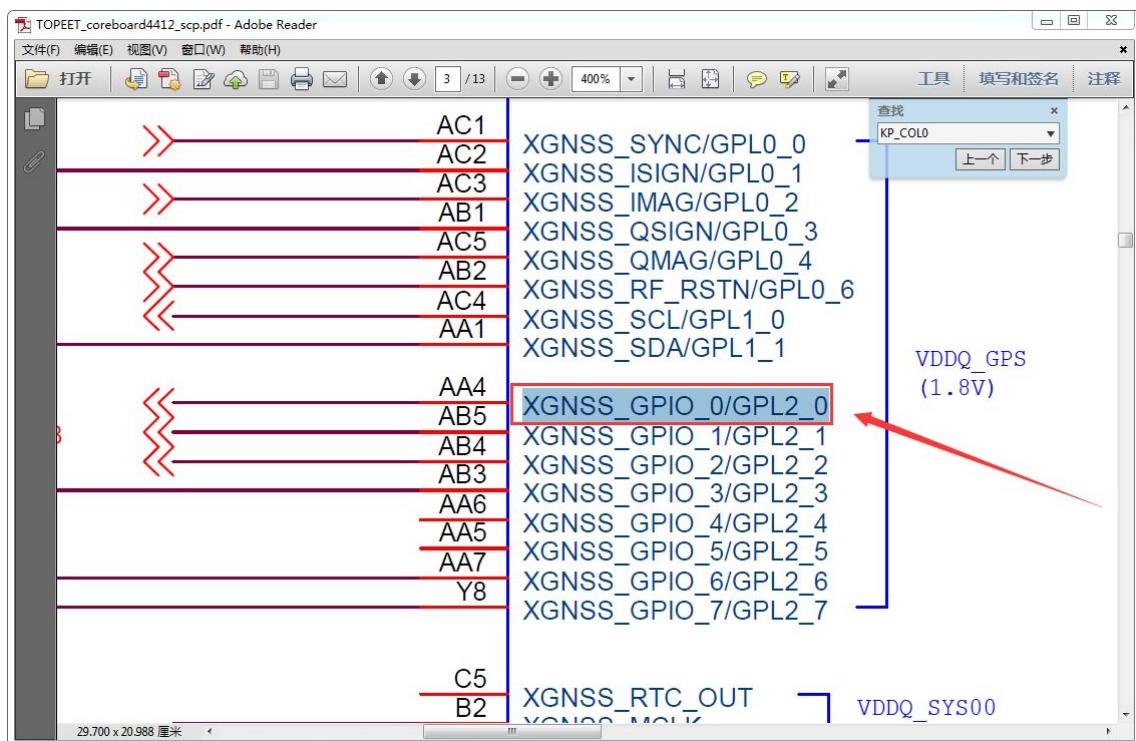
打开核心板 pdf 格式的原理图，查找对应网络 KP\_COL0。



如下图所示，可以查到对应4412芯片的管脚AA4。



如下图所示，可以看到管脚说明“XGNSS\_GPIO\_0/GPL2\_0”



# 实验 12-13 物理地址虚拟地址以及 GPIO 初始化

## 12.1 本章导读

Linux 系统中对 IO 的操作，专门设计了一套函数，通过这些函数就可以访问 IO，但是这些函数在控制 GPIO 相关寄存器的时候，要用到 MMU 内存管理单元。所以必须先理解内存管理单元，才能理解 Linux 如何控制 GPIO，本实验给大家介绍物理地址虚拟地址就是属于内存管理单元的内容。

### 12.1.1 工具

#### 12.1.1.1 硬件工具

1 ) PC 机

#### 12.1.1.2 软件工具

1 ) 虚拟机 Vmware

2 ) Ubuntu12.04.2

### 12.1.2 预备课程

无

### 12.1.3 视频资源

本节配套视频为“视频 12\_物理地址虚拟地址”

本节配套视频为“视频 13\_GPIO 初始化”

## 12.2 学习目标

本章需要学习以下内容：

了解物理地址

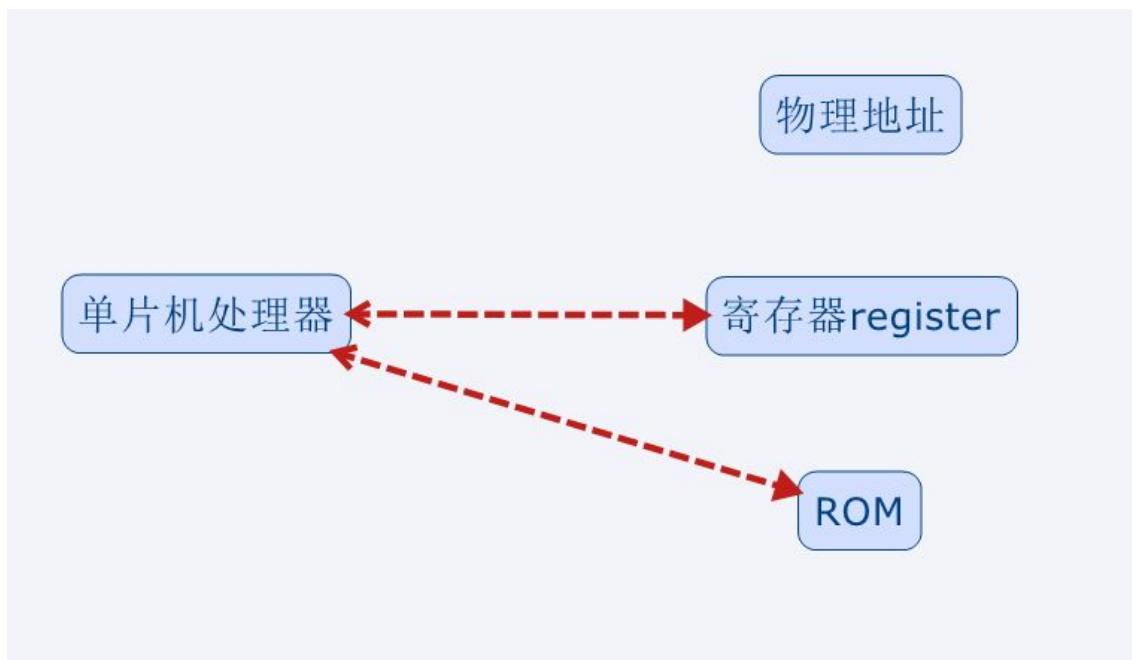
了解虚拟地址

了解内存管理单元

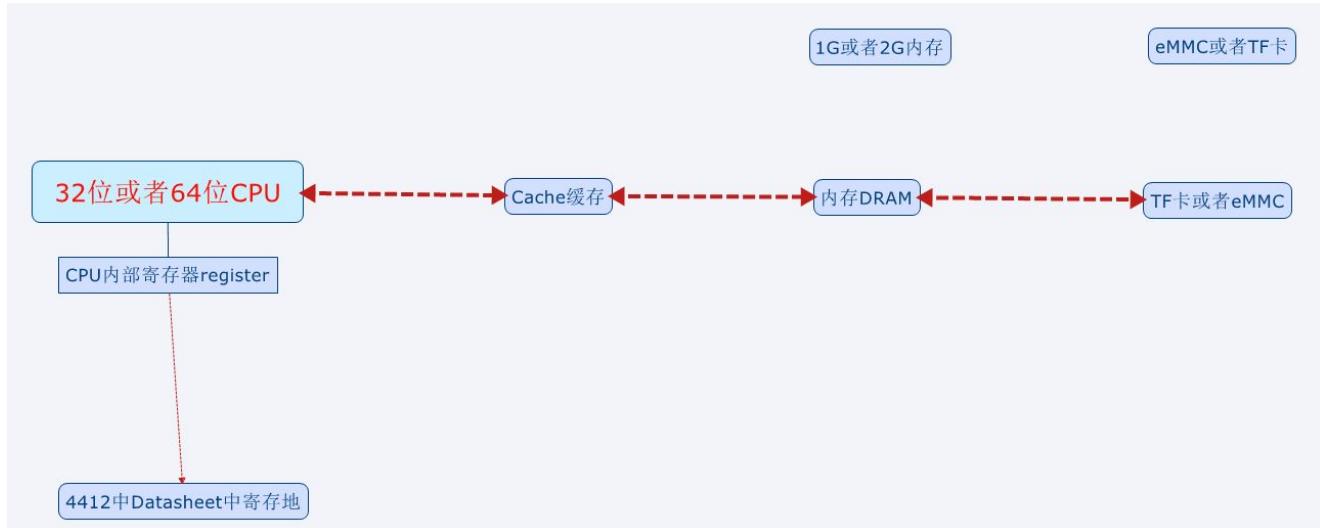
## 12.3 单片机处理器和现代处理器

在学习单片机的时候，都是可以直接对寄存器和地址操作。

如下图所示，在单片机的世界中，存储器结构比较简单，一般就分为寄存器 Reg 和一个只读存储器 ROM。



如下图所示，是现代中央处理器存储器。其中的 DRAM 就是非常熟悉的内存，32 位处理器一般最大内存是 4G。Cache 是高速缓存，用于解决内存和 CPU 之间速度不匹配的问题。ROM 的种类也是繁多，TF 卡、硬盘、eMMC、flash 等等。



如上图所示，其中 CPU 的内部寄存器是和 4412Datasheet 中的寄存器一一对应，在单片机中一般是直接对寄存器进行操作，但是在可以运行带有操作系统的 CPU 中，例如 linux，就不太可能直接进行操作，首先是会对地址进行宏定义，只需要调用已经写好的函数对这些宏定义进行操作，就相当于操作寄存器。

## 12.4 MMU 内存管理单元

如果理解了本节的内存管理单元，那么前面很多疑惑的地方都可以明白。

MMU 是中央处理用来管理虚拟内存、物理存储器的控制线路，同时将虚拟地址映射为物理地址。

先介绍一下 MMU 的由来。

在最初的 PC 上 , 内存都非常小 , 如果程序小于内存 , 也还是可以运行的。

随着应用程序的出现 , 出现了一个程序占用的空间大于内存 , 例如 1M 内存的 PC , 想要运行一个 4M 的程序 , 那么就有一个难题摆在面前了。程序员通常把程序分割为很多个小块 , 然后一个小块一个小块的运行 , 虽然调用是有操作系统解决的 , 但是分割却必须由程序员完成 , 这个过程费时费力 , 很无聊。人们找到了一个解决办法 , 那就是虚拟存储器。

虚拟存储器的基本思想是程序 , 数据 , 堆栈的总的大小可以超过物理存储器的大小 , 操作系统把当前使用的部分保留在内存中 , 而把其他未被使用的部分保存在磁盘上。比如对一个 16MB 的程序和一个内存只有 4MB 的机器 , 操作系统通过选择 , 可以决定各个时刻将哪 4M 的内容保留在内存中 , 并在需要时在内存和磁盘间交换程序片段 , 这样就可以把这个 16M 的程序运行在一个只具有 4M 内存机器上了 , 而这个 16M 的程序在运行前不必由程序员进行分割。

( 如果感兴趣可以找关于现代操作系统的书来看看 ) 。

## 12.5 物理地址虚拟地址以及 GPIO 的初始化

这部分内容可以参考配套的视频来学习。或者通过网络上的资料来理解这几个知识点。

# 实验 14 LEDS 驱动一

## 14.1 本章导读

本节实验介绍一个完整的 GPIO 驱动，以后在 Linux 中需要处理 GPIO 驱动都可以仿照或者移植这个驱动。

### 14.1.1 工具

#### 14.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 14.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端 ( 串口助手 )
- 4 ) 源码文件夹 “leds”

### 14.1.2 预备课程

#### 实验 12\_物理地址虚拟地址

## 实验 13\_GPIO 初始化

### 14.1.3 视频资源

本节配套视频为“视频 14 LEDS 驱动一”

## 14.2 学习目标

本章需要学习以下内容：

Led 硬件原理简单介绍

Led 管脚的调用、赋值以及配置

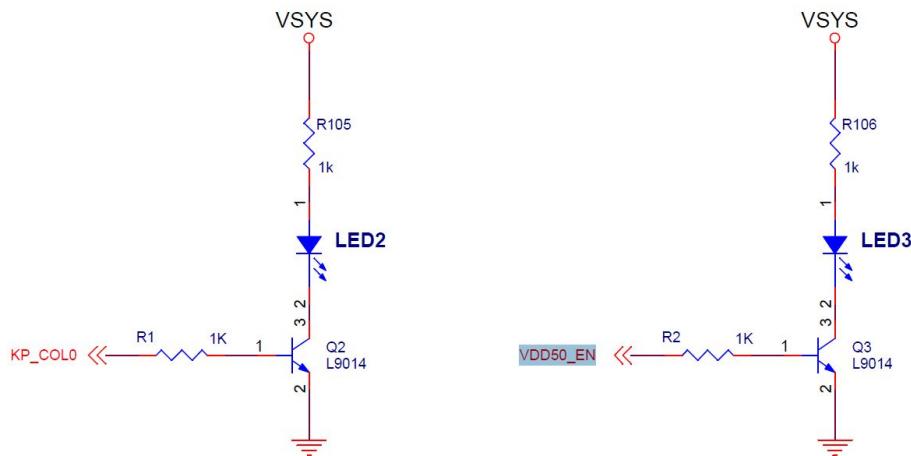
编写简单应用调用 LED 管脚，并测试

### 14.3 Led 硬件原理简单介绍

Led 的电路比较简单，一般是使用三极管搭建一个控制电路。

如下图所示，是原理图中两个 Led 的控制电路。KP\_COL0 和 VDD50\_EN 网络控制 Led 的通断。

## LED



如上图所示。

当 KP\_COL0 和 VDD50\_EN 网络时高电平的时候，三极管 L9014 的 BE 导通，CE 导通，相当于 5V 的 VSYS 电压加到 1K 和 Led 小灯上，小灯就会亮。

当 KP\_COL0 和 VDD50\_EN 网络时低电平的时候，三极管 L9014 的 BE 会截止，CE 截止，相当于 5V 的 VSYS 电压加到 1K、Led 小灯和一个无限大的电阻上，电流为零，小灯就会灭。

## 14.4 Led 管脚的调用、赋值以及配置

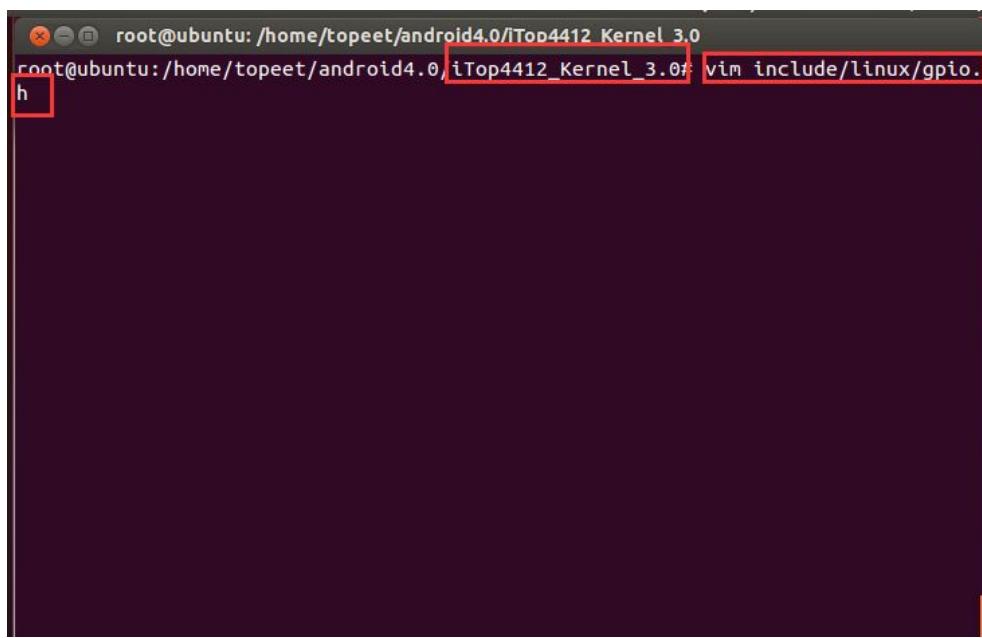
本节给大家介绍一部分涉及 GPIO 调用、赋值以及配置的函数。

### 14.4.1 GPIO 申请和释放函数

想用使用任何一个 GPIO 都必须先申请。

在头文件 “include/linux/gpio.h” 中有 Linux 默认的 GPIO 申请函数，这个头文件是属于嵌入式 Linux 平台，任何一个嵌入式 Linux 内核都可以这么使用。

如下图所示，在源码目录中使用命令 “vim include/linux/gpio.h” 打开该文件。



如下图所示，就是本节实验中需要用到的函数 gpio\_request。

A screenshot of a terminal window titled 'root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0'. The user is viewing the file 'gpio.h'. The code shown includes several functions: gpio\_is\_valid, gpio\_request, gpio\_request\_one, gpio\_request\_array, and gpio\_free. The 'gpio\_request' function is highlighted with a red rectangle. The terminal has a dark background with white text. At the bottom right, there are status indicators: '60,1' and '25%'.

如上图所示，简单介绍一下 gpio\_request 函数。

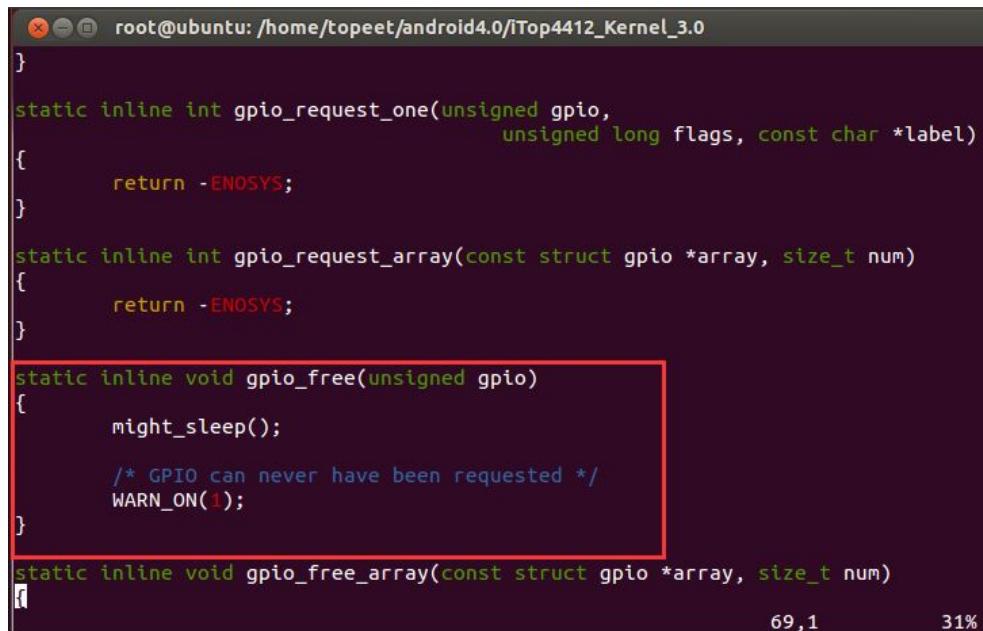
首先这个函数有一个重要的“检测”功能，就是如果其它地方申请了这个 IO，那么这里就会返回错误，提示已经被占用了，这是 Linux 中的一个标准用法。

gpio\_request 函数有两个参数

unsigned gpio , 申请的那个 GPIO , 一般是 GPIO 对应的宏定义

const char \*label , 为 GPIO 取个名字 , 便于阅读

如下图所示 , 和 gpio\_request 函数对应的是 gpio\_free 函数。



The screenshot shows a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The code displayed is part of the kernel's GPIO driver. A red box highlights the gpio\_free function. The code is as follows:

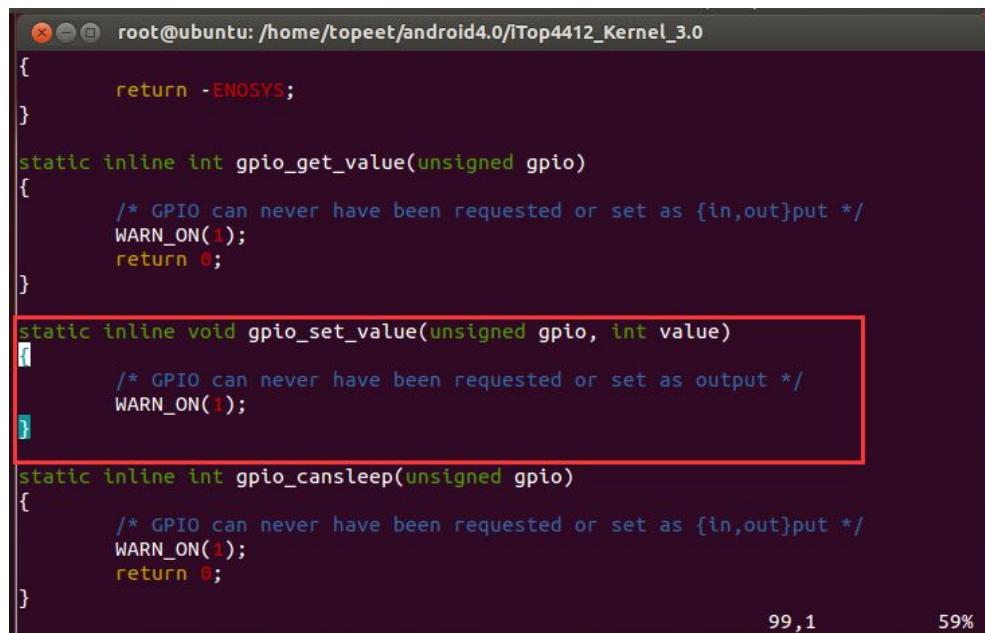
```
static inline void gpio_free(unsigned gpio)
{
    might_sleep();
    /* GPIO can never have been requested */
    WARN_ON(1);
}
```

At the bottom right of the terminal window, it says "69,1 31%".

在调用 gpio\_request 函数之后 , 向系统表明这个 IO 已经被占用了 , 在卸载驱动的时候一般需要调用 gpio\_free 函数将其释放。

gpio\_free 函数的参数比较简单 , 只有一个 GPIO 参数 , 使用 GPIO 对应的宏定义即可。

如下图所示 , 还有一个赋值函数 gpio\_set\_value。



The screenshot shows a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The code displayed is part of the Linux kernel's GPIO driver. It includes three functions: `gpio_get_value`, `gpio_set_value`, and `gpio_cansleep`. The `gpio_set_value` function is highlighted with a red rectangle. The code uses `WARN_ON(1)` to indicate that GPIO can never have been requested or set as {in,out}put.

```
{  
    return -ENOSYS;  
}  
  
static inline int gpio_get_value(unsigned gpio)  
{  
    /* GPIO can never have been requested or set as {in,out}put */  
    WARN_ON(1);  
    return 0;  
}  
  
static inline void gpio_set_value(unsigned gpio, int value)  
{  
    /* GPIO can never have been requested or set as output */  
    WARN_ON(1);  
}  
  
static inline int gpio_cansleep(unsigned gpio)  
{  
    /* GPIO can never have been requested or set as {in,out}put */  
    WARN_ON(1);  
    return 0;  
}
```

在将 GPIO 配置为输出模式之后，还需要给 GPIO 赋值，一般就是高电平和低电平两种。

两个参数分别为

`unsigned gpio` , `GPIO`

`int value` , 高电平 1 和低电平 0。

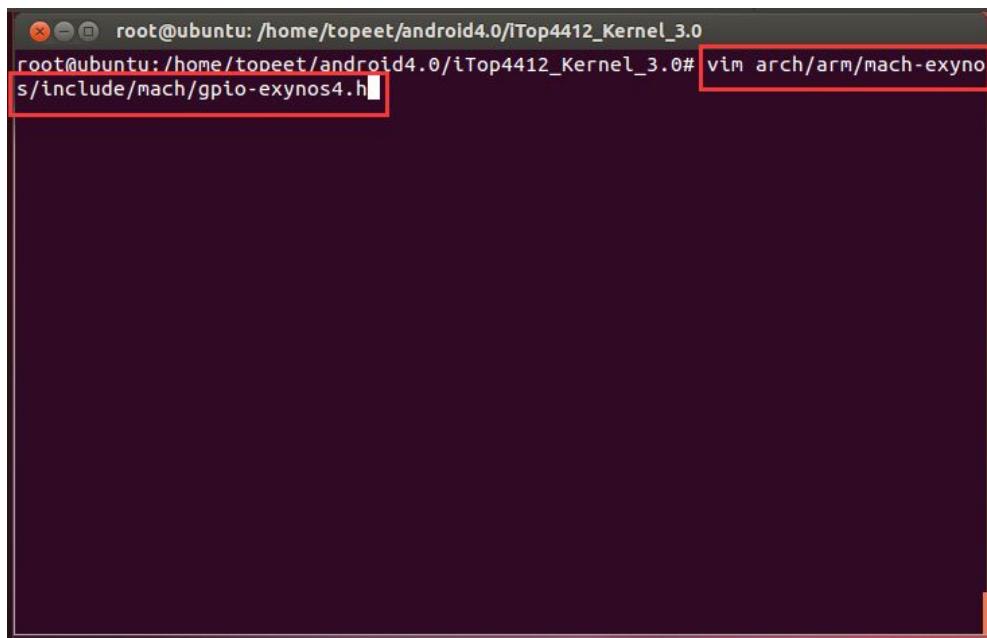
#### 14.4.2 GPIO 配置参数宏定义

GPIO 在 Linux 初始化 进行映射之后调用 GPIO 操作函数对 GPIO 宏定义进行操作就是对 GPIO 的操作。

这个 GPIO 宏定义文件都是由原厂提供，肯定是已经做好的，属于 BSP 板级开发包。

如下图所示，在源码目录中使用命令

“`vim arch/arm/mach-exynos/include/mach/gpio-exynos4.h`” 打开该文件。



如下图所示，可以看到所有的 GPIO 都已经定义了。

```
/* EXYNOS4 GPIO number definitions */
#define EXYNOS4_GPA0(_nr)          (EXYNOS4_GPIO_A0_START + (_nr))
#define EXYNOS4_GPA1(_nr)          (EXYNOS4_GPIO_A1_START + (_nr))
#define EXYNOS4_GPB(_nr)           (EXYNOS4_GPIO_B_START + (_nr))
#define EXYNOS4_GPC0(_nr)          (EXYNOS4_GPIO_C0_START + (_nr))
#define EXYNOS4_GPC1(_nr)          (EXYNOS4_GPIO_C1_START + (_nr))
#define EXYNOS4_GPD0(_nr)          (EXYNOS4_GPIO_D0_START + (_nr))
#define EXYNOS4_GPD1(_nr)          (EXYNOS4_GPIO_D1_START + (_nr))
#define EXYNOS4_GPF0(_nr)          (EXYNOS4_GPIO_F0_START + (_nr))
#define EXYNOS4_GPF1(_nr)          (EXYNOS4_GPIO_F1_START + (_nr))
#define EXYNOS4_GPF2(_nr)          (EXYNOS4_GPIO_F2_START + (_nr))
#define EXYNOS4_GPF3(_nr)          (EXYNOS4_GPIO_F3_START + (_nr))
#define EXYNOS4_GPK0(_nr)          (EXYNOS4_GPIO_K0_START + (_nr))
#define EXYNOS4_GPK1(_nr)          (EXYNOS4_GPIO_K1_START + (_nr))
#define EXYNOS4_GPK2(_nr)          (EXYNOS4_GPIO_K2_START + (_nr))
#define EXYNOS4_GPK3(_nr)          (EXYNOS4_GPIO_K3_START + (_nr))
#define EXYNOS4_GPL0(_nr)          (EXYNOS4_GPIO_L0_START + (_nr))
#define EXYNOS4_GPL1(_nr)          (EXYNOS4_GPIO_L1_START + (_nr))
#define EXYNOS4_GPL2(_nr)          (EXYNOS4_GPIO_L2_START + (_nr))
#define EXYNOS4_GPX0(_nr)          (EXYNOS4_GPIO_X0_START + (_nr))
#define EXYNOS4_GPX1(_nr)          (EXYNOS4_GPIO_X1_START + (_nr))
#define EXYNOS4_GPX2(_nr)          (EXYNOS4_GPIO_X2_START + (_nr))
```

The terminal window title is "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The content of the file "arch/arm/mach-exynos/s/include/mach/gpio-exynos4.h" is displayed, showing various GPIO definitions using the EXYNOS4\_GPIO macro followed by a start address and a number. The bottom right corner of the terminal window shows "140,0-1" and "78%".

在原理图中查找 KP\_COL0、VDD50\_EN 网络，最终连接到 4412 上的部分如下图所示。



如上图所示，则两个 Led 的宏定义为 EXYNOS4\_GPIO2(0) , EXYNOS4\_GPK1(1)。

#### 14.4.3 GPIO 配置函数和参数

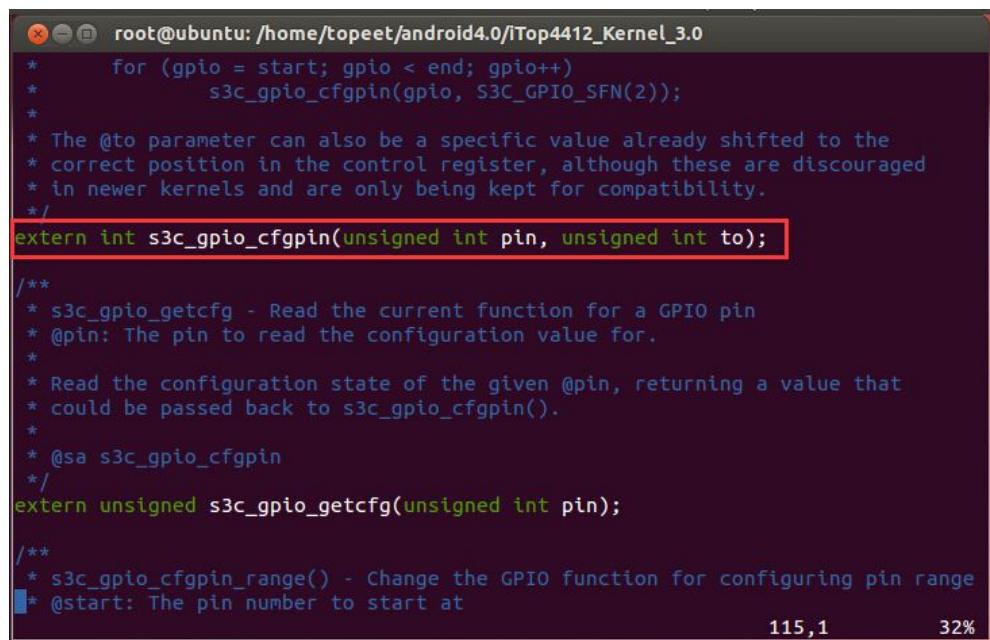
在 Linux 中，对 GPIO 的配置函数以及参数都已经集成到三星板级开发包中。

先来看一下配置函数，如下图所示，在源码目录中使用命令

“vim arch/arm/plat-samsung/include/plat/gpio-cfg.h” 打开该文件。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim arch/arm/plat-samsung/include/plat/gpio-cfg.h
```

如下图所示，s3c\_gpio\_cfgpin 函数就是本节实验需要的。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
*      for (gpio = start; gpio < end; gpio++)
*          s3c_gpio_cfgpin(gpio, S3C_GPIO_SFN(2));
*
* The @to parameter can also be a specific value already shifted to the
* correct position in the control register, although these are discouraged
* in newer kernels and are only being kept for compatibility.
*/
extern int s3c_gpio_cfgpin(unsigned int pin, unsigned int to);

/**
 * s3c_gpio_getcfg - Read the current function for a GPIO pin
 * @pin: The pin to read the configuration value for.
 *
 * Read the configuration state of the given @pin, returning a value that
 * could be passed back to s3c_gpio_cfgpin().
 *
 * @sa s3c_gpio_cfgpin
 */
extern unsigned s3c_gpio_getcfg(unsigned int pin);

/**
 * s3c_gpio_cfgpin_range() - Change the GPIO function for configuring pin range
 * @start: The pin number to start at
 */

```

115,1 32%

如上图所示，函数 `extern int s3c_gpio_cfgpin(unsigned int pin, unsigned int to);`

一般来说带有 `s3cxx` 的函数就是三星平台能够通用的函数。

`s3c_gpio_cfgpin` 管脚配置函数有两个参数

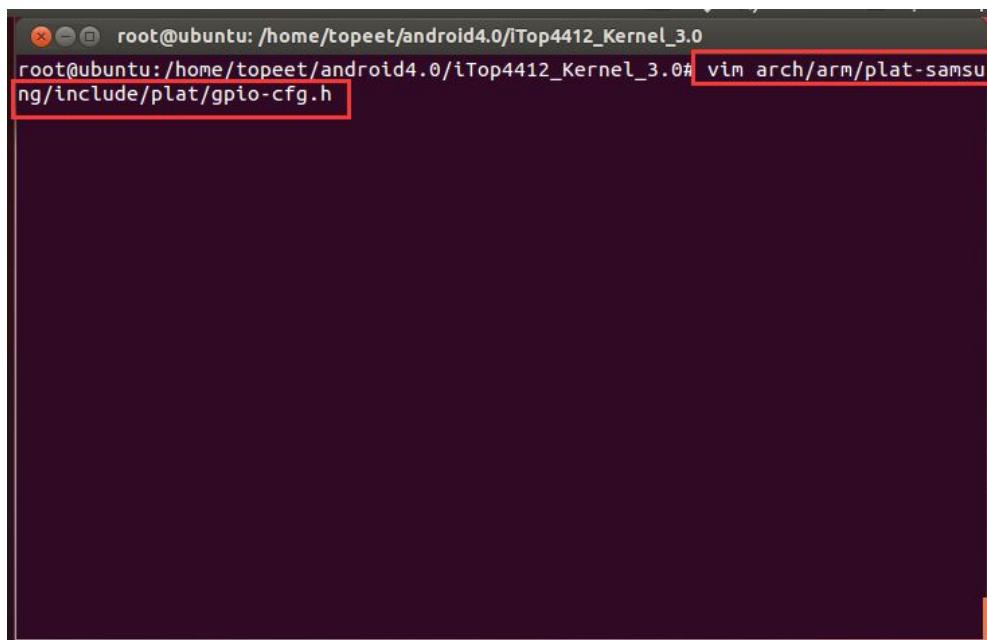
参数 `unsigned int pin` , 管脚

参数 `unsigned int to` , 配置参数。

再来看一下配置参数，如下图所示，在源码目录中使用命令

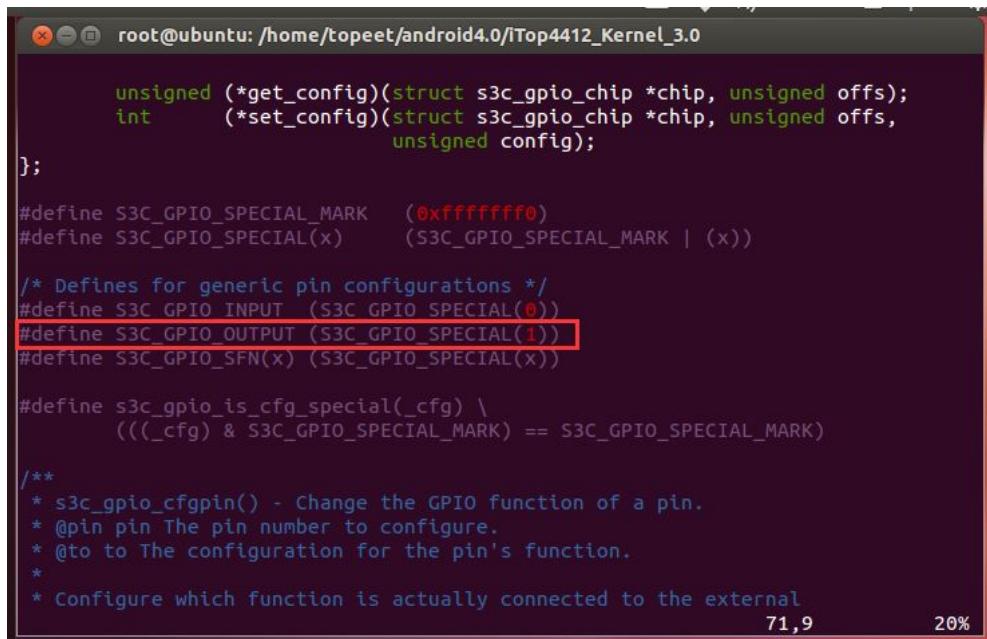
`"vim arch/arm/plat-samsung/include/plat/gpio-cfg.h"` 打开该文件，配置参数和函

数是在同一个函数中。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0# vim arch/arm/plat-samsung/include/plat/gpio-cfg.h
```

如下图所示，对于 GPIO 需要将其配置为输出模式，对应 S3C\_GPIO\_OUTPUT 宏定义。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0

unsigned (*get_config)(struct s3c_gpio_chip *chip, unsigned offs);
int      (*set_config)(struct s3c_gpio_chip *chip, unsigned offs,
                      unsigned config);
};

#define S3C_GPIO_SPECIAL_MARK  (0xffffffff0)
#define S3C_GPIO_SPECIAL(x)    (S3C_GPIO_SPECIAL_MARK | (x))

/* Defines for generic pin configurations */
#define S3C_GPIO_INPUT  (S3C_GPIO_SPECIAL(0))
#define S3C_GPIO_OUTPUT (S3C_GPIO_SPECIAL(1))
#define S3C_GPIO_SFN(x) (S3C_GPIO_SPECIAL(x))

#define s3c_gpio_is_cfg_special(_cfg) \
    (((_cfg) & S3C_GPIO_SPECIAL_MARK) == S3C_GPIO_SPECIAL_MARK)

/**
 * s3c_gpio_cfgpin() - Change the GPIO function of a pin.
 * @pin pin The pin number to configure.
 * @to to The configuration for the pin's function.
 *
 * Configure which function is actually connected to the external
71,9 20%
```

## 14.5 编写简单应用调用 LED 管脚，并测试

在前面的 devicenode\_linux\_module.c 文件上添加代码，首先将文件名 devicenode\_linux\_module.c 改为 leds.c。

先处理一下编译文件 Makefile，如下图所示，将 devicenode\_linux\_module 改为 leds。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件itop4412_hello.o
obj-m += leds.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

接着修改 leds.c 文件。

首先添加需要的头文件，如下图所示，分别是申请 GPIO、配置函数、配置参数、GPIO 宏定义等的头文件。然后将设备节点名称由 hello\_ctl123 修改为 hello\_ctl

```
#include <linux/init.h>
#include <linux/module.h>

/*驱动注册的头文件，包含驱动的结构体和注册和卸载的函数*/
#include <linux/platform_device.h>
/*注册杂项设备头文件*/
#include <linux/miscdevice.h>
/*注册设备节点的文件结构体*/
#include <linux/fs.h>

/*Linux中申请GPIO的头文件*/
#include <linux/gpio.h>
/*三星平台的GPIO配置函数头文件*/
/*三星平台EXYNOS系列平台，GPIO配置参数宏定义头文件*/
#include <plat/gpio-cfg.h>
#include <mach/gpio.h>
/*三星平台4412平台，GPIO宏定义头文件*/
#include <mach/gpio-exynos4.h>

#define DRIVER_NAME "hello_ctl"
#define DEVICE_NAME "hello_ctl"
```

然后需要修改的就是 probe 函数，一般说来 GPIO 的初始化都是在 probe 中。如下图所示，

调用配置函数以及配置函数。

```
static int hello_probe(struct platform_device *pdev) {
    int ret;

    printk(KERN_EMERG "\tinitialized\n");

    ret = gpio_request(EXYNOS4_GPL2(0), "LEDS");
    if(ret < 0){
        printk(KERN_EMERG "gpio_request EXYNOS4_GPL2(0) failed!\n");
        return ret;
    }

    s3c_gpio_cfgpin(EXYNOS4_GPL2(0), S3C_GPIO_OUTPUT);
    gpio_set_value(EXYNOS4_GPL2(0), 0);

    misc_register(&hello_dev);

    return 0;
}
```

然后就是修改一下 ioctl 函数，在 Linux 中对 GPIO 的控制一般是使用 ioctl，虽然 write 函数也可以实现类似的功能，但是 ioctl 函数的效率高一些。如下图所示，根据应用传入的参数给 GPIO 赋值。

```
static long hello_ioctl( struct file *files, unsigned int cmd, unsigned long arg){  
    printk("cmd is %d,arg is %d\n",cmd,arg);  
  
    if(cmd > 1){  
        printk(KERN_EMERG "cmd is 0 or 1\n");  
    }  
    if(arg > 1){  
        printk(KERN_EMERG "arg is only 1\n");  
    }  
  
    gpio_set_value(EXYNOS4_GPL2(0),cmd);  
  
    return 0;  
}
```

如上图所示，先对于参数做一个简单的判断，然后给 led 赋值。

接着再来看一下应用，如下图所示，应用比较简单，调用延时函数，首先将 Led 点亮三秒，然后再灭掉三秒，再点亮。

```
#include <stdio.h>  
  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <sys/ioctl.h>  
  
main(){  
    int fd;  
    char *hello_node = "/dev/hello_ctl";  
  
    /*O_RDWR只读打开,O_NDELAY非阻塞方式*/  
    if((fd = open(hello_node,O_RDWR|O_NDELAY))<0){  
        printf("APP open %s failed",hello_node);  
    }  
    else{  
        printf("APP open %s success",hello_node);  
        ioctl(fd,1,1);  
        sleep(3);  
        ioctl(fd,0,1);  
        sleep(3);  
        ioctl(fd,1,1);  
    }  
  
    close(fd);  
}
```

在 Ubuntu 系统下新建 leds 文件夹，将写好的 leds 和编译脚本拷贝到 leds 文件夹下，使用 Makefile 命令编译驱动，使用 “arm-none-linux-gnueabi-gcc -o invoke\_leds invoke\_leds.c -static” 命令编译应用。如下图所示。

The screenshot shows a terminal window with the following command history:

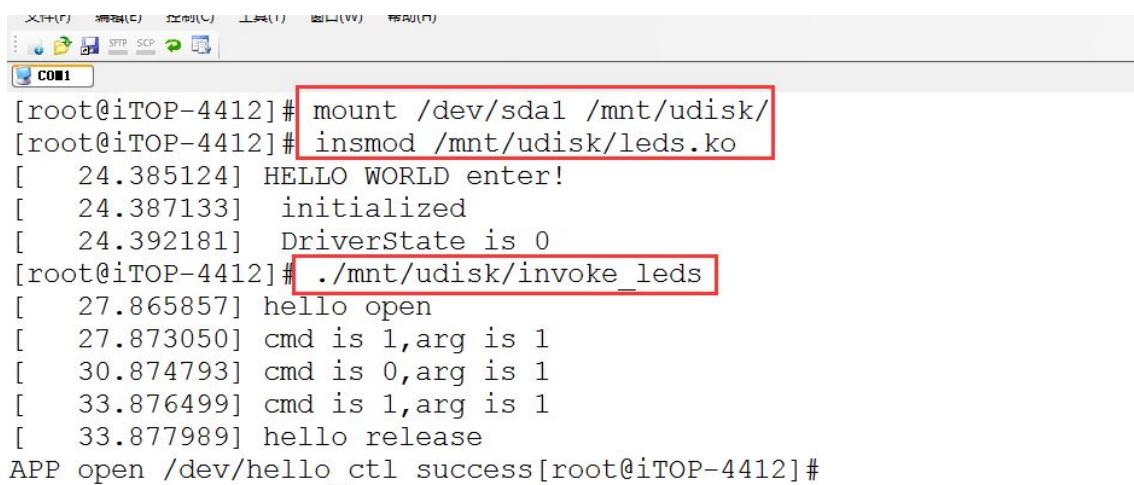
```
root@ubuntu:/home/topeet/leds
root@ubuntu:/home/topeet# mkdir leds
root@ubuntu:/home/topeet# cd leds/
root@ubuntu:/home/topeet/leds# ls
invoke_leds.c  leds.c  Makefile
root@ubuntu:/home/topeet/leds# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/leds modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/leds/leds.o
/home/topeet/leds/leds.c: In function 'hello_ioctl':
/home/topeet/leds/leds.c:28: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/topeet/leds/leds.mod.o
  LD [M] /home/topeet/leds/leds.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/leds# arm-none-linux-gnueabi-gcc -o invoke_leds invoke_
leds.c -static
root@ubuntu:/home/topeet/leds# ls
invoke_leds  led
invoke_leds.c  led.s  led.mod.c  led.o  modules.order
leds.ko  led.mod.o  Makefile  Module.symvers
root@ubuntu:/home/topeet/leds#
```

将上图中的文件 invoke\_leds 和 leds.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘符，

使用命令 “insmod /mnt/udisk/leds.ko” 加载驱动 leds.ko，

使用命令 “./mnt/udisk/invoke\_leds” 运行小应用 invoke\_leds，如下图所示。



The screenshot shows a terminal window titled 'COM1' with the following command history:

```
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/leds.ko
[ 24.385124] HELLO WORLD enter!
[ 24.387133] initialized
[ 24.392181] DriverState is 0
[root@iTOP-4412]# ./mnt/udisk/invoke_leds
[ 27.865857] hello open
[ 27.873050] cmd is 1,arg is 1
[ 30.874793] cmd is 0,arg is 1
[ 33.876499] cmd is 1,arg is 1
[ 33.877989] hello release
APP open /dev/hello_ctl success [root@iTOP-4412]#
```

经过上面的操作可观察到 led 小灯会一亮一灭一亮，中间大概间隔三秒钟。

# 实验 15 LEDS 驱动二

## 15.1 本章导读

考虑到用户在实际开发中可能需要更多的 GPIO，本实验给用户提供 32 个 GPIO，需要注意的是有一部分是复用的 GPIO，需要首先对内核进行配置和编译之后才能使用。例如 WIFI 模块和摄像头接口用到的 GPIO，如果当做 GPIO 使用，那么就无法使用 WIFI 模块和摄像头模块。

### 15.1.1 工具

#### 15.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 15.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “gpis”

## 15.1.2 预备课程

实验 14 LEDS 驱动一

## 15.1.3 视频资源

本节配套视频为“视频 15\_LED 驱动二”

## 15.2 学习目标

本章需要学习以下内容：

如何将功能复用的 IO 配置为 GPIO

## 15.3 操作过程

因为前面关于 GPIO 的使用都已经介绍的差不多了，现在直接给大家介绍操作过程。

如下图所示，提供了 32 个 GPIO。

```
/*led的两个IO，网络是KP_COLO, VDD50_EN*/
/*蜂鸣器的1个IO，网络是MOTOR_PWM*/
/*矩阵键盘的8个IO，网络是CHG_FLT, HOOK_DET, CHG_UOK, XEINT14_BAK,
GM_INT1, 6260_GPIO1, CHG_COK, XEINT29/KP_ROW13/ALV_DBG25*/
/*摄像头的14个IO，网络是CAM_MCLK, CAM2M_RST, CAM2M_PWDN,
CAM_D5, CAM_D7, CAM_D6, CAM_D4, CAM_D3, CAM_D2, CAM_D1,
CAM_PCLK, CAM_D0, CAM_VSYNC, CAM_HREF。
I2C_SDA7, I2C_SCL7也是可以设置为GPIO，不过总线一般不要去动它*/
/*WIFI模块的7个IO, WIFI_D3, WIFI_CMD, WIFI_D1, WIFI_CLK, WIFI_D0, WIFI_D2, GPC1_1*/
/*串口RX和TX等也是可以设置为GPIO，一般不要动它*/
/*数组中有32个引出到端子或者模块的IO，还有类似sd卡等也是可以作为GPIO,
其它引到连接器但是没有使用的GPIO等等*/
/*SCP管脚编号和POP的稍微有点不同，下面是SCP的*/
```

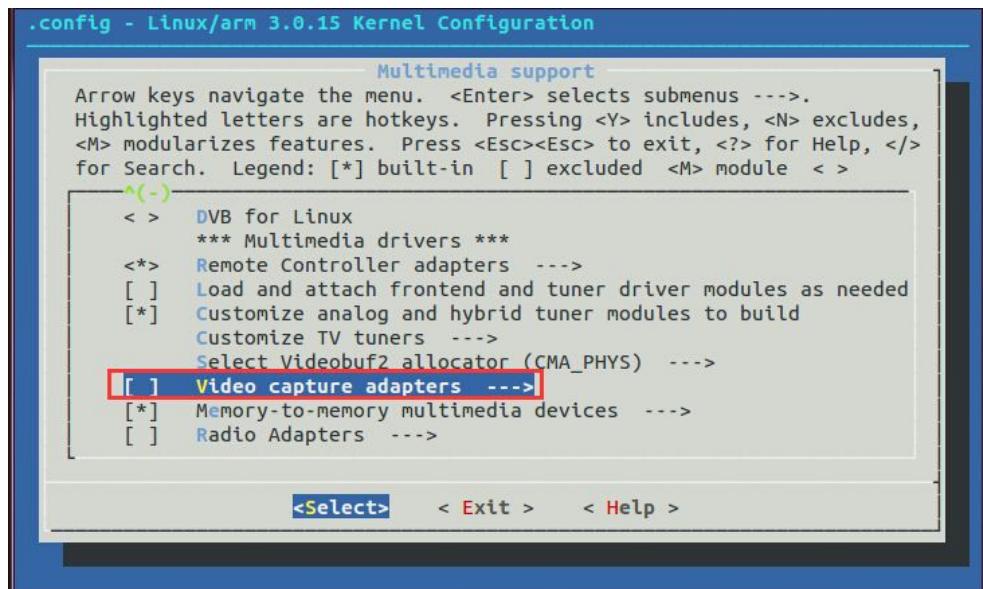
先需要去掉调用了这些 GPIO 的相关驱动。

## 1 ) 去掉摄像头驱动 VIDEO\_OV5640

Device Drivers --->

Multimedia support(MEDIA\_SUPPORT [=y]) --->

Video capture adapters(VIDEO\_CAPTURE\_DRIVERS [=y]) ( 去掉 ) --->



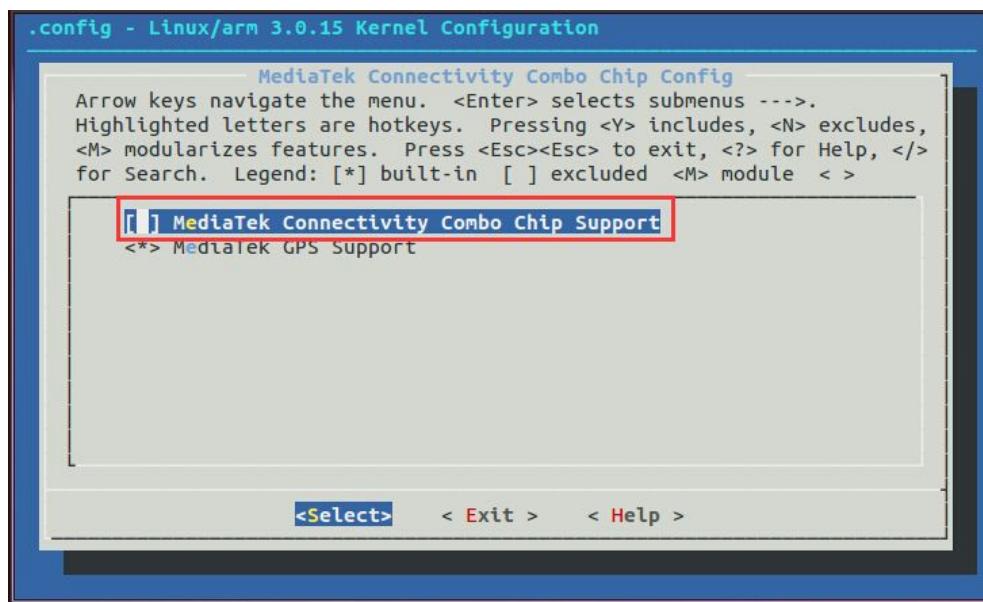
## 2 ) 去掉 WIFI 驱动 MTK\_COMBO\_CHIP\_MT662

Device Drivers --->

MediaTek Connectivity Combo Chip Config --->

MediaTek Connectivity Combo Chip Support (MTK\_COMBO [=y]) ( 去掉 ) --->

Select Chip (<choice> [=y]) --->

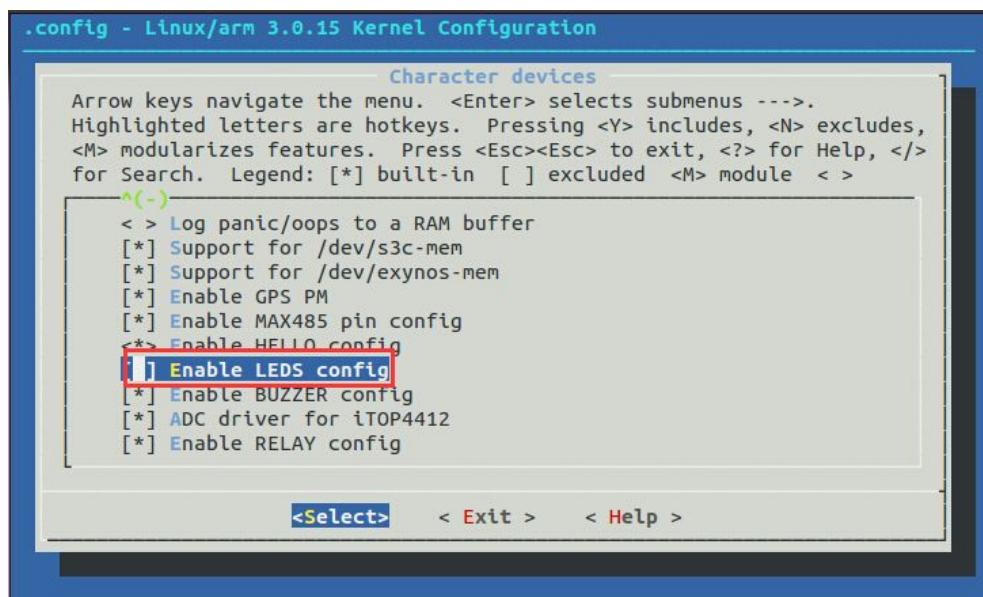


### 3 ) 去掉 leds 的驱动

Device Drivers --->

Character devices --->

Enable LEDS config --->

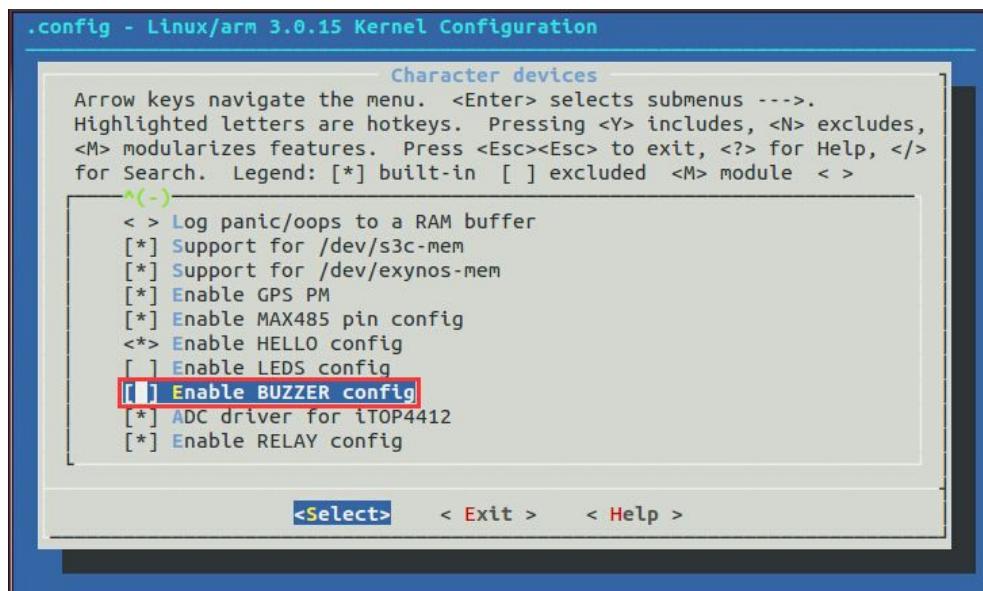


### 4 ) 去掉 Buzzer 的驱动

Device Drivers --->

Character devices --->

Enable BUZZER config --->



修改完之后重新编译内核，将新生成的内核二进制文件 zImage 烧写到开发板。

接着将前一个实验的 leds.c 改为 gpios.c。

修改一下 Makefile 文件，如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件itop4412_hello.o
obj-m += gpios.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)}Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)}当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules
    |
#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.o
```

然后将这些 GPIO 打包为一个数组，数组如下图所示，然后定义一下数组长度 LED\_NUM。

```
static int led_gpios[] = {
    EXYNOS4_GPL2(0), EXYNOS4_GPK1(1),
    EXYNOS4_GPD0(0),

    EXYNOS4_GPX1(0), EXYNOS4_GPX1(3), EXYNOS4_GPX1(5), EXYNOS4_GPX1(6),
    EXYNOS4_GPX3(0), EXYNOS4_GPX2(6), EXYNOS4_GPX2(7), EXYNOS4_GPX3(5),

    EXYNOS4212_GPJ1(3), EXYNOS4_GPL0(1), EXYNOS4_GPL0(3), EXYNOS4212_GPJ1(0),
    EXYNOS4212_GPJ1(2), EXYNOS4212_GPJ1(1), EXYNOS4212_GPJO(7), EXYNOS4212_GPJO(6),
    EXYNOS4212_GPJO(5), EXYNOS4212_GPJO(4), EXYNOS4212_GPJO(0), EXYNOS4212_GPJO(3),
    EXYNOS4212_GPJO(1), EXYNOS4212_GPJO(2),

    EXYNOS4_GPK3(6), EXYNOS4_GPK3(1), EXYNOS4_GPK3(4), EXYNOS4_GPK3(0),
    EXYNOS4_GPK3(3), EXYNOS4_GPK3(5), EXYNOS4_GPC1(1),
};

#define LED_NUM      ARRAY_SIZE(led_gpios)
```

将设备节点的名称修改为 hello\_gpio，如下图所示。

```
#define DRIVER_NAME "hello_ctl"
#define DEVICE_NAME "hello_gpio"
```

```
MODULE_LICENSE ("Dual BSD/GPL");
MODULE_AUTHOR ("TOPEET");
```

如下图所示，先在 probe 函数中初始化。

```
static int hello_probe(struct platform_device *pdev) {
    int ret,i;

    printk(KERN_EMERG "\tinitialized\n");

    for(i=0; i<LED_NUM; i++)
    {
        ret = gpio_request(led_gpios[i], "LED");
        if (ret) {
            printk("%s: request GPIO %d for LED failed, ret = %d\n",
                   DRIVER_NAME, i, ret);
        }
        else{
            s3c_gpio_cfgpin(led_gpios[i], S3C_GPIO_OUTPUT);
            gpio_set_value(led_gpios[i], 1);
        }
    }

    gpio_set_value(led_gpios[2], 0);

    misc_register(&hello_dev);
    if(ret<0)
    {
        printk("leds:register device failed!\n");
        goto exit;
    }
    return 0;
}

exit:
    misc_deregister(&hello_dev);
    return ret;
    return 0;
}
```

如下图所示，然后是 ioctl 函数中写一个简单的 switch 语句。

```
static long hello_ioctl( struct file *files, unsigned int cmd, unsigned long arg) {
    printk("cmd is %d,arg is %d\n",cmd,arg);

    switch(cmd)
    {
        case 0:
        case 1:
            if (arg > LED_NUM) {
                return -EINVAL;
            }

            gpio_set_value(led_gpios[arg], cmd);
            break;

        default:
            return -EINVAL;
    }

    gpio_set_value(led_gpios[2], 0);

    return 0;
}
```

如下图所示，最后是在 remove 函数中添加 gpio\_free 释放 GPIO。

```
static int hello_remove(struct platform_device *pdev) {
    int i;

    printk(KERN_EMERG "\tremove\n");

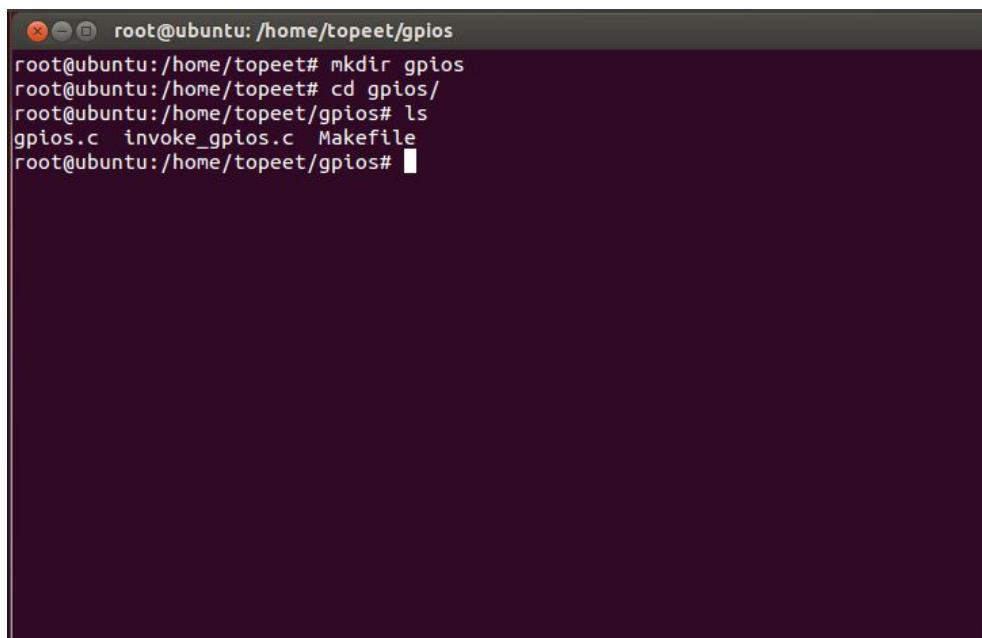
    for(i=0; i<LED_NUM; i++)
    {
        gpio_free(led_gpios[i]);
    }

    misc_deregister(&hello_dev);
    return 0;
}
```

接着简单的修改一下应用。

```
int main(int argc , char **argv){  
    int fd,i,cmd=2;  
    char *hello_node = "/dev/hello_gpio";  
    char *cmd0 = "0";  
    char *cmd1 = "1";  
  
    printf("argv[0] is %s;argv[1] is %s;",argv[0],argv[1]);  
  
    if(strcmp(argv[1], cmd0) == 0){  
        cmd = 0;  
        printf("cmd is 0!\n");  
    }  
    if(strcmp(argv[1], cmd1) == 0){  
        cmd = 1;  
        printf("cmd is 1!\n");  
    }  
  
/*O_RDWR只读打开,O_NDELAY非阻塞方式*/  
    if((fd = open(hello_node,O_RDWR|O_NDELAY))<0){  
        printf("APP open %s failed!\n",hello_node);  
    }  
    else{  
        printf("APP open %s success!\n",hello_node);  
        for(i=0;i<GPIO5;i++){  
            ioctl(fd,cmd,i);  
            printf("APP ioctl %s ,cmd is %d,i is %d!\n",hello_node,cmd,i);  
        }  
    }  
    close(fd);  
}
```

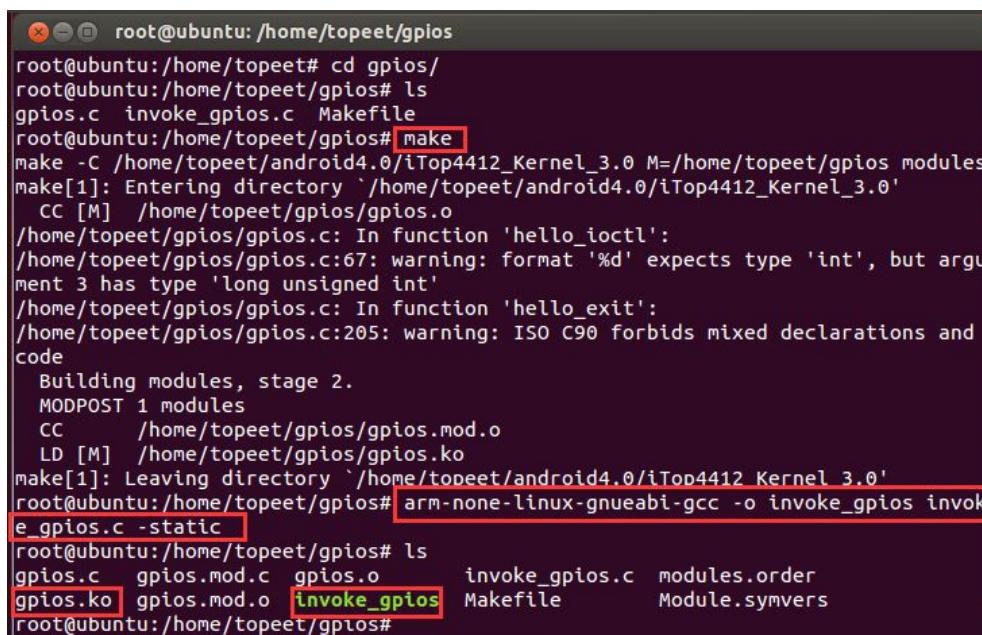
在 Ubuntu 系统下新建 gpios 文件夹，将写好的 gpios、编译脚本以及应用拷贝到 gpios 文件夹下，如下图所示。



```
root@ubuntu:/home/topeet/gpios
root@ubuntu:/home/topeet# mkdir gpios
root@ubuntu:/home/topeet# cd gpios/
root@ubuntu:/home/topeet/gpios# ls
gpios.c invoke_gpios.c Makefile
root@ubuntu:/home/topeet/gpios#
```

使用 Makefile 命令编译驱动，然后使用

“arm-none-linux-gnueabi-gcc -o invoke\_gpios invoke\_gpios.c -static” 命令编译应用，如下图所示。



```
root@ubuntu:/home/topeet/gpios
root@ubuntu:/home/topeet# cd gpios/
root@ubuntu:/home/topeet/gpios# ls
gpios.c invoke_gpios.c Makefile
root@ubuntu:/home/topeet/gpios# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/gpios modules
make[1]: Entering directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/gpios/gpios.o
/home/topeet/gpios/gpios.c: In function 'hello_ioctl':
/home/topeet/gpios/gpios.c:67: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
/home/topeet/gpios/gpios.c: In function 'hello_exit':
/home/topeet/gpios/gpios.c:205: warning: ISO C90 forbids mixed declarations and code
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/topeet/gpios/gpios.mod.o
  LD [M] /home/topeet/gpios/gpios.ko
make[1]: Leaving directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/gpios# arm-none-linux-gnueabi-gcc -o invoke_gpios invoke_gpios.c -static
root@ubuntu:/home/topeet/gpios# ls
gpios.c gpios.mod.c gpios.o invoke_gpios.c modules.order
gpios.ko gpios.mod.o invoke_gpios Makefile Module.symvers
root@ubuntu:/home/topeet/gpios#
```

将上图中的文件 invoke\_gpios 和 gpios.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘符。

使用命令 “insmod /mnt/udisk/gpios.ko” 加载驱动 gpios.ko，

使用命令 “./mnt/udisk/invoke\_gpios 0” 或者 “./mnt/udisk/invoke\_gpios 1” 运行小

应用 invoke\_gpios，如下图所示。

```
[root@iTOP-4412]# insmod /mnt/udisk/gpios.ko
[ 313.913783] HELLO WORLD enter!
[ 313.915457] initialized
[ 313.920964] DriverState is 0
[root@iTOP-4412]# lsmod
gpios 2587 0 - Live 0xbff000000
[root@iTOP-4412]# ./mnt/udisk/invite_gpios 0
argv[0] is ./mnt/[ 329.128562] hello open
[ 329.130901] cmd is 0,arg is 0
[ 329.133844] cmd is 0,arg is 1
[ 329.136811] cmd is 0,arg is 2
[ 329.139739] cmd is 0,arg is 3
[ 329.142724] cmd is 0,arg is 4
[ 329.145660] cmd is 0,arg is 5
[ 329.148592] cmd is 0,arg is 6
[ 329.151553] cmd is 0,arg is 7
[ 329.154492] cmd is 0,arg is 8
[ 329.157455] cmd is 0,arg is 9
[ 329.160431] cmd is 0,arg is 10
[ 329.163432] cmd is 0,arg is 11
[ 329.166482] cmd is 0,arg is 12
[ 329.169508] cmd is 0,arg is 13
[ 329.172558] cmd is 0,arg is 14
[ 329.175595] cmd is 0,arg is 15
[ 329.178621] cmd is 0,arg is 16
[ 329.181671] cmd is 0,arg is 17
[ 329.184697] cmd is 0,arg is 18
[ 329.187747] cmd is 0,arg is 19
[ 329.190786] cmd is 0,arg is 20
[ 329.193854] cmd is 0,arg is 21
```

如上图所示，使用命令 “./mnt/udisk/invoke\_gplos 0” 之后灯会灭，然后其它的 GPIO 也会都成为低电平。

使用命令 “./mnt/udisk/invoke\_gplos 0” 之后灯会亮，然后其它的 GPIO 也会都成为高电平。

也可以检查一下运行应用之后有没有错误，如果有错误，多半是因为没有将调用对应 GPIO 的驱动去除，导致 GPIO 被占用了。

# 实验 16 驱动模块传参数

## 16.1 本章导读

加载模块的时候还可以通过 insmod 命令传参数，掌握了这个知识点之后调试起来会方便很多。

### 16.1.1 工具

#### 16.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 16.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “module\_param”

### 16.1.2 预备课程

无

### 16.1.3 视频资源

本节配套视频为“视频 16\_驱动模块传参数”

### 16.2 学习目标

本章需要学习以下内容：

加载模块的时候，向驱动模块传参数

### 16.3 实验操作

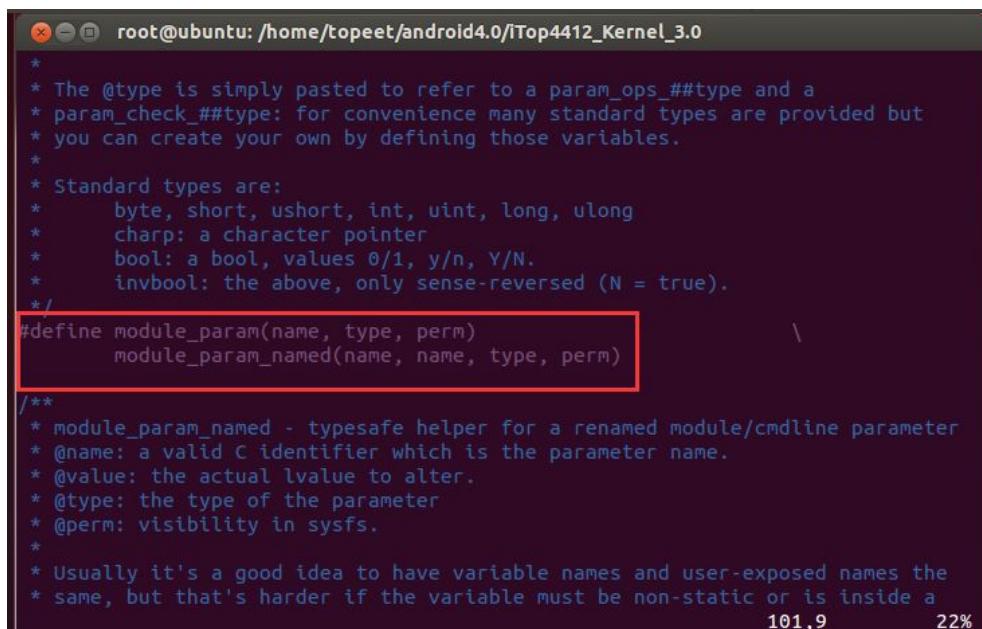
在应用程序中，可以通过 main 函数向其中传参数，这个功能大家用起来已经很熟练了。

实际上在加载模块的时候也是可以向其中传参数的。

在头文件“include/linux/moduleparam.h”中包含了向模块传参数的函数。这个功能是集成的，在任何 linux 系统之中都可以使用。

参数传递有两个函数，分别是函数 module\_param 和函数 module\_param\_array。

函数 module\_param 支持单个参数传递，在头文件中，如下图所示。



```
* The @type is simply pasted to refer to a param_ops_##type and a
* param_check_##type: for convenience many standard types are provided but
* you can create your own by defining those variables.
*
* Standard types are:
*     byte, short, ushort, int, uint, long, ulong
*     charp: a character pointer
*     bool: a bool, values 0/1, y/n, Y/N.
*     invbool: the above, only sense-reversed (N = true).
*/
#define module_param(name, type, perm) \
    module_param_named(name, name, type, perm)

/***
 * module_param_named - typesafe helper for a renamed module/cmdline parameter
 * @name: a valid C identifier which is the parameter name.
 * @value: the actual lvalue to alter.
 * @type: the type of the parameter
 * @perm: visibility in sysfs.
 *
 * Usually it's a good idea to have variable names and user-exposed names the
 * same, but that's harder if the variable must be non-static or is inside a
```

101,9 22%

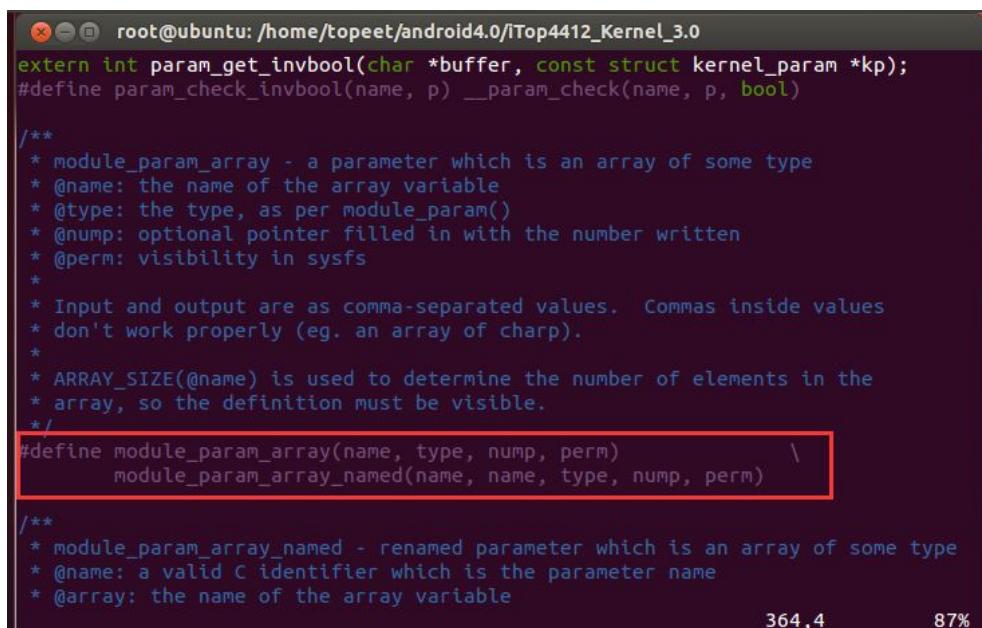
如上图所示，这个宏定义函数有三个参数分别如下。

参数 name，模块参数的名称；

参数 type，模块参数的数据类型（支持 int long short uint ulong ushort 类型）；

参数 perm，模块参数的访问权限（S\_IRUSR 参数表示所有文件所有者可读）。

函数 module\_param\_array 支持多个参数传递，在头文件中，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
extern int param_get_invbool(char *buffer, const struct kernel_param *kp);
#define param_check_invbool(name, p) __param_check(name, p, bool)

/**
 * module_param_array - a parameter which is an array of some type
 * @name: the name of the array variable
 * @type: the type, as per module_param()
 * @num: optional pointer filled in with the number written
 * @perm: visibility in sysfs
 *
 * Input and output are as comma-separated values. Commas inside values
 * don't work properly (eg. an array of charp).
 *
 * ARRAY_SIZE(@name) is used to determine the number of elements in the
 * array, so the definition must be visible.
 */
#define module_param_array(name, type, num, perm) \
    module_param_array_named(name, name, type, num, perm)

/**
 * module_param_array_named - renamed parameter which is an array of some type
 * @name: a valid C identifier which is the parameter name
 * @array: the name of the array variable

```

364,4 87%

如上图所示，如上图所示，这个宏定义函数有四个参数分别如下。

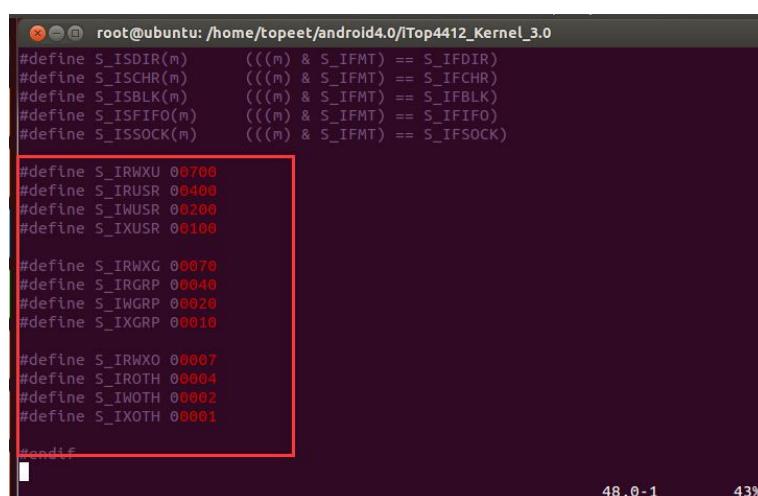
参数 name，模块参数的名称；

参数 type，模块参数的数据类型（支持 int long short uint ulong ushort 类型）；

参数 num，保存参数的数量；

参数 perm，模块参数的访问权限（S\_IRUSR 参数表示所有文件所有者可读）。

这里再介绍一下参数 perm ,参数 perm 表示此参数在 sysfs 文件系统中所对应的文件节点的属性，其权限在 “include/linux/stat.h” 中有定义，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
#define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR)
#define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR)
#define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK)
#define S_ISIFO(m) (((m) & S_IFMT) == S_IFIFO)
#define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)

#define S_IRWXU 00700
#define S_IRUSR 00400
#define S_IWUSR 00200
#define S_IXUSR 00100

#define S_IRWXG 00070
#define S_IRGRP 00040
#define S_IWGRP 00020
#define S_IXGRP 00010

#define S_IRWXO 00007
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001

#endif
```

48,0-1 43%

部分常用参数权限解释如下。

- #defineS\_IRUSR 00400 文件所有者可读
- #defineS\_IWUSR 00200 文件所有者可写
- #defineS\_IXUSR 00100 文件所有者可执行
- #defineS\_IRGRP 00040 与文件所有者同组的用户可读
- #defineS\_IWGRP 00020
- #defineS\_IXGRP 00010
- #defineS\_IROTH 00004 与文件所有者不同组的用户可读
- #defineS\_IWOTH 00002
- #defineS\_IXOTH 00001

其它的可以使用下面的方法来判断：

可以将数字最后三位转化为二进制:xxx xxx xxx,高位往低位依次看,第一位为 1 表示文件所有者可读,第二位为 1 表示文件所有者可写,第三位为 1 表示文件所有者可执行;接下来三位表示文件所有者同组成员的权限;再下来三位为不同组用户权限。

接着在 02\_DriverModule\_01 例程中基础上做本实验。

先修改一下 Makefile。

将原来的 “rm -rf \*.o” 改为 “rm -rf \*.mod.c \*.o \*.order \*.ko \*.mod.o \*.symvers” 。

将 “mini\_linux\_module” 改为 “module\_param” 。

如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += module_param.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${KDIR} Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${PWD} 当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules
    |
#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

接着将 02\_DriverModule\_01 的例程改为 module\_param.c。

如下图所示，添加传参数功能的头文件 “linux/moduleparam.h” 和 “linux/stat.h” 。

```
#include <linux/init.h>
/*包含初始化宏定义的头文件,代码中的module_init和module_exit在此文件中*/
#include <linux/module.h>
/*包含初始化加载模块的头文件,代码中的MODULE_LICENSE在此头文件中*/

/*定义module_param module_param_array的头文件*/
#include <linux/moduleparam.h>
/*定义module_param module_param_array中perm的头文件*/
#include <linux/stat.h>
```

如下图所示，然后调用函数 module\_param 和 module\_param\_array 接收参数。

```
MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/

static int module_arg1,module_arg2;
static int int_array[50];
static int int_num;

module_param(module_arg1,int,S_IRUSR);
module_param(module_arg2,int,S_IRUSR);
module_param_array(int_array,int,&int_num,S_IRUSR);
```

接着添加代码在初始化的时候将数据打印出去，如下图所示。

```
static int hello_init(void)
{
    int i;

    printk(KERN_EMERG "module_arg1 is %d!\n",module_arg1);
    printk(KERN_EMERG "module_arg2 is %d!\n",module_arg2);

    for(i=0;i<int_num;i++){
        printk(KERN_EMERG "int_array[%d] is %d!\n",i,int_array[i]);
    }

    printk(KERN_EMERG "Hello World enter!\n");
    /*打印信息， KERN_EMERG表示紧急信息*/
    return 0;
}
```

在 Ubuntu 系统下新建 module\_param 文件夹，将写好的 module\_param.c、编译脚本以及应用拷贝到 module\_param 文件夹下，如下图所示。

```
root@ubuntu:/home/topeet/module_param
root@ubuntu:/home/topeet# mkdir module_param
root@ubuntu:/home/topeet# cd module_param/
root@ubuntu:/home/topeet/module_param# ls
Makefile module_param.c
root@ubuntu:/home/topeet/module_param#
```

使用 Makefile 命令编译驱动命令 “Make” 编译应用，如下图所示。

```
root@ubuntu:/home/topeet/module_param
root@ubuntu:/home/topeet# mkdir module_param
root@ubuntu:/home/topeet# cd module_param/
root@ubuntu:/home/topeet/module_param# ls
Makefile module_param.c
root@ubuntu:/home/topeet/module_param# make
make -C /home/topeet/android4.0/iTop4412_kernel_3.0 M=/home/topeet/module_param
modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
CC [M] /home/topeet/module_param/module_param.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/topeet/module_param/module_param.mod.o
LD [M] /home/topeet/module_param/module_param.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/module_param# ls
Makefile module_param.ko module_param.mod.o modules.order
module_param.c module_param.mod.c module_param.o Module.symvers
root@ubuntu:/home/topeet/module_param#
```

将上图中的文件 module\_param.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘。

使用命令 “insmod /mnt/udisk/module\_param.ko module\_arg1=10 module\_arg2=20 int\_array=11,12,13,14,15,16,17,18” 加载驱动 module\_param.ko , 如下图所示。

```
[root@iTOP-4412]# insmod /mnt/udisk/module_param.ko module_arg1=10 module_arg2=20 int_array=11,12,13,14,15,16,17,18
[ 63.510332] module_arg1 is 10!
[ 63.511906] module_arg2 is 20!
[ 63.514944] int_array[0] is 11!
[ 63.519408] int_array[1] is 12!
[ 63.521238] int_array[2] is 13!
[ 63.524318] int_array[3] is 14!
[ 63.527482] int_array[4] is 15!
[ 63.530604] int_array[5] is 16!
[ 63.533693] int_array[6] is 17!
[ 63.536830] int_array[7] is 18!
[ 63.539941] Hello World enter!
[root@iTOP-4412]#
```

另外还可以使用 cat 命令在系统的 module 下查参数。

使用命令 “cat /sys/module/module\_param/parameters/xxx” , 如下图所示。

```
[root@iTOP-4412]# cat /sys/module/module_param/parameters
cat: read error: Is a directory
[root@iTOP-4412]# cat /sys/module/module_param/parameters/
int_array module_arg1 module_arg2
[root@iTOP-4412]# cat /sys/module/module_param/parameters/int_array
11,12,13,14,15,16,17,18
[root@iTOP-4412]# cat /sys/module/module_param/parameters/module_arg1
10
[root@iTOP-4412]# cat /sys/module/module_param/parameters/module_arg2
20
[root@iTOP-4412]#
```

另外在 Ubuntu 的 module\_param 文件夹下可以使用以下清除命令 “make clean” , 可以看到除了 Makefile 文件和\*.c 文件其它都被清除了。如下图所示。

```
root@ubuntu:/home/topeet/module_param
root@ubuntu:/home/topeet# mkdir module_param
root@ubuntu:/home/topeet# cd module_param/
root@ubuntu:/home/topeet/module_param# ls
Makefile module_param.c
root@ubuntu:/home/topeet/module_param# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/module_param
modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/module_param/module_param.o
  Building modules, stage 2.
  MODPOST 1 modules
    CC      /home/topeet/module_param/module_param.mod.o
    LD [M] /home/topeet/module_param/module_param.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/module_param# ls
Makefile           module_param.ko     module_param.mod.o  modules.order
module_param.c     module_param.mod.c  module_param.o    Module.symvers
root@ubuntu:/home/topeet/module_param# make clean
rm -rf *.mod.c *.mod.o *.order *.ko *.mod.o *.symvers
root@ubuntu:/home/topeet/module_param# ls
Makefile module_param.c
root@ubuntu:/home/topeet/module_param#
```

# 实验 17 静态申请字符类设备号

## 17.1 本章导读

这里开始介绍的是纯粹的字符设备，前面学习的是杂项设备，主设备号已经固定为 10，。

但是考虑到大家学习之后，也会自己申请主设备号以及次设备号，就是标准的字符设备，所以从这一个实验开始给大家介绍相关的知识。

字符设备分为静态申请和动态申请，静态申请就是主设备号是程序员手动分配了，动态申请是系统给分配，本实验先介绍静态申请的方法。

### 17.1.1 工具

#### 17.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 17.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）

4 ) 源码文件夹 “request\_cdev\_num”

### 17.1.2 预备课程

实验 16 驱动模块传参数

### 17.1.3 视频资源

本节配套视频为 “视频 17\_静态申请字符类设备号”

## 17.2 学习目标

本章需要学习以下内容：

静态申请字符类设备号

进一步理解主设备号和次设备号

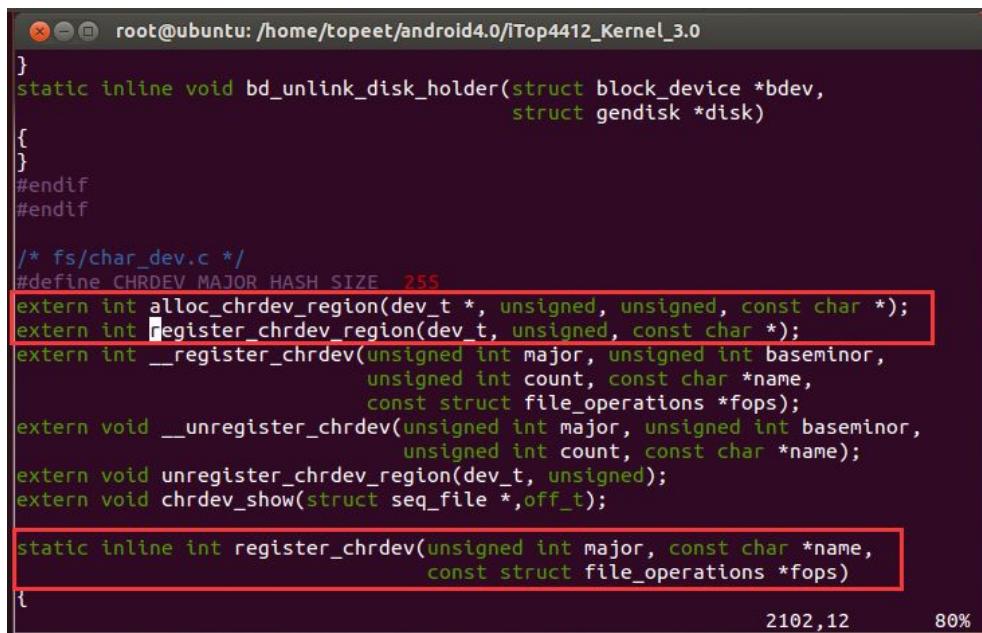
### 17.3 字符设备基本知识

前面已经提到过很多次，Linux 的设备主要分为三大类，字符设备、块设备、网络设备。

前面带大家写的驱动是杂项设备，它和字符设备唯一的区别就是主设备号已经搞定了，不需要像字符设备那样去手动申请。

这里先给大家介绍几个常用的申请字符类设备的函数。

如下图所示，在头问价 “include/linux/fs.h” 中，可以找到三个注册字符设备的函数。这三个分别是函数 register\_chrdev\_region ,函数 alloc\_chrdev\_region ,函数 register\_chrdev()。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
}
static inline void bd_unlink_disk_holder(struct block_device *bdev,
                                         struct gendisk *disk)
{
}

#endif
#endif

/* fs/char_dev.c */
#define_CHRDEV_MAJOR_HASH_SIZE 255
extern int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *);
extern int register_chrdev_region(dev_t, unsigned, const char *);
extern int __register_chrdev(unsigned int major, unsigned int baseminor,
                            unsigned int count, const char *name,
                            const struct file_operations *fops);
extern void __unregister_chrdev(unsigned int major, unsigned int baseminor,
                               unsigned int count, const char *name);
extern void unregister_chrdev_region(dev_t, unsigned);
extern void chrdev_show(struct seq_file *, off_t);

static inline int register_chrdev(unsigned int major, const char *name,
                                 const struct file_operations *fops)
{

```

如上图所示，三个函数的区别如下。

函数 `register_chrdev_region()` 是提前知道设备的主次设备号,再去申请设备号。

函数 `alloc_chrdev_region()` 是动态分配主次设备号。

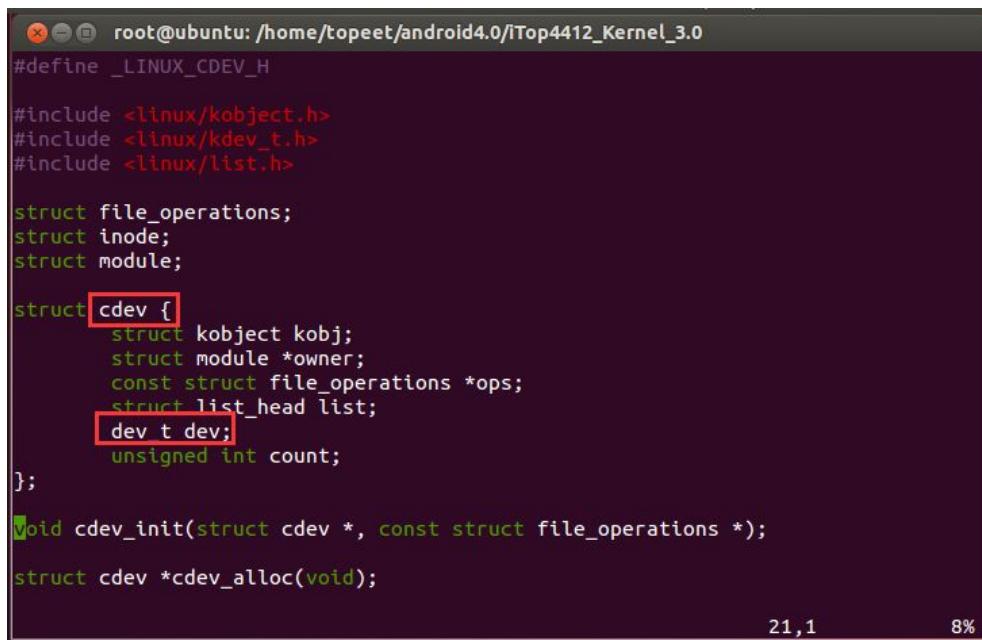
函数 `register_chrdev()`。是老版本的设备号注册方式 ,只分配主设备号。从设备号在 `mknod` 的时候指定。这个函数现在虽然仍然可以支持，但是已经不再使用了。

本节实验主要介绍的是动态申请函数 `register_chrdev_region()`。

在函数 `extern int register_chrdev_region(dev_t, unsigned, const char *)` 中，它的参数 `dev_t` 相关的知识会慢慢的介绍到，提到的部分大家需要理解并应用，而且 Linux 系统中还有几个函数和参数是专门为它服务的。

在头文件 “`include/linux/cdev.h`” 中，如下图所示。

在 `cdev` 中有专门一个参数 `dev_t dev`。



The screenshot shows a terminal window titled "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The code displayed is a portion of a header file, likely `<linux/cdev.h>`. It defines the `cdev` structure and its initialization function. The `dev_t dev;` field is highlighted with a red box.

```
#define _LINUX_CDEV_H

#include <linux/kobject.h>
#include <linux/kdev_t.h>
#include <linux/list.h>

struct file_operations;
struct inode;
struct module;

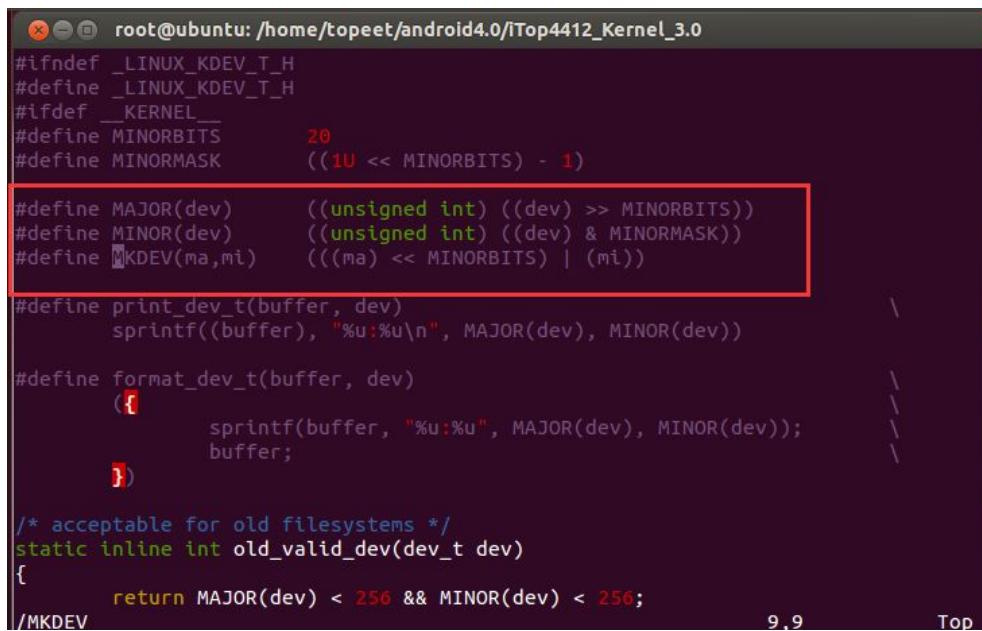
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};

void cdev_init(struct cdev *, const struct file_operations *);
struct cdev *cdev_alloc(void);
```

如上图所示，`cdev` 类型是字符设备描述的结构，其中的设备号必须用 “`dev_t`” 类型来描述，高 12 位为主设备号，低 20 位为次设备号。

把 `dev` 理解为二进制数，就很容易理解了，`dev_t` 是一个 32 位类型的数，前 12 位表示主设备，后 20 位表示次设备号。

如下图所示，在头文件 “`include/linux/kdev_t.h`” 中，有一些专门用来处理 `dev_t` 数据类型的宏定义。



```
#ifndef _LINUX_KDEV_T_H
#define _LINUX_KDEV_T_H
#ifndef __KERNEL__
#define MINORBITS      20
#define MINORMASK      ((1U << MINORBITS) - 1)

#define MAJOR(dev)      (((unsigned int) ((dev) >> MINORBITS)))
#define MINOR(dev)      (((unsigned int) ((dev) & MINORMASK)))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))

#define print_dev_t(buffer, dev)
    sprintf((buffer), "%u:%u\n", MAJOR(dev), MINOR(dev))

#define format_dev_t(buffer, dev)
    ({
        sprintf(buffer, "%u:%u", MAJOR(dev), MINOR(dev));
        buffer;
    })

/* acceptable for old filesystems */
static inline int old_valid_dev(dev_t dev)
{
    return MAJOR(dev) < 256 && MINOR(dev) < 256;
}/MKDEV
```

如上图所以，三个函数比较容易理解。

MAJOR(dev) , 就是对 dev 操作 , 提取高 12 位主设备号 ;

MINOR(dev) , 就是对 dev 操作 , 提取低 20 位数次设备号 ;

MKDEV(ma,mi) , 就是对主设备号和低设备号操作 , 合并为 dev 类型。

## 17.4 实验操作

在视频教程 “16\_驱动模块传参数” 的基础上做这个实验。

先修改一下 Makefile 文件 , 如下图所示 , 将 module\_param 改为 request\_cdev\_num 。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += request_cdev_num.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)} Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)} 当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

将视频教程“16\_驱动模块传参数”中的文件“module\_param.c”改写为

“request\_cdev\_num.c”。

如下图所示，先调用头文件，然后将主设备号和设备号通过模块参数传入，定义此设备号数。

```
/*包含初始化加载模块的头文件,代码中的MODULE_LICENSE在此头文件中*/  
  
/*定义module_param module_param_array的头文件*/  
#include <linux/moduleparam.h>  
/*定义module_param module_param_array中perm的头文件*/  
#include <linux/stat.h>  
  
/*三个字符设备函数*/  
#include <linux/fs.h>  
/*MKDEV转换设备号数据类型的宏定义*/  
#include <linux/kdev_t.h>  
/*定义字符设备的结构体*/  
#include <linux/cdev.h>  
  
#define DEVICE_NAME "sscdev"  
#define DEVICE_MINOR_NUM 2  
#define DEV_MAJOR 0  
#define DEV_MINOR 0  
  
MODULE_LICENSE("Dual BSD/GPL");  
/*声明是开源的，没有内核版本限制*/  
MODULE_AUTHOR("iTOPEET_dz");  
/*声明作者*/  
  
int numdev_major = DEV_MAJOR;  
int numdev_minor = DEV_MINOR;  
  
/*输入主设备号*/  
module_param(numdev_major,int,S_IRUSR);  
/*输入次设备号*/  
module_param(numdev_minor,int,S_IRUSR);
```

接着将入口函数和出口函数名称修改一下，“hello\_init” 和 “hello\_exit” 改为 “scdev\_init”

和 “scdev\_exit” 。

```
module_init(scdev_init);  
/*初始化函数*/  
module_exit(scdev_exit);  
/*卸载函数*/
```

如下图所示，在入口和出口函数中调用函数 register\_chrdev\_region 和函数 unregister\_chrdev\_region。

```
static int scdev_init(void)
{
    int ret = 0;
    dev_t num_dev;

    printk(KERN_EMERG "numdev_major is %d!\n", numdev_major);
    printk(KERN_EMERG "numdev_minor is %d!\n", numdev_minor);

    if(numdev_major){
        num_dev = MKDEV(numdev_major,numdev_minor);
        ret = register_chrdev_region(num_dev,DEVICE_MINOR_NUM,DEVICE_NAME);
    }
    else{
        printk(KERN_EMERG "numdev_major %d is failed!\n", numdev_major);
    }

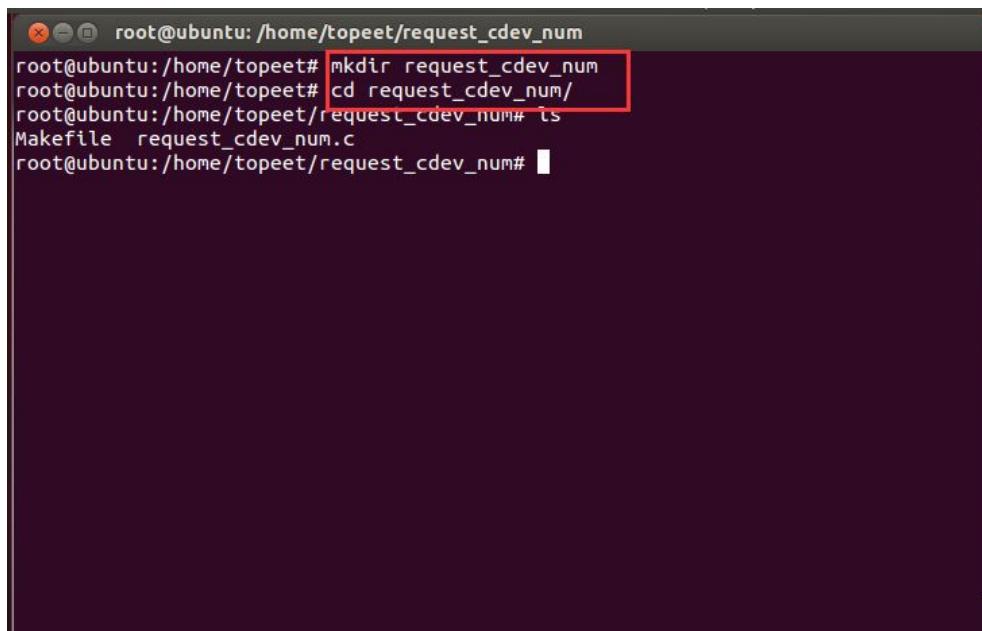
    if(ret<0){
        printk(KERN_EMERG "register_chrdev_region req %d is failed!\n", numdev_major);
    }

    printk(KERN_EMERG "scdev_init!\n");
    /*打印信息， KERN_EMERG表示紧急信息*/
    return 0;
}

static void scdev_exit(void)
{
    printk(KERN_EMERG "scdev_exit!\n");
    unregister_chrdev_region(MKDEV(numdev_major,numdev_minor),DEVICE_MINOR_NUM);
}
```

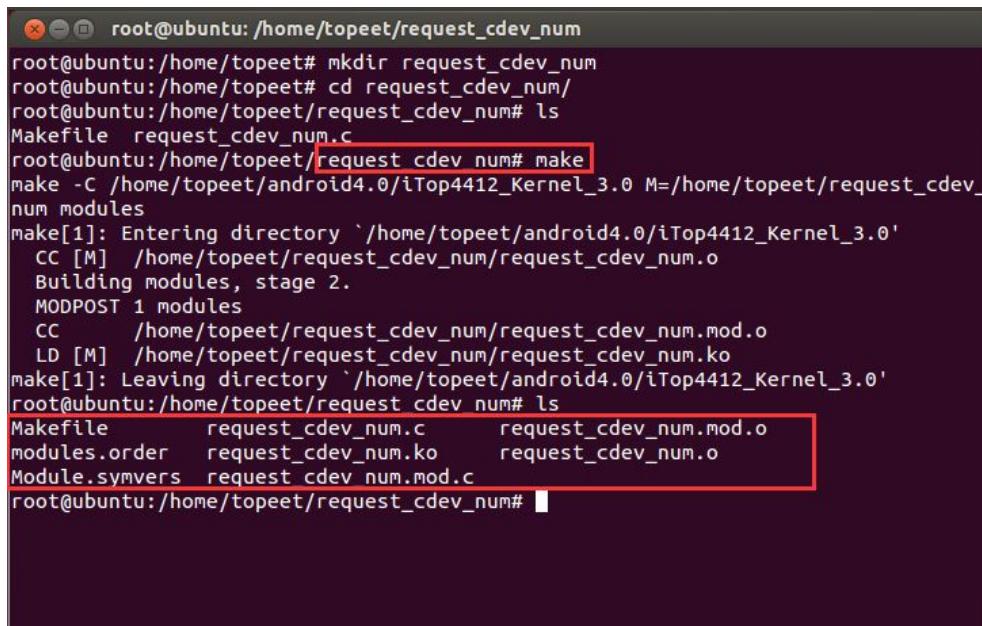
如上图所示，先将主设备号和次设备号默认为 0，然后做一个简单的判断。如果没有参数传入，默认为零，就会提示注册失败；如果参数传入的话，和已有的主次设备号有重复，也会失败。

在 Ubuntu 系统下新建 request\_cdev\_num 文件夹，将写好的 request\_cdev\_num.c、编译脚本拷贝到 request\_cdev\_num 文件夹下，如下图所示。



```
root@ubuntu:/home/topeet/request_cdev_num
root@ubuntu:/home/topeet# mkdir request_cdev_num
root@ubuntu:/home/topeet# cd request_cdev_num/
root@ubuntu:/home/topeet/request_cdev_num# ls
Makefile request_cdev_num.c
root@ubuntu:/home/topeet/request_cdev_num#
```

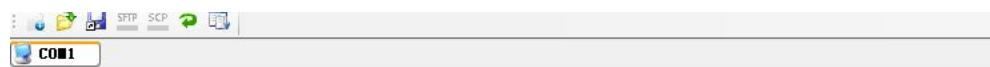
使用 Makefile 命令编译驱动命令 “Make” 编译应用，如下图所示。



```
root@ubuntu:/home/topeet/request_cdev_num
root@ubuntu:/home/topeet# mkdir request_cdev_num
root@ubuntu:/home/topeet# cd request_cdev_num/
root@ubuntu:/home/topeet/request_cdev_num# ls
Makefile request_cdev_num.c
root@ubuntu:/home/topeet/request_cdev_num# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/request_cdev_num modules
make[1]: Entering directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/request_cdev_num/request_cdev_num.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/request_cdev_num/request_cdev_num.mod.o
  LD [M] /home/topeet/request_cdev_num/request_cdev_num.ko
make[1]: Leaving directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/request_cdev_num# ls
Makefile      request_cdev_num.c      request_cdev_num.mod.o
modules.order  request_cdev_num.ko    request_cdev_num.o
Module.symvers request_cdev_num.mod.c
root@ubuntu:/home/topeet/request_cdev_num#
```

将上图中的文件 request\_cdev\_num.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘，如下图所示。



```
[root@iTOP-4412]# mount /dev/sdal /mnt/udisk/
[root@iTOP-4412]#
```

使用命令“cat /proc/devices”查看已经被注册的主设备号，发现设备号9没有被注册，也可以使用其它没有被使用的主设备号，如下图所示。

```
[root@iTOP-4412]# cat /proc/devices
Character devices:
1 mem
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
10 misc
13 input
21 sg
29 fb
81 video4linux
89 i2c
108 ppp
116 alsa
128 ptm
136 pts
153 rc522_test
166 ttyACM
180 usb
188 ttyUSB
189 usb_device
204 ttySAC
216 rfcomm
250 roccat
251 BaseRemoteCtl
252 media
253 ttyGS
254 rtc
```

这里使用设备号 9，使用加载模块的命令 “insmod /mnt/udisk/request\_cdev\_num.ko numdev\_major=9 numdev\_minor=0” 加载驱动 request\_cdev\_num.ko，如下图所示，加载成功。

```
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
254 device-mapper
[root@iTOP-4412]# insmod /mnt/udisk/request_cdev_num.ko numdev_major=9 numdev_mi
nor=0
[ 227.626029] numdev_major is 9!
[ 227.627628] numdev_minor is 0!
[ 227.630723] scdev_init!
```

加载之后可以再次使用命令 “cat /proc/devices” 查看，如下图所示，主设备号 9 已经被驱动所占用。

```
[root@iTOP-4412]# cat /proc/devices
Character devices:
  1 mem
  4 ttys
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  9 sscdev
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
108 ppp
116 alsa
128 ptm
136 pts
153 rc522_test
166 ttyACM
180 usb
188 ttyUSB
189 usb device
```



接着如下图所示，使用命令“rmmod request\_cdev\_num.ko”卸载模块，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]# lsmod
request_cdev_num 1429 0 - Live 0xbff000000
[root@iTOP-4412]# rmmod request_cdev_num
[ 379.431137] scdev_exit!
[root@iTOP-4412]#
```

然后再使用命令“cat /proc/devices”查看主设备号，可以看到设备号 9 又处于空闲状态了。

```
request_cdev_num 1429 0 - Live 0xbff000000
[root@iTOP-4412]# rmmod request_cdev_num
[ 379.431137] scdev_exit!
[root@iTOP-4412]# cat /proc/devices
Character devices:
1 mem
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
10 misc
13 input
21 sg
29 fb
81 video4linux
89 i2c
```

# 实验 18 动态申请字符类设备号

## 18.1 本章导读

前一期实验已经介绍了如何静态申请字符类设备号，这个实验给大家介绍如何动态申请设备号。

### 18.1.1 工具

#### 18.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 18.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “request\_ascdev\_num”

### 18.1.2 预备课程

实验 17\_静态申请字符类设备号

### 18.1.3 视频资源

本节配套视频为“视频 18\_动态申请字符类设备号”

## 18.2 学习目标

本章需要学习以下内容：

动态申请字符类设备号

## 18.3 实验操作

需要的基础知识在“实验 17\_静态申请字符类设备号”中都已经介绍过了，包括设备申请函数 alloc\_chrdev\_region()，dev\_t 设备号，以及头文件“include/linux/fs.h”。

这里简单介绍一下动态申请函数。

```
extern int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *);
```

函数有四个参数

参数\*dev，存放返回的设备号；

参数 unsigned，一般为 0；

参数 unsigned，次设备号连续编号范围；

参数 const char \*，设备名称；

函数调用成功则返回 0,反之返回-1。

将“实验 17\_静态申请字符类设备号”中的文件“request\_cdev\_num.c”改为“request\_ascdev\_num.c”，然后添加动态申请设备号的代码，如下图所示。

```
static int scdev_init(void)
{
    int ret = 0;
    dev_t num_dev;

    printk(KERN_EMERG "numdev_major is %d!\n", numdev_major);
    printk(KERN_EMERG "numdev_minor is %d!\n", numdev_minor);

    if(numdev_major) {
        num_dev = MKDEV(numdev_major,numdev_minor);
        ret = register_chrdev_region(num_dev,DEVICE_MINOR_NUM,DEVICE_NAME);
    }
    else{
        /*动态注册设备号*/
        ret = alloc_chrdev_region(&num_dev,numdev_minor,DEVICE_MINOR_NUM,DEVICE_NAME);
        /*获得主设备号*/
        numdev_major = MAJOR(num_dev);
        printk(KERN_EMERG "adev_region req %d !\n",numdev_major);
    }

    if(ret<0){
        printk(KERN_EMERG "register_chrdev_region req %d is failed!\n",numdev_major);
    }

    printk(KERN_EMERG "scdev_init!\n");
    /*打印信息，KERN_EMERG表示紧急信息*/
    return 0;
}
```

将设备节点宏定义由“sscdev”改为“ascdev”，如下图所示。

```
#define DEVICE_NAME "ascdev"
#define DEVICE_MINOR_NUM 2
#define DEV_MAJOR 0
#define DEV_MINOR 0

MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/

int numdev_major = DEV_MAJOR;
int numdev_minor = DEV_MINOR;

/*输入主设备号*/
```

然后修改一下 Makefile 文件，如下图所示，将 “request\_cdev\_num” 改为 “request\_ascdev\_num”。

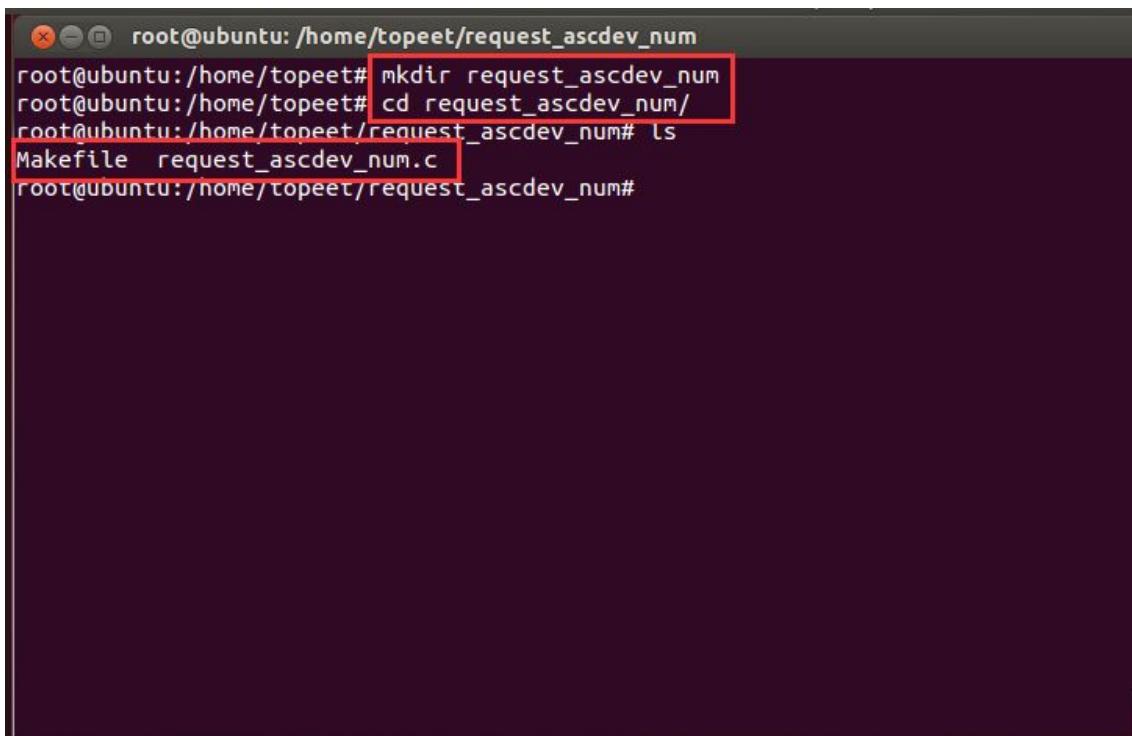
```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += request_ascdev_num.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

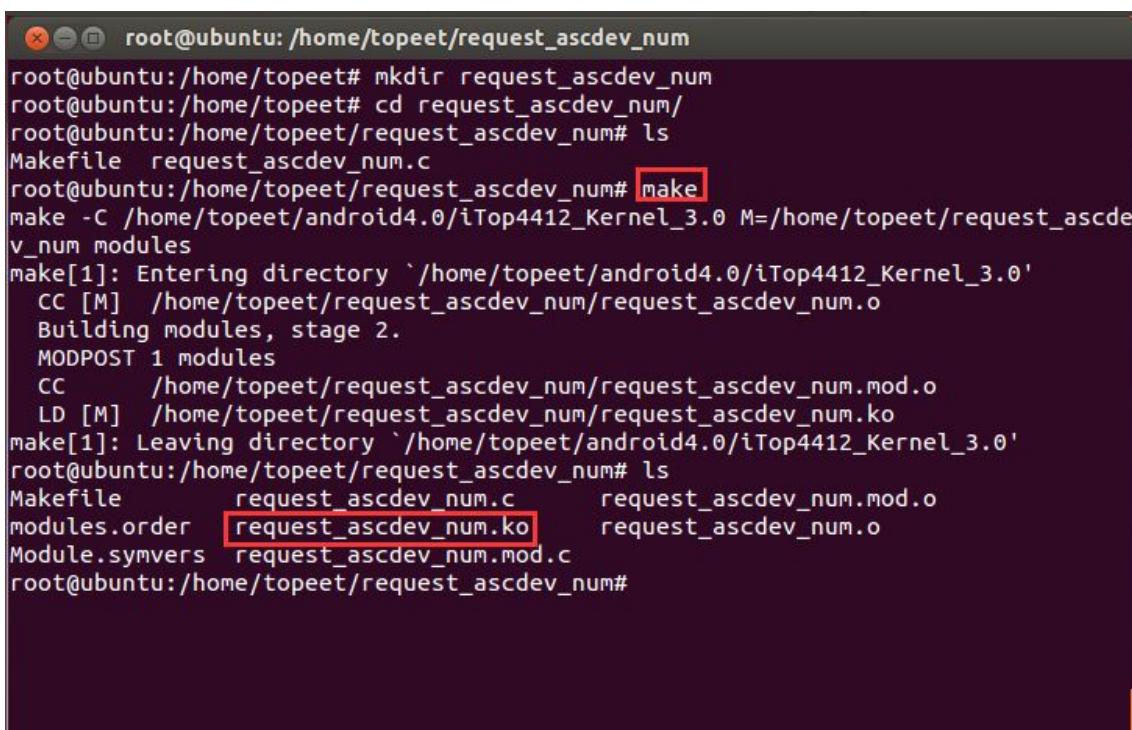
#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR)Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD)当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules
    |
#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

在 Ubuntu 系统下，使用命令 “mkdir request\_ascdev\_num” 新建文件夹 “request\_ascdev\_num”，将写好的文件 request\_ascdev\_num.c、编译脚本拷贝到 module\_param 文件夹下，如下图所示。



```
root@ubuntu:/home/topeet/request_ascdev_num
root@ubuntu:/home/topeet# mkdir request_ascdev_num
root@ubuntu:/home/topeet# cd request_ascdev_num/
root@ubuntu:/home/topeet/request_ascdev_num# ls
Makefile  request_ascdev_num.c
root@ubuntu:/home/topeet/request_ascdev_num#
```

使用 Makefile 命令编译驱动命令 “Make” 编译应用，如下图所示，生成驱动模块 “request\_ascdev\_num.ko”。



```
root@ubuntu:/home/topeet/request_ascdev_num
root@ubuntu:/home/topeet# mkdir request_ascdev_num
root@ubuntu:/home/topeet# cd request_ascdev_num/
root@ubuntu:/home/topeet/request_ascdev_num# ls
Makefile  request_ascdev_num.c
root@ubuntu:/home/topeet/request_ascdev_num# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/request_ascdev_num modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M]  /home/topeet/request_ascdev_num/request_ascdev_num.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/request_ascdev_num/request_ascdev_num.mod.o
  LD [M]  /home/topeet/request_ascdev_num/request_ascdev_num.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/request_ascdev_num# ls
Makefile      request_ascdev_num.c      request_ascdev_num.mod.o
modules.order  request_ascdev_num.ko    request_ascdev_num.o
Module.symvers request_ascdev_num.mod.c
root@ubuntu:/home/topeet/request_ascdev_num#
```

将上图中的文件 request\_ascdev\_num.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘，然后使用命令 “insmod /mnt/udisk/request\_ascdev\_num.ko” 加载模块 request\_ascdev\_num.ko，如下图所示。

```
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/request_ascdev_num.ko
[ 71.273612] numdev_major is 0!
[ 71.275311] numdev_minor is 0!
[ 71.278379] adev_region_req 249 !
[ 71.289533] scdev_init!
```

使用命令 “cat /proc/devices” 查看已经被注册的主设备号，如下图所示。

```
[    71.278379] adev_region req 249 !
[    71.289533] scdev init!
[root@iTOP-4412]# cat /proc/devices
Character devices:
  1 mem
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
108 ppp
116 alsa
128 ptm
136 pts
153 rc522_test
166 ttyACM
180 usb
188 ttyUSB
189 usb_device
204 ttySAC
216 rfcomm
249 ascdev
250 roccat
251 BaseRemoteCtl
252 media
253 ttyGS
254 rtc
```

如上图所示，可以看到主设备号 249 已经动态被申请到。

这个模块也是可以通过模块参数来静态配置，和上一期实验中的用法一样，这里就不重复介绍了。

# 实验 19 注册字符类设备

## 19.1 本章导读

前面几期详细介绍了如何申请设备号，这一期实验介绍如何注册字符类设备。

前面介绍的注册杂项设备，在这一步的处理非常简单，只要在平台文件中添加一个结构体和一个指针调用即可。字符设备和前面的杂项设备唯一的区别就是多了这一步注册设备，难度并不大。

### 19.1.1 工具

#### 19.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 19.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “register\_cdev”

## 19.1.2 预备课程

实验 17 静态申请字符类设备号

实验 18 动态申请字符类设备号

## 19.1.3 视频资源

本节配套视频为“视频 19\_注册字符类设备”

## 19.2 学习目标

本章需要学习以下内容：

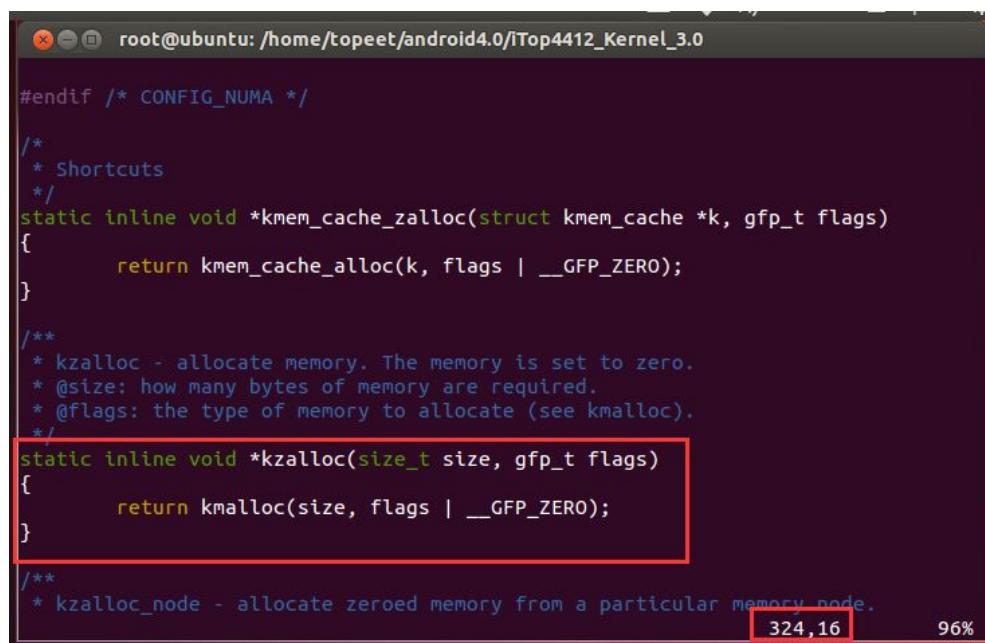
给设备分配内存空间

注册字符类设备的整个过程

## 19.3 分配内存空间

前面介绍的杂项设备并没有分配内存空间这个过程，是因为系统自带的代码已经给杂项设备分配好了。

Linux 中注册字符类设备需要首先申请内存空间，有一个专门分配小内存空间的函数 kmalloc，这个函数在头文件“include/linux/slab.h”中，如下图所示，使用命令“vim include/linux/slab.h”打开头文件。



```
#endif /* CONFIG_NUMA */

/*
 * Shortcuts
 */
static inline void *kmem_cache_zalloc(struct kmem_cache *k, gfp_t flags)
{
    return kmem_cache_alloc(k, flags | __GFP_ZERO);
}

/**
 * kzalloc - allocate memory. The memory is set to zero.
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate (see kmalloc).
 */
static inline void *kzalloc(size_t size, gfp_t flags)
{
    return kmalloc(size, flags | __GFP_ZERO);
}

/**
 * kzalloc_node - allocate zeroed memory from a particular memory node.

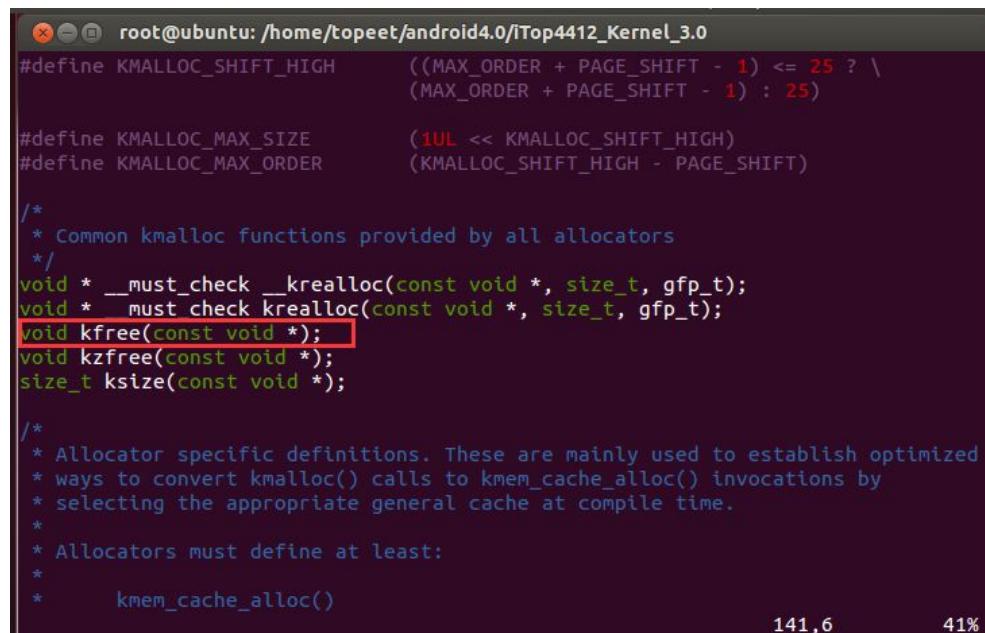
```

如上图所示，函数 static inline void \*kzalloc(size\_t size, gfp\_t flags)有两个参数，

参数 size\_t size：申请的内存大小(最大 128K) ,

参数 gfp\_t flags：常用参数 GFP\_KERNEL，代表优先权，内存不够可以延迟分配。

和申请内存的函数 kzalloc 对应的是 kfree 释放函数，如下图所示。



```
#define KMALLOC_SHIFT_HIGH      ((MAX_ORDER + PAGE_SHIFT - 1) <= 25 ? \
                                (MAX_ORDER + PAGE_SHIFT - 1) : 25)

#define KMALLOC_MAX_SIZE         (1UL << KMALLOC_SHIFT_HIGH)
#define KMALLOC_MAX_ORDER        (KMALLOC_SHIFT_HIGH - PAGE_SHIFT)

/*
 * Common kmalloc functions provided by all allocators
 */
void * __must_check __krealloc(const void *, size_t, gfp_t);
void * must_check_krealloc(const void *, size_t, gfp_t);
void kfree(const void *);
void kzfree(const void *);
size_t ksize(const void *);

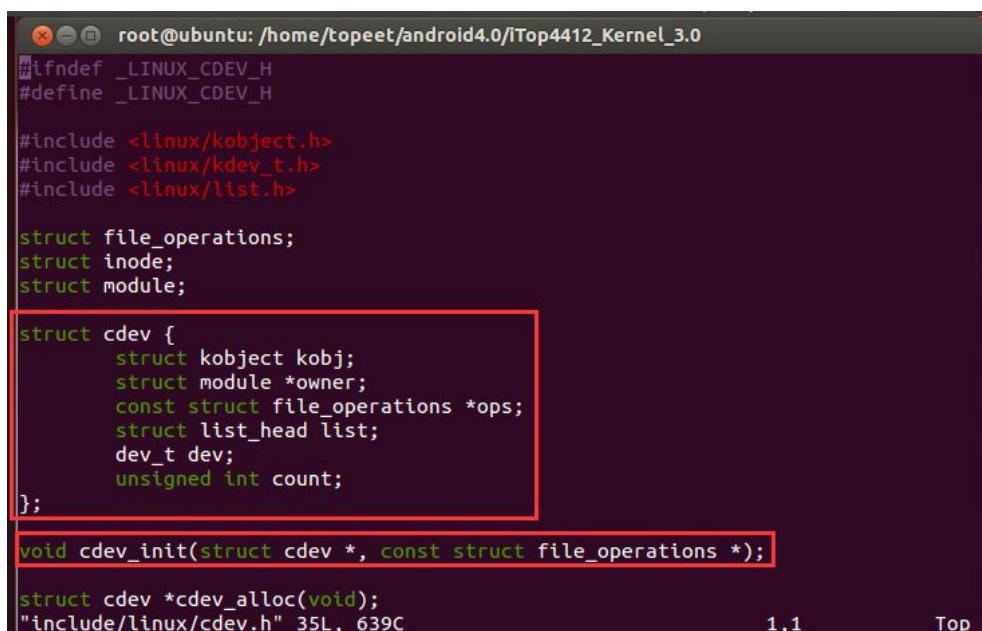
/*
 * Allocator specific definitions. These are mainly used to establish optimized
 * ways to convert kmalloc() calls to kmem_cache_alloc() invocations by
 * selecting the appropriate general cache at compile time.
 *
 * Allocators must define at least:
 *
 *     kmem_cache_alloc()

```

void kfree(const void \*)函数只有一个参数，就是内存的指针，这个指针由申请内存的函数 kzalloc 返回。

## 19.4 注册字符类设备的函数

注册字符类设备的初始化函数为 cdev\_init，这个函数在头文件 “include/linux/cdev.h” 中，使用命令 “vim include/linux/cdev.h” 打开这个头文件如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
#ifndef _LINUX_CDEV_H
#define _LINUX_CDEV_H

#include <linux/kobject.h>
#include <linux/kdev_t.h>
#include <linux/list.h>

struct file_operations;
struct inode;
struct module;

struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};

void cdev_init(struct cdev *, const struct file_operations *);

struct cdev *cdev_alloc(void);
"include/linux/cdev.h" 35L, 639C
```

如上图所示红框中的函数“

void cdev\_init(struct cdev \*, const struct file\_operations \*)”

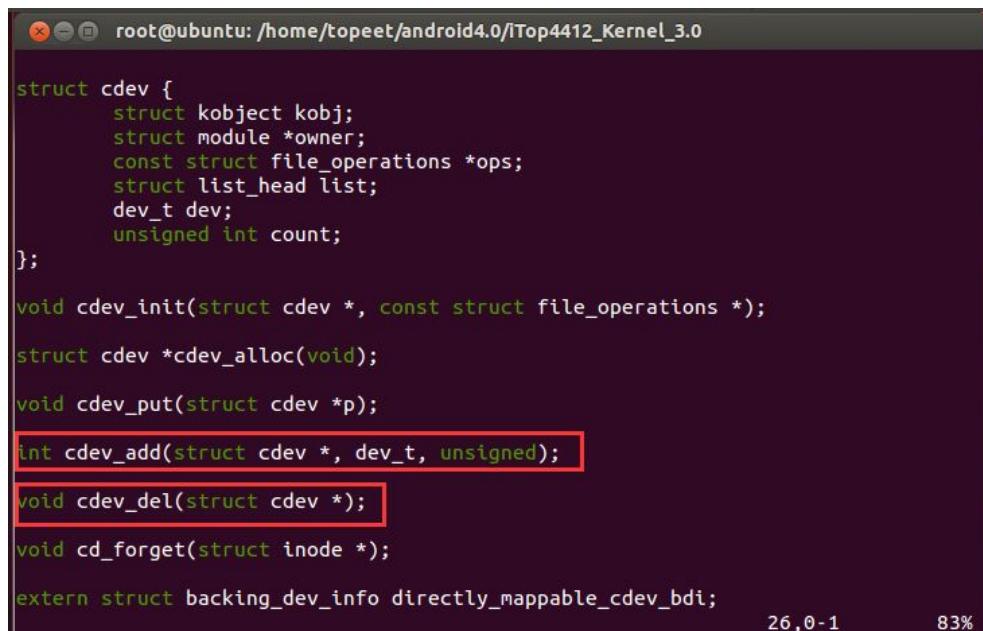
和结构体 “cdev”。

cdev\_init 函数有两个参数，

参数 struct cdev \* : cdev 字符设备文件结构体

参数 const struct file\_operations \* : file\_operations 结构体，这个结构体已经用过很多次了，就不再介绍了。

注册字符设备的函数为 cdev\_add，这个函数也是在头文件 “include/linux/cdev.h” 中，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};

void cdev_init(struct cdev *, const struct file_operations *);

struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);

int cdev_add(struct cdev *, dev_t, unsigned); int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
void cd_forget(struct inode *);

extern struct backing_dev_info directly_mappable_cdev_bdi;
```

26,0-1

83%

如上图所示，和注册驱动的函数 cdev\_add 对应的还有卸载驱动的函数 cdev\_del。

注册驱动的函数 int cdev\_add(struct cdev \*, dev\_t, unsigned);有三个参数：

参数 struct cdev \* : cdev 字符设备文件结构体

参数 dev\_t : 设备号 dev,前面已经介绍和使用过了

参数 unsigned : 设备范围大小

卸载驱动的函数 void cdev\_del(struct cdev \*)只有一个参数，cdev 字符设备结构体

## 19.5 实验操作

将“18\_动态申请字符类设备号”中的文件“request\_ascdev\_num.c”改为“register\_cdev.c”。

首先修改一下 Makefile 文件，如下图所示，将“request\_ascdev\_num”改为“register\_cdev”。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += register_cdev.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

如下图所示，添加申请内存空间函数的头文件“linux/slab.h”，然后定义内存空间大小为3000。

```
#include <linux/cdev.h>

/*分配内存空间函数头文件*/
#include <linux/slab.h>

#define DEVICE_NAME "ascdev"
#define DEVICE_MINOR_NUM 2
#define DEV_MAJOR 0
#define DEV_MINOR 0
#define REGDEV_SIZE 3000

MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/
```

接着将申请的内存数据控件打个包为结构体 reg\_dev，因为是有两个子设备，所以接着定义一个结构体数组\*my\_devices。

```
module_param(numdev_major,int,S_IRUSR);
/*输入次设备号*/
module_param(numdev_minor,int,S_IRUSR);

struct reg_dev
{
    char *data;
    unsigned long size;

    struct cdev cdev;
};

struct reg_dev *my_devices;
```

然后申请内存空间，并将内存空间先清零，接着对设备进行初始化，如下图所示。

```
if(ret<0){  
    printk(KERN_EMERG "register_chrdev_region req %d is failed!\n",numdev_major);  
}  
  
my_devices = kmalloc(DEVICE_MINOR_NUM * sizeof(struct reg_dev), GFP_KERNEL);  
if(!my_devices){  
    ret = -ENOMEM;  
    goto fail;  
}  
memset(my_devices, 0, DEVICE_MINOR_NUM * sizeof(struct reg_dev));  
  
/*设备初始化*/  
for(i=0;i<DEVICE_MINOR_NUM;i++){  
    my_devices[i].data = kmalloc(REGDEV_SIZE,GFP_KERNEL);  
    memset(my_devices[i].data, 0, REGDEV_SIZE);  
    /*设备注册到系统*/  
    reg_init_cdev(&my_devices[i],i);  
}
```

如上图所示，用到多次 memset 函数，第一次由于没有规定 my\_devices[i].data 的大小，所以只是对默认大小的数据赋值为 0，在设备初始化的循环中，又重新对 my\_devices[i].data 申请了 REGDEV\_SIZE 大小的数据，所以需要重新赋值为 0。

然后如下图所示，是自定义的函数 reg\_init\_cdev，在这个函数中会将 my\_devices 和编号传过去。

```
struct file_operations my_fops = {
    .owner = THIS_MODULE,
};

/*设备注册到系统*/
static void reg_init_cdev(struct reg_dev *dev,int index){
    int err;
    int devno = MKDEV(numdev_major,numdev_minor+index);

    /*数据初始化*/
    cdev_init(&dev->cdev,&my_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &my_fops;

    /*注册到系统*/
    err = cdev_add(&dev->cdev,devno,1);
    if(err){
        printk(KERN_EMERG "cdev_add %d is fail! %d\n",index,err);
    }
    else{
        printk(KERN_EMERG "cdev_add %d is success!\n",index);
    }
}
```

如上图所示，首先需要将设备号使用 MKDEV 转化成 dev\_t 类型，然后初始化，接着给 dev->cdev 赋值。

这里只给 cdev 添加了两个参数 owner 和 ops，

参数 owner 默认为 THIS\_MODULE，

参数 ops 这里只定义了其中的参数 owner。

接着添加一下 fail 部分，如下图所示。

```
printk(KERN_EMERG "scdev_init!\n");
/*打印信息，KERN_EMERG表示紧急信息*/
return 0;

fail:
/*注销设备号*/
unregister_chrdev_region(MKDEV(numdev_major,numdev_minor),DEVICE_MINOR_NUM);
printk(KERN_EMERG "kmalloc is fail!\n");

return ret;
}
```

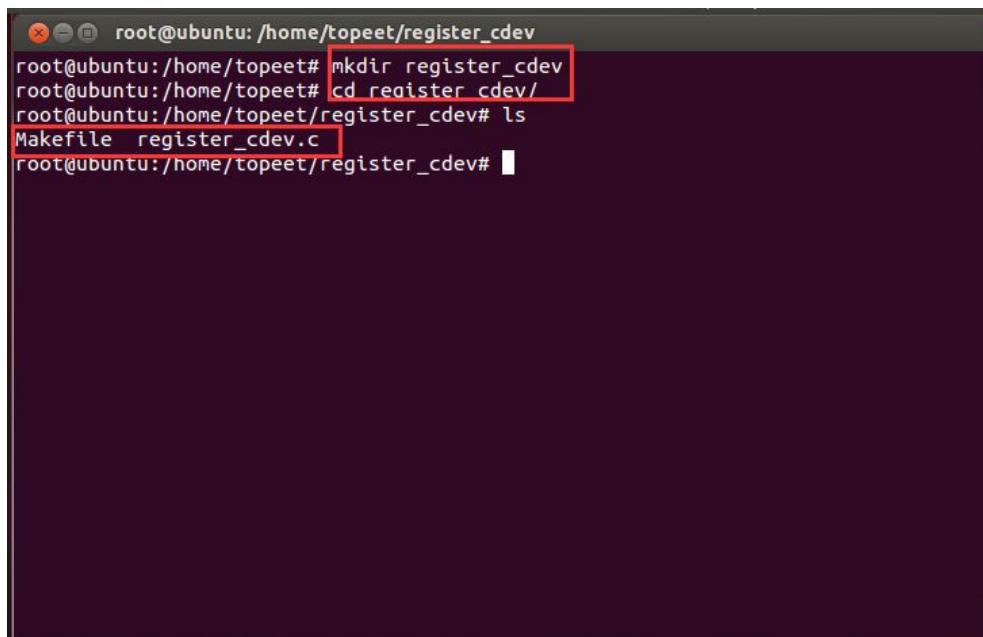
最后添加模块出口函数的代码，如下图所示。

```
static void scdev_exit(void)
{
    int i;
    printk(KERN_EMERG "scdev_exit!\n");

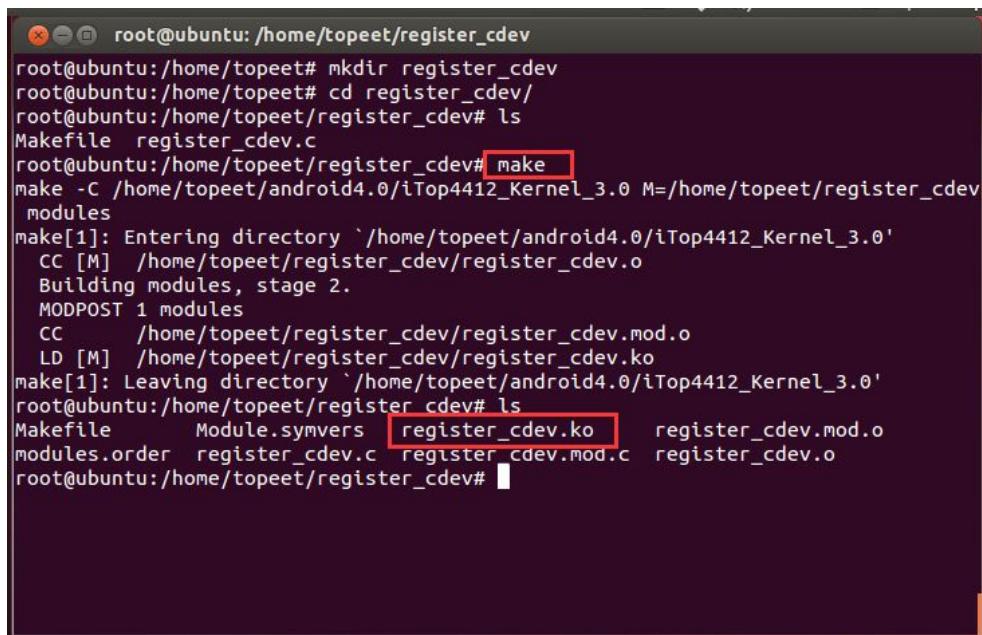
    /*除去字符设备*/
    for(i=0;i<DEVICE_MINOR_NUM;i++){
        cdev del(&(my_devices[i].cdev));
    }

    unregister_chrdev_region(MKDEV(numdev_major,numdev_minor),DEVICE_MINOR_NUM);
}
```

修改完之后，在 Ubuntu 系统下使用命令 “mkdir register\_cdev” 新建 “register\_cdev” 文件夹，将写好的 register\_cdev.c、编译脚本拷贝到 register\_cdev 文件夹下，如下图所示。



使用 Makefile 命令编译驱动命令 “Make” 编译应用，如下图所示。



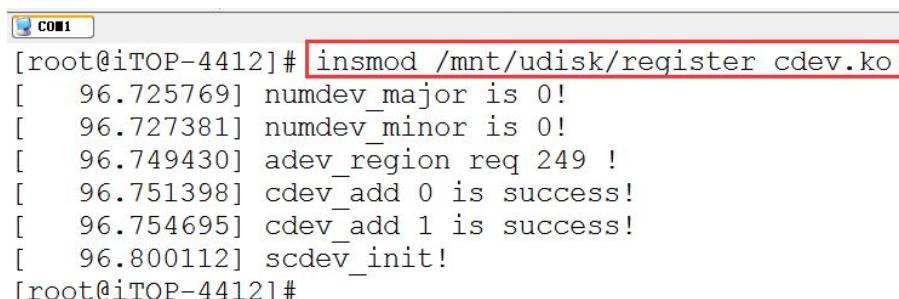
```
root@ubuntu:/home/topeet/register_cdev
root@ubuntu:/home/topeet# mkdir register_cdev
root@ubuntu:/home/topeet# cd register_cdev/
root@ubuntu:/home/topeet/register_cdev# ls
Makefile  register_cdev.c
root@ubuntu:/home/topeet/register_cdev# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/register_cdev
modules
make[1]: Entering directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
CC [M] /home/topeet/register_cdev/register_cdev.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/topeet/register_cdev/register_cdev.mod.o
LD [M] /home/topeet/register_cdev/register_cdev.ko
make[1]: Leaving directory '/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/register_cdev# ls
Makefile      Module.symvers  register_cdev.ko      register_cdev.mod.o
modules.order  register_cdev.c  register_cdev.mod.c  register_cdev.o
root@ubuntu:/home/topeet/register_cdev#
```

将上图中的文件 register\_cdev.ko 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
```

如下图所示，使用命令 “insmod /mnt/udisk/register\_cdev.ko” 加载驱动。



```
[root@iTOP-4412]# insmod /mnt/udisk/register_cdev.ko
[ 96.725769] numdev_major is 0!
[ 96.727381] numdev_minor is 0!
[ 96.749430] adev_region req 249 !
[ 96.751398] cdev_add 0 is success!
[ 96.754695] cdev_add 1 is success!
[ 96.800112] scdev_init!
[root@iTOP-4412]#
```

可以看到根据上图中的打印信息判断，设备注册已经成功。

# 实验 20 生成字符类设备节点

## 20.1 本章导读

本实验介绍一下如何生成字符设备的设备节点，这部分和前面注册杂项设备类似，不过在调用生成设备节点的时候需要额外的添加一个设备类。

### 20.1.1 工具

#### 20.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 20.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “create\_cnode”

### 20.1.2 预备课程

#### 实验 19 注册字符类设备

### 20.1.3 视频资源

本节配套视频为“视频 20\_生成字符类设备节点”

## 20.2 学习目标

本章需要学习以下内容：

了解设备类的概念

初始化代码中创建设备节点

手动创建设备节点

## 20.3 创建设备类

在前面介绍的设备中的模型，例如总线 bus、设备 device、驱动 driver 都是有明确的定义。。

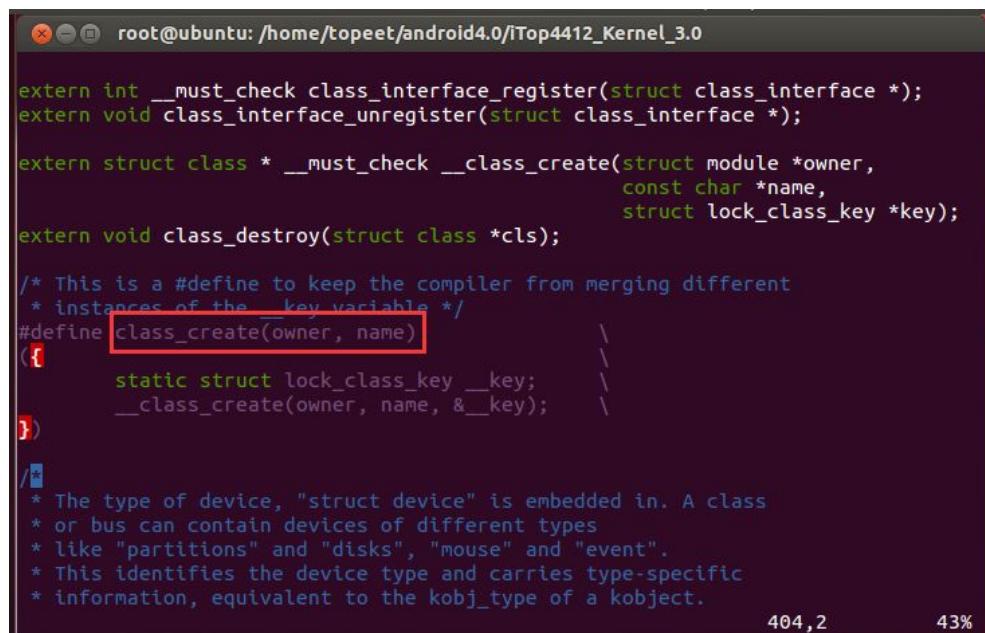
bus 代表总线，device 代表实际的设备和接口，driver 代表驱动。

Linux 中的 class 是设备类，它是一个抽象的概念，没有对应的实体。它是提供给用户接口相似的一类设备的集合。常见的有输入子系统 input、usb、串口 tty、块设备 block 等。

以 4412 的串口为例，它有四个串口，不可能为每一个串口都重复申请设备以及设备节点，因为它们有类似的地方，而且很多代码都是重复的地方，所以引入了一个抽象的类，将其打包为 ttySACX，在实际调用串口的时候，只需要修改 X 值，就可以调用不同的串口。

对于本实验中的设备，它有两个子设备，将对应两个设备节点，所以需要用到 class 类这样一个概念。

创建设备类的函数 class\_create 在头文件 “include/linux/device.h” 中，使用命令 “vim include/linux/device.h” 打开头文件，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
extern int __must_check class_interface_register(struct class_interface *);
extern void class_interface_unregister(struct class_interface *);

extern struct class * __must_check __class_create(struct module *owner,
                                                 const char *name,
                                                 struct lock_class_key *key);

extern void class_destroy(struct class *cls);

/* This is a #define to keep the compiler from merging different
 * instances of the _key variable */
#define class_create(owner, name)
({ \
    static struct lock_class_key __key; \
    __class_create(owner, name, &__key); \
})

/*
 * The type of device, "struct device" is embedded in. A class
 * or bus can contain devices of different types
 * like "partitions" and "disks", "mouse" and "event".
 * This identifies the device type and carries type-specific
 * information, equivalent to the kobj_type of a kobject.

```

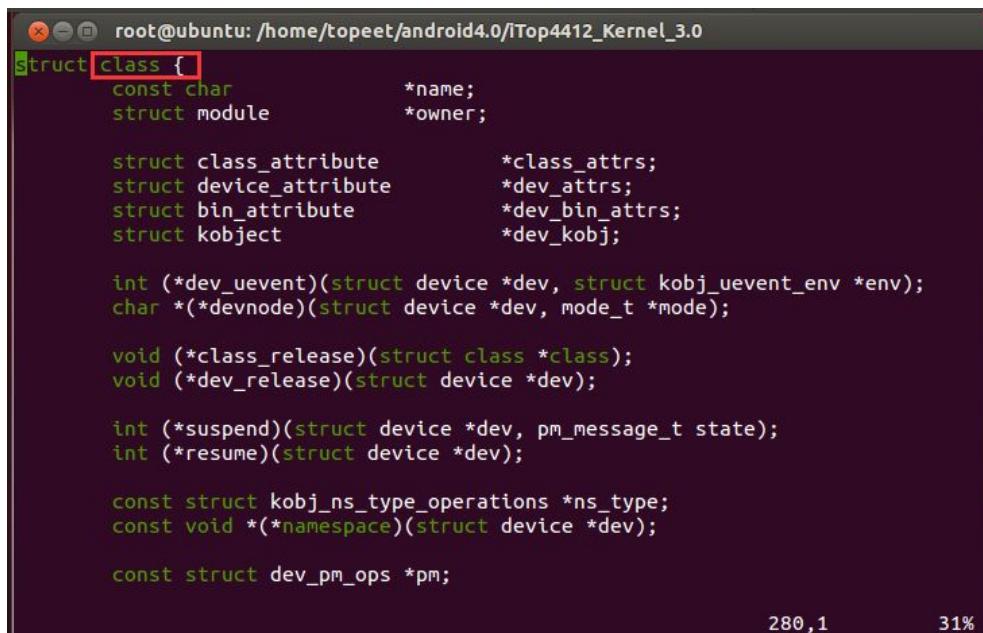
如上图所示，函数 class\_create(owner, name) 只有两个参数

参数 owner : 一般是 THIS\_MODULE

参数 name : 设备名称

调用函数 class\_create 会返回一个 class 结构体变量

class 结构体变量在头文件 include/linux/device.h 的 280 行，如下图所示。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
struct class {
    const char          *name;
    struct module       *owner;

    struct class_attribute      *class_attrs;
    struct device_attribute     *dev_attrs;
    struct bin_attribute       *dev_bin_attrs;
    struct kobject             *dev_kobj;

    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);
    char *(*devnode)(struct device *dev, mode_t *mode);

    void (*class_release)(struct class *class);
    void (*dev_release)(struct device *dev);

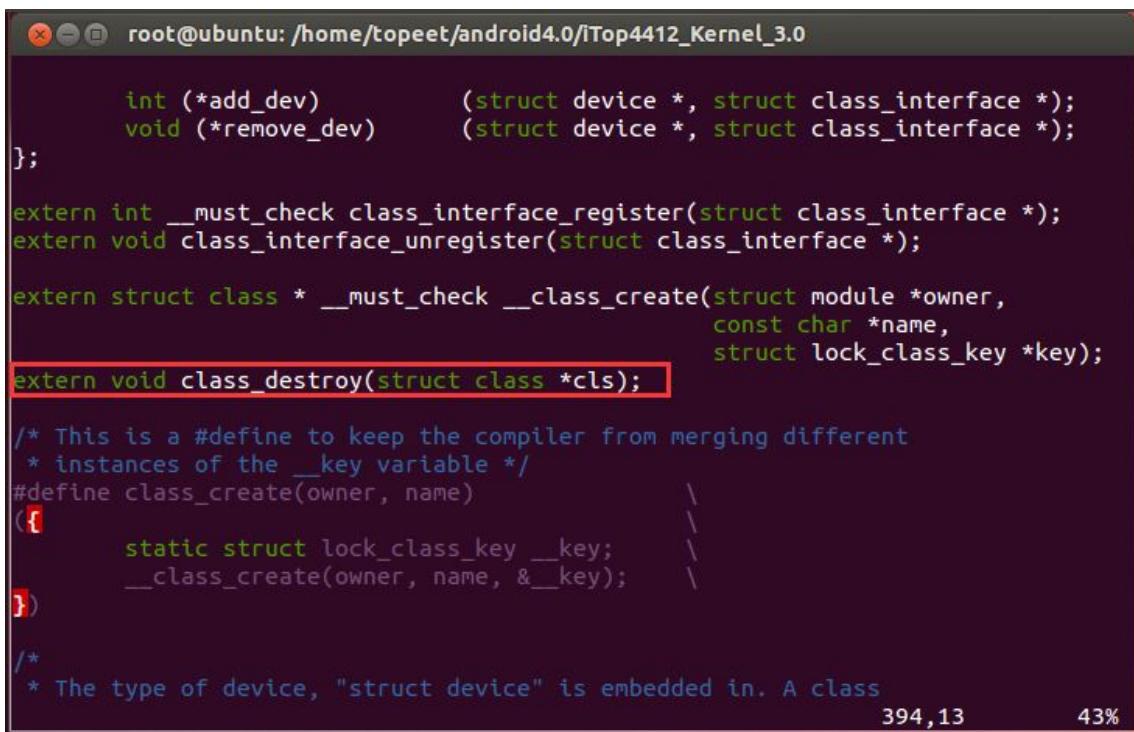
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct kobj_ns_type_operations *ns_type;
    const void *(*namespace)(struct device *dev);

    const struct dev_pm_ops *pm;
}
```

如上图所示，看着复杂，其实不用管，在实际应用中就是给创建设备节点用的。在代码中，只需要定义一个 class 变量，然后在创建设备节点的时候作为一个参数赋值使用即可。

还有释放设备类 class 的函数 class\_destroy，就只有一个参数 class。这个函数也是在头文件“include/linux/device.h”中，如下图所示。



The screenshot shows a terminal window with the following content:

```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0

    int (*add_dev)      (struct device *, struct class_interface *);
    void (*remove_dev)  (struct device *, struct class_interface *);
};

extern int __must_check class_interface_register(struct class_interface *);
extern void class_interface_unregister(struct class_interface *);

extern struct class * __must_check __class_create(struct module *owner,
                                                 const char *name,
                                                 struct lock_class_key *key);

extern void class_destroy(struct class *cls);

/* This is a #define to keep the compiler from merging different
 * instances of the __key variable */
#define class_create(owner, name)
({ \
    static struct lock_class_key __key; \
    __class_create(owner, name, &__key); \
})

/*
 * The type of device, "struct device" is embedded in. A class
 */

394,13          43%
```

## 20.4 创建字符设备节点

创建设备节点的函数 `device_create` 在头文件 “`include/linux/device.h`” , 如下图所示。

The screenshot shows a terminal window with the following kernel source code:

```
* for information on use.  
*/  
extern int __must_check device_bind_driver(struct device *dev);  
extern void device_release_driver(struct device *dev);  
extern int __must_check device_attach(struct device *dev);  
extern int __must_check driver_attach(struct device_driver *drv);  
extern int __must_check device_reprobe(struct device *dev);  
  
/*  
 * Easy functions for dynamically creating devices on the fly  
 */  
extern struct device *device_create_vargs(struct class *cls,  
                                         struct device *parent,  
                                         dev_t devt,  
                                         void *drvdata,  
                                         const char *fmt,  
                                         va_list args);  
extern struct device *device_create(struct class *cls, struct device *parent,  
                                   dev_t devt, void *drvdata,  
                                   const char *fmt, ...)  
    __attribute__((format(sprintf, 5, 6)));  
extern void device_destroy(struct class *cls, dev_t devt);
```

Terminal status bar: 738, 3-24 82%

函数 `extern struct device *device_create(struct class *cls, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...);` 中的参数比较多。

参数 `struct class *cls` : 设备所属于的类 , 前面创建类的返回值

参数 `struct device *parent` : 设备的父设备 , NULL

参数 `dev_t devt` : 设备号

参数 `void *drvdata` : 设备数据 , NULL

参数 `const char *fmt` : 设备名称

如上图所示 , 还有一个摧毁设备节点的函数 `extern void device_destroy(struct class *cls, dev_t devt);` ; 只有两个参数 , 分别是设备类和设备号。

## 20.5 实验操作

将“19\_注册字符类设备”中的“register\_cdev.c”文件改为“create\_cnode.c”，然后添加注册类以及创建设备节点的代码。

如下图所示，首先添加头文件“linux/device.h”，然后将设备名称改为chardevnode。

```
/*定义字符设备的结构体*/
#include <linux/cdev.h>
/*分配内存空间函数头文件*/
#include <linux/slab.h>

/*包含函数device_create 结构体class等头文件*/
#include <linux/device.h>

#define DEVICE_NAME "chardevnode"
#define DEVICE_MINOR_NUM 2
#define DEV_MAJOR 0
#define DEV_MINOR 0
#define REGDEV_SIZE 3000

MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/
```

然后定义一个设备类，如下图所示。

```
MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/

int numdev_major = DEV_MAJOR;
int numdev_minor = DEV_MINOR;

/*输入主设备号*/
module_param(numdev_major,int,S_IRUSR);
/*输入次设备号*/
module_param(numdev_minor,int,S_IRUSR);

static struct class *myclass;

struct reg_dev
{
```

如下图所示，在初始化函数中添加申请设备类以及创建设备节点的代码。

```
    }
    myclass = class_create(THIS_MODULE,DEVICE_NAME);

    my_devices = kmalloc(DEVICE_MINOR_NUM * sizeof(struct reg_dev), GFP_KERNEL);
    if(!my_devices) {
        ret = -ENOMEM;
        goto fail;
    }
    memset(my_devices, 0, DEVICE_MINOR_NUM * sizeof(struct reg_dev));

    /*设备初始化*/
    for(i=0; i<DEVICE_MINOR_NUM; i++) {
        my_devices[i].data = kmalloc(REGDEV_SIZE, GFP_KERNEL);
        memset(my_devices[i].data, 0, REGDEV_SIZE);
        /*设备注册到系统*/
        reg_init_cdev(&my_devices[i], i);

        /*创建设备节点*/
        device_create(myclass, NULL, MKDEV(numdev_major, numdev_minor+i), NULL, DEVICE_NAME"%d", i);
    }
}
```

最后在退出函数中添加释放设备以及释放内存的代码，如下图所示。

```

static void scdev_exit(void)
{
    int i;
    printk(KERN_EMERG "scdev_exit!\n");

    /*除去字符设备*/
    for(i=0;i<DEVICE_MINOR_NUM;i++) {
        cdev_del(&(my_devices[i].cdev));
        /*摧毁设备节点函数*/
        device_destroy(myclass,MKDEV(numdev_major,numdev_minor+i));
    }
    /*释放设备class*/
    class_destroy(myclass);
    /*释放内存*/
    kfree(my_devices);

    unregister_chrdev_region(MKDEV(numdev_major,numdev_minor),DEVICE_MINOR_NUM);
}

```

然后修改一下 Makefile 编译文件，如下图所示。

```

#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += create_cnode.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

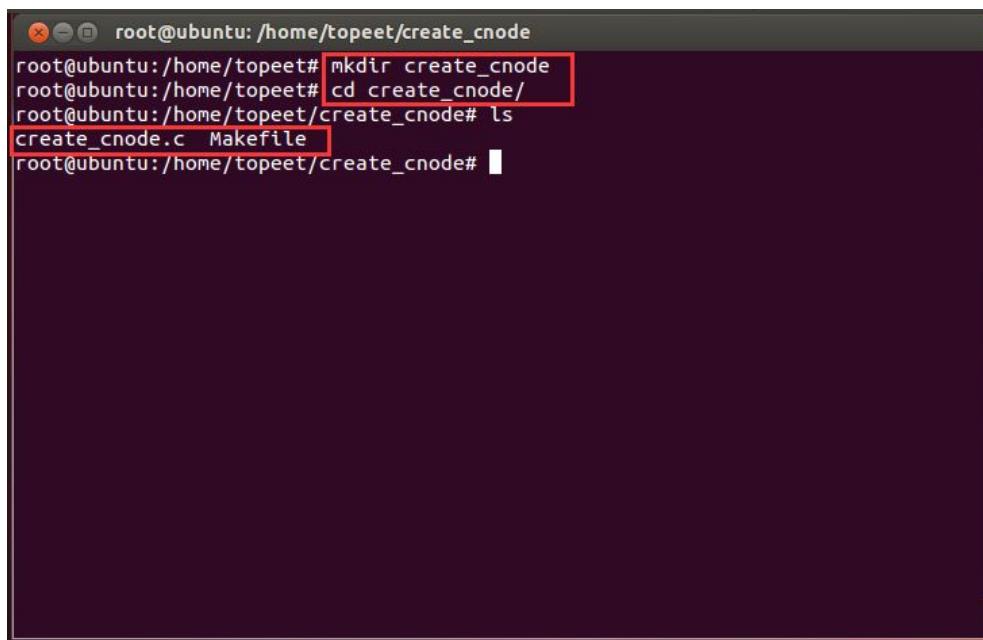
#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers

```

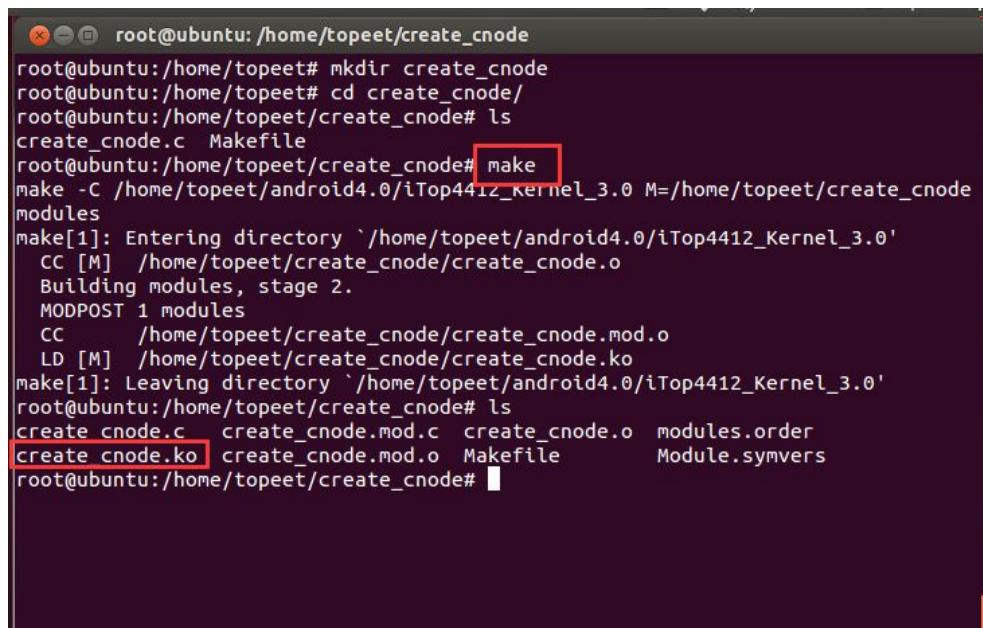
修改完成之后，在 Ubuntu 系统下使用命令 “mkdir create\_cnode” 新建文件夹

“create\_cnode” ，然后将修改好的驱动文件 “create\_cnode.c” 和 Makefile 文件拷贝到 “create\_cnode” 中，如下图所示。



```
root@ubuntu:/home/topeet/create_cnode
root@ubuntu:/home/topeet# mkdir create_cnode
root@ubuntu:/home/topeet# cd create_cnode/
root@ubuntu:/home/topeet/create_cnode# ls
create_cnode.c Makefile
root@ubuntu:/home/topeet/create_cnode#
```

使用编译命令“make” 编译驱动，如下图所示。



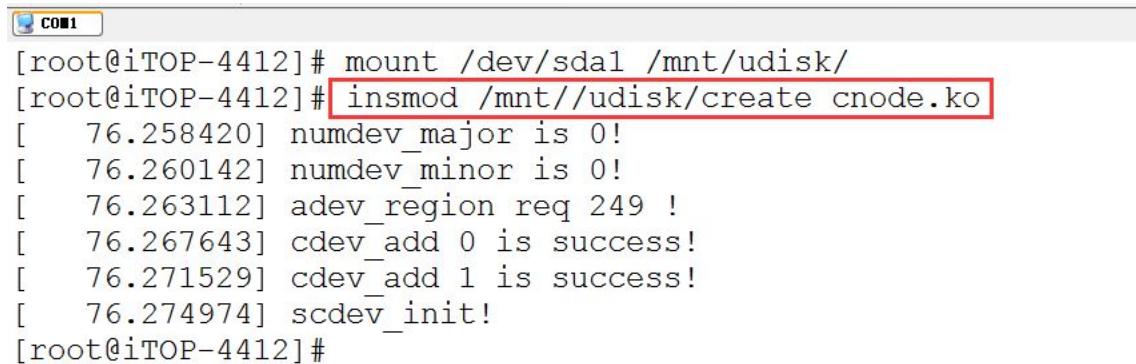
```
root@ubuntu:/home/topeet/create_cnode
root@ubuntu:/home/topeet# mkdir create_cnode
root@ubuntu:/home/topeet# cd create_cnode/
root@ubuntu:/home/topeet/create_cnode# ls
create_cnode.c Makefile
root@ubuntu:/home/topeet/create_cnode# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/create_cnode
modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/create_cnode/create_cnode.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/create_cnode/create_cnode.mod.o
  LD [M] /home/topeet/create_cnode/create_cnode.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/create_cnode# ls
create_cnode.c  create_cnode.mod.c  create_cnode.o  modules.order
create_cnode.ko  create_cnode.mod.o  Makefile       Module.symvers
root@ubuntu:/home/topeet/create_cnode#
```

将生成的驱动模块“create\_cnode.ko”拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令“mount /dev/sda1 /mnt/udisk/”加载 U 盘，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sdal /mnt/udisk/
```

如下图所示，使用命令 “insmod /mnt/udisk/create\_cnode.ko” 加载模块，如下图所示。



```
COM1
[root@iTOP-4412]# mount /dev/sdal /mnt/udisk/
[root@iTOP-4412]# insmod /mnt//udisk/create_cnode.ko
[    76.258420] numdev_major is 0!
[    76.260142] numdev_minor is 0!
[    76.263112] adev_region req 249 !
[    76.267643] cdev_add 0 is success!
[    76.271529] cdev_add 1 is success!
[    76.274974] scdev_init!
[root@iTOP-4412]#
```

使用命令 “ls /sys/class/” 可以查看到生成的 class，如下图所示。

```
[root@iTOP-4412]# ls /sys/class/
android_usb  firmware      koneplus      mmc_host      regulator     spi_master
arvo         graphics       kovaplus      net           rtc          switch
backlight    i2c-adapter   lcd            power_supply  sscsi_device  tty
bdi          i2c-dev       lirc           ppp           sscsi_disk   usb_device
block        ieee80211    mdio_bus      pyra          sscsi_generic video4linux
bluetooth   input         mem            rc            sscsi_host
chardevnode  kone         misc           rc522         sound
[root@iTOP-4412]#
```

使用命令 “ls /dev” 可以查看到生成的两个设备节点，如下图所示。

```
[root@iTOP-4412]# ls /dev
AGPS          kmsg           ram1           srp_ctrl
adc           log            ram10          tty
alarm         loop0          ram11          tty1
android_adb   loop1          ram12          tty2
ashmem         loop2          ram13          tty3
bus           loop3          ram14          tty4
chardevnode0  loop4          ram15          ttyGS0
chardevnode1  loop5          ram2           ttyGS1
console        loop6          ram3           ttyGS2
cpu_dma_latency  loop7          ram4           ttyGS3
exynos-mem    mapper         ram5           ttys0
fb0           max485_ctl_pin ram6           ttys1
fb1           mem            ram7           ttys2
fb2           mmcblk0       ram8           ttys3
fb3           mmcblk0p1     ram9           ttysAC0
fb4           mmcblk0p2     random          ttysAC1
full          mmcblk0p3     rc522          ttysAC2
fuse          mmcblk0p4     relay_ctl      ttysAC3
i2c-0         mtp_usb        root           uinput
i2c-1         network_latency rtc0           urandom
i2c-3         network_throughput  rtc1
i2c-4         null           s3c-mem       usb_accessory
i2c-5         pmem            sda            usbdev1.1
i2c-7         pmem_gpu1     sda1           usbdev1.2
input          ppp             sg0            usbdev1.3
ion           ptmx           shm            usbdev1.4
keychord       pts             snd            watchdog
kmem          ram0           srp            xt_qtaguid
                           ram10          zero
```

这里再介绍一点额外的知识，也是可以通过命令来创建设备节点的，如下图所示，使用命令“mknod dev/test0 c 249 0”和“mknod dev/test1 c 249 1”。

```
[root@iTOP-4412]# mknod dev/test0 c 249 0
[root@iTOP-4412]# mknod dev/test1 c 249 1
[root@iTOP-4412]# ls /dev/
AGPS          log           ram11        tty
adc           loop0         ram12        tty1
alarm         loop1         ram13        tty2
android_adb  loop2         ram14        tty3
ashmem        loop3         ram15        tty4
bus           loop4         ram2         ttyGS0
chardevnode0  loop5         ram3         ttyGS1
chardevnode1  loop6         ram4         ttyGS2
console       loop7         ram5         ttyGS3
cpu_dma_latency mapper       ram6         ttyS0
exynos-mem   max485_ctl_pin ram7         ttyS1
fb0           mem           ram8         ttyS2
fb1           mmcblk0      ram9         ttyS3
fb2           mmcblk0p1    random       ttySAC0
fb3           mmcblk0p2    rc522        ttySAC1
fb4           mmcblk0p3    relay_ctl   ttySAC2
full          mmcblk0p4    root         ttySAC3
fuse          mtp_usb      rtc0         uinput
i2c-0         network_latency   rtc1         urandom
i2c-1         network_throughput s3c-mem    usb_accessory
i2c-3         null          sda          usbdev1.1
i2c-4         pmem          sda1         usbdev1.2
i2c-5         pmem_gpu1   sg0          usbdev1.3
i2c-7         PPP           shm          usbdev1.4
input         ptmx          snd          watchdog
ion           pts           srp          xt_qtaguid
keychord      ram0          srp_ctrl   zero
kmem          ram1          test0      test0
kmsg          ram10         test1      test1
[root@iTOP-4412]#
```

如上图所示，这里给设备号 249 申请了两个设备节点，如果设备号 249 有对应的驱动，用命令创建的设备节点和用代码创建的设备节点有一样的效果，都可以给提供给应用程序调用和操作。

# 实验 21 字符驱动

## 21.1 本章导读

本实验给大家介绍一个完整的字符驱动，字符驱动的框架结构必须要掌握。

### 21.1.1 工具

#### 21.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 21.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端 ( 串口助手 )
- 4 ) 源码文件夹 “char\_driver”

### 21.1.2 预备课程

#### 实验 20 生成字符类设备节点

### 21.1.3 视频资源

本节配套视频为“视频 21\_字符驱动”

### 21.2 学习目标

本章需要学习以下内容：

完善字符设备中 file\_operations 剩余的函数变量

编写简单的应用

### 21.3 实验操作

因为在前面杂项设备中已经介绍了这几个函数，这里不再重复。

将驱动视频教程 20 中的“create\_cnode.c”改为“char\_driver.c”。

如下图所示，添加打开关闭等操作的函数。

```
static int chardevnode_open(struct inode *inode, struct file *file){
    printk(KERN_EMERG "chardevnode_open is success!\n");
    return 0;
}
/*关闭操作*/
static int chardevnode_release(struct inode *inode, struct file *file){
    printk(KERN_EMERG "chardevnode_release is success!\n");
    return 0;
}
/*IO操作*/
static long chardevnode_ioctl(struct file *file, unsigned int cmd, unsigned long arg){
    printk(KERN_EMERG "chardevnode_ioctl is success! cmd is %d,arg is %d \n",cmd,arg);
    return 0;
}

ssize_t chardevnode_read(struct file *file, char __user *buf, size_t count, loff_t *f_ops){
    return 0;
}

ssize_t chardevnode_write(struct file *file, const char __user *buf, size_t count, loff_t *f_ops){
    return 0;
}

loff_t chardevnode_llseek(struct file *file, loff_t offset, int ence){
    return 0;
}
```

将结构体 file\_operations my\_fops 补全，如下图所示。

```
struct file_operations my_fops = {  
    .owner = THIS_MODULE,  
    .open = chardevnode_open,  
    .release = chardevnode_release,  
    .unlocked_ioctl = chardevnode_ioctl,  
    .read = chardevnode_read,  
    .write = chardevnode_write,  
    .llseek = chardevnode_llseek,  
};
```

然后修改一下 Makefile 编译文件，如下图所示。

```
#!/bin/bash  
#通知编译器我们要编译模块的哪些源码  
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o  
obj-m += char_driver.o  
  
#源码目录变量，这里用户需要根据实际情况选择路径  
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的  
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0  
  
#当前目录变量  
PWD ?= $(shell pwd)  
  
#make命名默认寻找第一个目标  
#make -C就是指调用执行的路径  
##$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0  
##$(PWD) 当前目录变量  
#modules要执行的操作  
all:  
    make -C $(KDIR) M=$(PWD) modules  
    |  
    #make clean执行的操作是删除后缀为o的文件  
clean:  
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

如下图所示，写一个简单的应用。

```
#include <stdio.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

main() {
    int fd;
    char *hello_node0 = "/dev/chardevnode0";
    char *hello_node1 = "/dev/chardevnode1";
/*O_RDWR只读打开,O_NDELAY非阻塞方式*/
    if((fd = open(hello_node0,O_RDWR|O_NDELAY))<0){
        printf("APP open %s failed!\n",hello_node0);
    }
    else{
        printf("APP open %s success!\n",hello_node0);
    }

    close(fd);

    if((fd = open(hello_node1,O_RDWR|O_NDELAY))<0){
        printf("APP open %s failed!\n",hello_node1);
    }
    else{
        printf("APP open %s success!\n",hello_node1);
    }

    close(fd);
}
```

修改完成之后，在 Ubuntu 系统下使用命令 “mkdir char\_driver” 新建文件夹 “char\_driver” ，然后将修改好的驱动文件 “char\_driver.c” 、 Makefile 文件拷贝到文件夹 “char\_driver” 中，如下图所示。

```
root@ubuntu:/home/topeet/char_driver
root@ubuntu:/home/topeet# mkdir char_driver
root@ubuntu:/home/topeet# cd char_driver/
root@ubuntu:/home/topeet/char_driver# ls
char_driver.c  Makefile
root@ubuntu:/home/topeet/char_driver#
```

使用编译命令“make”编译驱动，如下图所示。

```
root@ubuntu:/home/topeet/char_driver
root@ubuntu:/home/topeet# mkdir char_driver
root@ubuntu:/home/topeet# cd char_driver/
root@ubuntu:/home/topeet/char_driver# ls
char_driver.c  Makefile
root@ubuntu:/home/topeet/char_driver# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/char_driver modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M]  /home/topeet/char_driver/char_driver.o
/home/topeet/char_driver/char_driver.c: In function 'chardevnode_ioctl':
/home/topeet/char_driver/char_driver.c:64: warning: format '%d' expects type 'int',
      but argument 3 has type 'long unsigned int'
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/topeet/char_driver/char_driver.mod.o
  LD [M]  /home/topeet/char_driver/char_driver.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/char_driver# ls
char_driver.c  char_driver.mod.c  char_driver.o  modules.order
char_driver.ko  char_driver.mod.o  Makefile        Module.symvers
root@ubuntu:/home/topeet/char_driver#
```

将修改好的应用文件“invoke\_char\_driver.c”拷贝到文件夹“char\_driver”中，使用命令

"arm-none-linux-gnueabi-gcc -o invoke\_char\_driver invoke\_char\_driver.c -static"

编译应用，如下图所示。

```
root@ubuntu:/home/topeet/char_driver
char_driver.c  Makefile
root@ubuntu:/home/topeet/char_driver# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/char_driver modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/char_driver/char_driver.o
/home/topeet/char_driver/char_driver.c: In function 'chardevnode_ioctl':
/home/topeet/char_driver/char_driver.c:64: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
Building modules, stage 2.
MODPOST 1 modules
  CC  /home/topeet/char_driver/char_driver.mod.o
  LD [M] /home/topeet/char_driver/char_driver.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/char_driver# ls
char_driver.c  char_driver.mod.c  char_driver.o  modules.order
char_driver.ko  char_driver.mod.o  Makefile      Module.symvers
root@ubuntu:/home/topeet/char_driver# arm-none-linux-gnueabi-gcc -o invoke_char_driver invoke_char_driver.c -static
root@ubuntu:/home/topeet/char_driver# ls
char_driver.c  char_driver.mod.o  invoke_char_driver.c  Module.symvers
char_driver.ko  char_driver.o  Makefile
char_driver.mod.c  invoke_char_driver  modules.order
root@ubuntu:/home/topeet/char_driver#
```

将生成的驱动模块 "char\_driver.ko" 以及应用 "invoke\_char\_driver" 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 "mount /dev/sda1 /mnt/udisk/" 加载 U 盘，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
```

使用命令 "insmod /mnt/udisk/char\_driver.ko" 加载驱动模块。

 COM1  
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/  
[root@iTOP-4412]# insmod /mnt/udisk/char\_driver.ko  
[ 220.579074] numdev\_major is 0!  
[ 220.580681] numdev\_minor is 0!  
[ 220.583686] adev\_region req 249 !  
[ 220.588512] cdev\_add 0 is success!  
[ 220.591897] cdev\_add 1 is success!  
[ 220.595303] scdev\_init!  
[root@iTOP-4412]#

使用命令 “./mnt/udisk/char\_driver” 运行应用，调用驱动模块生成的设备节点，如下图所示，可以看到两个设备节点都可以正常打开，说明驱动底层的 open 函数可以正常使用。

```
[root@iTOP-4412]# ./mnt/udisk/char_driver  
[ 324.627849] chardevnode_open is success!  
APP open /dev/cha[ 324.635780] chardevnode_release is success!  
[ 324.639936] chardevnode_open is success!  
[ 324.643929] chardevnode_release is success!  
rdevnode0 success!  
APP open /dev/chardevnode1 success!  
[root@iTOP-4412]#
```

# 实验 22 字符类 GPIOS

## 22.1 本章导读

本期实验是对前面知识的一个小结，将字符类驱动的框架和 GPIO 操作函数的结合，就可以完成一个字符类 GPIO 驱动。

### 22.1.1 工具

#### 22.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 22.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “char\_driver\_leds”

### 22.1.2 预备课程

#### 实验 21 字符驱动

### 22.1.3 视频资源

本节配套视频为“视频 22\_字符类 GPIO”

### 22.2 学习目标

本章需要学习以下内容：

将前面杂项设备中对 GPIO 操作的函数引入到字符设备中

### 22.3 实验操作

在视频教程中只做了简单的演示，因为所有的知识点在前面都已经介绍过了。

将“21\_字符驱动”中的文件“char\_driver.c”改为“char\_driver\_leds.c”，因为 char\_driver\_leds.c 文件中的代码有点长了，所以将宏定义和结构体定义放到头文件“char\_driver\_leds.h”中，如下图所示。

```
#ifndef _CHAR_DRIVER_LEDS_H_
#define _CHAR_DRIVER_LEDS_H_

#ifndef DEVICE_NAME
#define DEVICE_NAME "chardevnode"
#endif

#ifndef DEVICE_MINOR_NUM
#define DEVICE_MINOR_NUM 2
#endif

#ifndef DEV_MAJOR
#define DEV_MAJOR 0
#endif

#ifndef DEV_MINOR
#define DEV_MINOR 0
#endif

#ifndef REGDEV_SIZE
#define REGDEV_SIZE 3000
#endif

struct reg_dev
{
    char *data;
    unsigned long size;

    struct cdev cdev;
};

#endif
```

然后添加 GPIO 操作的头文件，以及自定义的头文件，如下图所示。

```
/*自定义头文件*/
#include "char_driver_leds.h"

/*Linux中申请GPIO的头文件*/
#include <linux/gpio.h>
/*三星平台的GPIO配置函数头文件*/
/*三星平台EXYNOS系列平台，GPIO配置参数宏定义头文件*/
#include <plat/gpio-cfg.h>
/*三星平台4412平台，GPIO宏定义头文件*/
#include <mach/gpio-exynos4.h>

MODULE_LICENSE("Dual BSD/GPL");
/*声明是开源的，没有内核版本限制*/
MODULE_AUTHOR("iTOPEET_dz");
/*声明作者*/
```

将两个 gpio 定义为数组，如下图所示。

```
static int led_gpios[] = {
    EXYNOS4_GPL2(0), EXYNOS4_GPK1(1),
};

#define LED_NUM      ARRAY_SIZE(led_gpios)
```

如下图所示，自定义一个 GPIO 初始化函数 gpio\_init。

```
static int gpio_init(void){
    int i=0,ret;

    for(i=0;i<LED_NUM;i++){
        ret = gpio_request(led_gpios[i], "LED");
        if (ret) {
            printk("%s: request GPIO %d for LED failed, ret = %d\n", DEVICE_NAME,i,ret);
            return -1;
        }
        else{
            s3c_gpio_cfgpin(led_gpios[i], S3C_GPIO_OUTPUT);
            gpio_set_value(led_gpios[i], 1);
        }
    }

    return 0;
}
```

在驱动入口的函数中，创建设备节点成功之后再调用 GPIO 初始化函数，如下图所示。

```
    /*创建设备节点*/
    device_create(myclass,NULL,MKDEV(numdev_major,numdev_minor+i),NULL,DEVICE_NAME"%d",i);
}

ret = gpio_init();
if(ret){
    printk(KERN_EMERG "gpio_init failed!\n");
}

printk(KERN_EMERG "scdev_init!\n");
/*打印信息，KERN_EMERG表示紧急信息*/
return 0;
```

然后在驱动的出口函数中添加释放 GPIO 的代码，如下图所示。

```
static void scdev_exit(void)
{
    int i;
    printk(KERN_EMERG "scdev_exit!\n");

    /*除去字符设备*/
    for(i=0;i<DEVICE_MINOR_NUM;i++){
        cdev_del(&(my_devices[i].cdev));
        /*摧毁设备节点函数d*/
        device_destroy(myclass,MKDEV(numdev_major,numdev_minor+i));
    }
    /*释放设备class*/
    class_destroy(myclass);
    /*释放内存*/
    kfree(my_devices);

    /*释放GPIO*/
    for(i=0;i<LED_NUM;i++){
        gpio_free(led_gpios[i]);
    }

    unregister_chrdev_region(MKDEV(numdev_major,numdev_minor),DEVICE_MINOR_NUM);
}
```

然后在 ioctl 中添加 io 操作的代码，如下图所示。

```

/*IO操作*/
static long chardevnode_ioctl(struct file *file, unsigned int cmd, unsigned long arg){
    switch(cmd)
    {
        case 0:
        case 1:
            if (arg > LED_NUM) {
                return -EINVAL;
            }

            gpio_set_value(led_gpios[arg], cmd);
            break;

        default:
            return -EINVAL;
    }

    printk(KERN_EMERG "chardevnode_ioctl is success! cmd is %d,arg is %d \n",cmd,arg);

    return 0;
}

```

如下图所示，修改一下 Makefile 文件。

```

#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译iTop4412_hello.c这个文件编译成中间文件mini_linux_module.o
obj-m += char_driver_leds.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#${(KDIR)}Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#${(PWD)}当前目录变量
#modules要执行的操作
all:
    make -C ${KDIR} M=${PWD} modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers

```

编写一个简单的应用 “invoke\_char\_gpios.c” ，如下图所示。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

/*argv[1] is cmd , argv[2] is io_arg*/
int main(int argc , char **argv){
    int fd;
    char *lednode = "/dev/chardevnode0";

    /*O_RDWR只读打开,O_NDELAY非阻塞方式*/
    if((fd = open(lednode,O_RDWR|O_NDELAY))<0) {
        printf("APP open %s failed!\n",lednode);
    }
    else{
        printf("APP open %s success!\n",lednode);
        ioctl(fd,atoi(argv[1]),atoi(argv[2]));
        printf("APP ioctl %s ,cmd is %s! io_arg is %s!\n",lednode,argv[1],argv[2]);
    }

    close(fd);
}
```

修改完成之后，在 Ubuntu 系统下使用命令 “mkdir char\_driver\_leds” 新建文件夹

“char\_driver\_leds” ，然后将修改好的驱动文件 “char\_driver\_leds.c” 、头文件

“char\_driver\_leds.h” 、Makefile 文件以及应用文件 “invoke\_char\_gpios.c” 拷贝到文件夹

“char\_driver\_leds” 中，如下图所示。

```
root@ubuntu:/home/topeet# mkdir char_driver_leds
root@ubuntu:/home/topeet# cd char_driver_leds/
root@ubuntu:/home/topeet/char_driver_leds# ls
char_driver_leds.c  char_driver_leds.h  invoke_char_gpios.c  Makefile
root@ubuntu:/home/topeet/char_driver_Leds#
```

使用编译命令“make” 编译驱动，如下图所示。

```
root@ubuntu:/home/topeet# mkdir char_driver_leds
root@ubuntu:/home/topeet# cd char_driver_leds/
root@ubuntu:/home/topeet/char_driver_leds# ls
char_driver_leds.c  char_driver_leds.h  invoke_char_gpios.c  Makefile
root@ubuntu:/home/topeet/char_driver_leds# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/char_driver_leds modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/char_driver_leds/char_driver_leds.o
/home/topeet/char_driver_leds/char_driver_leds.c: In function 'chardevnode_ioctl':
:
/home/topeet/char_driver_leds/char_driver_leds.c:84: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/char_driver_leds/char_driver_leds.mod.o
  LD [M] /home/topeet/char_driver_leds/char_driver_leds.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/char_driver_leds# ls
char_driver_leds.c      char_driver_leds.mod.o  modules.order
char_driver_leds.h      char_driver_leds.o     Module.symvers
char_driver_leds.ko      invoke_char_gpios.c
char_driver_leds.mod.c  Makefile
root@ubuntu:/home/topeet/char_driver_leds#
```

使用命令

“arm-none-linux-gnueabi-gcc -o invoke\_char\_gpios invoke\_char\_gpios.c -static”

编译应用程序“invoke\_char\_gpios”，如下图所示。

```
root@ubuntu:/home/topeet/char_driver_leds
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/char_driver_leds/char_driver_leds.o
/home/topeet/char_driver_leds/char_driver_leds.c: In function 'chardevnode_ioctl':
:
/home/topeet/char_driver_leds/char_driver_leds.c:84: warning: format '%d' expects type 'int', but argument 3 has type 'long unsigned int'
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/topeet/char_driver_leds/char_driver_leds.mod.o
  LD [M] /home/topeet/char_driver_leds/char_driver_leds.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/char_driver_leds# ls
char_driver_leds.c      char_driver_leds.mod.o  modules.order
char_driver_leds.h       char_driver_leds.o     Module.symvers
char_driver_leds.ko     invoke_char_gpios.c
char_driver_leds.mod.c  Makefile
root@ubuntu:/home/topeet/char_driver_leds# arm-none-linux-gnueabi-gcc -o invoke_
char_gpios invoke_char_gpios.c -static
root@ubuntu:/home/topeet/char_driver_leds# ls
char_driver_leds.c      char_driver_leds.mod.o  Makefile
char_driver_leds.h       char_driver_leds.o     modules.order
char_driver_leds.ko     invoke_char_gpios        Module.symvers
char_driver_leds.mod.c  invoke_char_gpios.c
root@ubuntu:/home/topeet/char_driver_leds#
```

将生成的驱动模块 “char\_driver\_leds.ko” 以及应用 “invoke\_char\_gpios” 拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令 “mount /dev/sda1 /mnt/udisk/” 加载 U 盘，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
```

使用命令 “insmod /mnt/udisk/char\_driver\_leds.ko” 加载驱动模块，如下图所示。

```
COM1
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sdal /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/char driver leds.ko
[ 23.528827] numdev_major is 0!
[ 23.530533] numdev_minor is 0!
[ 23.533485] adev_region req 249 !
[ 23.545821] cdev_add 0 is success!
[ 23.555677] cdev_add 1 is success!
[ 23.565667] scdev_init!
[root@iTOP-4412]#
```

使用命令 “./mnt/udisk/invite\_char\_gpios 0 1” , 运行应用。参数 1 为命令 , 参数 2 为 GPIO 编号 , 如下图所示。

```
COM1
[root@iTOP-4412]# ./mnt/udisk/invite_char_gpios 0 1
[ 156.096445] chardevnode_open is success!
APP open /dev/cha[ 156.104623] chardevnode_ioctl is success! cmd is 0,arg is 1
rdevnode0 success!
APP ioctl /dev/ch[ 156.120194] chardevnode_release is success!
ardevnode0 ,cmd is 0! io_arg is 1!
[root@iTOP-4412]#
```

运行如上图所示命令之后 , 可以看到小灯会灭一个。

另外如果大家想将两个设备节点定义为不同的操作 , 那么就需要额外的定义 “file\_operations” 。

然后 , 在如下图所示的代码处 , 分别对设备定义不同的 “file\_operations” 即可。

```
static void reg_init_cdev(struct reg_dev *dev,int index){  
    int err;  
    int devno = MKDEV(numdev_major,numdev_minor+index);  
  
    /*数据初始化*/  
    cdev_init(&dev->cdev,&my_fops);  
    dev->cdev.owner = THIS_MODULE;  
    dev->cdev.ops = &my_fops;  
  
    /*注册到系统*/  
    err = cdev_add(&dev->cdev,devno,1);  
    if(err){  
        printk(KERN_EMERG "cdev_add %d is fail! %d\n",index,err);  
    }  
    else{  
        printk(KERN_EMERG "cdev_add %d is success!\n",numdev_minor+index);  
    }  
}
```

# 实验 23 proc 文件系统

## 23.1 本章导读

所有的 Linux 文件系统都自带 proc 文件系统，类似于 window 系统的任务管理器，在调试驱动的时候会用到。

学习方法类似前面的“linux 命令”，了解到有这个功能，学习几个基本的命令，然后在后面用到的时候去网上查找其对应用法即可。

proc 文件系统一般是只读，也是可以通过编写代码给 proc 中添加信息，感兴趣可以在网上找一找代码来实现，用处不是很大，系统自带的功能已经够用了。

### 23.1.1 工具

#### 23.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) PC 机
- 3 ) 串口

#### 23.1.1.2 软件工具

- 1 ) 虚拟机 Vmware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）

### 23.1.2 预备课程

无

### 23.1.3 视频资源

本节配套视频为“视频 23\_proc 文件系统”

## 23.2 学习目标

本章需要学习以下内容：

了解 proc 文件系统

## 23.3 实验操作

在开发板中输入命令“cat proc/meminfo”，如下图所示。

```
[root@iTOP-4412]# cat proc/meminfo
MemTotal:           953748 kB
MemFree:            939020 kB
Buffers:             476 kB
Cached:              2744 kB
SwapCached:          0 kB
Active:              1560 kB
Inactive:            1988 kB
Active(anon):        332 kB
Inactive(anon):      0 kB
Active(file):        1228 kB
Inactive(file):      1988 kB
Unevictable:          0 kB
Mlocked:              0 kB
HighTotal:            261120 kB
HighFree:             257480 kB
LowTotal:             692628 kB
```

常用参数如下：

MemTotal:总内存

MemFree : 空闲内存

Cached : 缓存

Active : 活跃内存

Inactive : 非活跃内存

在开发板中输入命令 “cat /proc/cpuinfo” , 如下图所示。

```
[root@iTOP-4412]# cat /proc/cpuinfo
cpu id          : 0xe4412011
Processor       : ARMv7 Processor rev 0 (v7l)
processor       : 0
BogoMIPS        : 1992.29

Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant    : 0x3
CPU part       : 0xc09
CPU revision   : 0

Hardware        : SMDK4X12
Revision        : 0000
Serial          : 0000000000000000
```

常用参数如下：

cpu id : cpu 代号

Processor : 处理器

在开发板中输入命令 “cat /proc/interrupts” ，如下图所示。

```
[root@iTOP-4412]# cat /proc/interrupts
          CPU0
 24:      112  s3c-uart    s5pv210-uart
 26:      640  s3c-uart    s5pv210-uart
 98:          0      GIC    s3c-pl330.0
 99:          0      GIC    s3c-pl330.1
100:         0      GIC    s3c-pl330.2
107:         0      GIC    s3c2410-wdt
108:         0      GIC    s3c2410-rtc alarm
121:          9      GIC  mct_comp_irq
123:     23616      GIC    s3c2440-i2c.1
125:          1      GIC    s3c2440-i2c.3
126:          60     GIC    s3c2440-i2c.4
127:          0      GIC    s3c2440-i2c.5
129:          6      GIC    s3c2440-i2c.7
134:     17526      GIC  ehci_hcd:usb1
135:          41     GIC    s3c-udc
139:          0      GIC    mmc1
140:          52     GIC    mmc2
141:        1477      GIC    mmc0
160:          0      GIC    samsung-rp
281:          0  COMBINER    s3cfb
352:          1  exynos-eint
359:          0  exynos-eint    s3c-sdhci.2
367:          1  exynos-eint    s5m87xx-irq
370:          0  exynos-eint    switch-gpio
428:          0    s5m8767  rtc-alarm0
```

如上图所示，显示的是一些中断相关的参数，如果新注册了中断，都会在这里显示。

## 23.4 proc 参数介绍

下面是网上介绍的一个相对详细的帖子，我给大家做了简单的编辑，大家可以浏览一下，对其有个整体上的把握即可。

Linux 系统上的/proc 目录是一种文件系统，即 proc 文件系统。与其它常见的文件系统不同的是，/proc 是一种伪文件系统（也即虚拟文件系统），存储的是当前内核运行状态的一系列特殊文件，用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息，甚至可以通过更改其中某些文件来改变内核的运行状态。

基于/proc 文件系统如上所述的特殊性，其内的文件也常被称作虚拟文件，并具有一些独特的特点。例如，其中有些文件虽然使用查看命令查看时会返回大量信息，但文件本身的大小却会显示为 0 字节。此外，这些特殊文件中大多数文件的时间及日期属性通常为当前系统时间和日期，这跟它们随时会被刷新（存储于 RAM 中）有关。

为了查看及使用上的方便，这些文件通常会按照相关性进行分类存储于不同的目录甚至子目录中，如/proc/scsi 目录中存储的就是当前系统上所有 SCSI 设备的相关信息，/proc/N 中存储的则是系统当前正在运行的进程的相关信息，其中 N 为正在运行的进程（可以想象得到，在某进程结束后其相关目录则会消失）。

大多数虚拟文件可以使用文件查看命令如 cat、more 或者 less 进行查看，有些文件信息表述的内容可以一目了然，但也有文件的信息却不怎么具有可读性。不过，这些可读性较差的文件在使用一些命令如 apm、free、lspci 或 top 查看时却可以有着不错的表现。

## 一、 进程目录中的常见文件介绍

/proc 目录中包含许多以数字命名的子目录，这些数字表示系统当前正在运行进程的进程号，里面包含对应进程相关的多个信息文件。

```
[root@rhel5 ~]# ll /proc
```

```
total 0
```

dr-xr-xr-x	5	root	root	0 Feb 8 17:08	1
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	10
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	11
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	1156
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	139
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	140
dr-xr-xr-x	5	root	root	0 Feb 8 17:08	141
dr-xr-xr-x	5	root	root	0 Feb 8 17:09	1417
dr-xr-xr-x	5	root	root	0 Feb 8 17:09	1418

上面列出的是/proc 目录中一些进程相关的目录，每个目录中是当程本身相关信息的文件。

下面是作者系统 ( RHEL5.3 ) 上运行的一个 PID 为 2674 的进程 saslauthd 的相关文件，其中有些文件是每个进程都会具有的，后文会对这些常见文件做出说明。

```
[root@rhel5 ~]# ll /proc/2674
total 0
dr-xr-xr-x 2 root root 0 Feb  8 17:15 attr
-r----- 1 root root 0 Feb  8 17:14 auxv
-r--r--r-- 1 root root 0 Feb  8 17:09 cmdline
-rw-r--r-- 1 root root 0 Feb  8 17:14 coredump_filter
-r--r--r-- 1 root root 0 Feb  8 17:14 cpuset
lrwxrwxrwx 1 root root 0 Feb  8 17:14 cwd -> /var/run/saslauthd
-r----- 1 root root 0 Feb  8 17:14 environ
lrwxrwxrwx 1 root root 0 Feb  8 17:09 exe -> /usr/sbin/saslauthd
dr-x----- 2 root root 0 Feb  8 17:15 fd
-r----- 1 root root 0 Feb  8 17:14 limits
-rw-r--r-- 1 root root 0 Feb  8 17:14 loginuid
-r--r--r-- 1 root root 0 Feb  8 17:14 maps
-rw----- 1 root root 0 Feb  8 17:14 mem
-r--r--r-- 1 root root 0 Feb  8 17:14 mounts
```

```
-r----- 1 root root 0 Feb  8 17:14 mountstats  
-rw-r--r-- 1 root root 0 Feb  8 17:14 oom_adj  
-r--r--r-- 1 root root 0 Feb  8 17:14 oom_score  
lrwxrwxrwx 1 root root 0 Feb  8 17:14 root -> /  
-r--r--r-- 1 root root 0 Feb  8 17:14 schedstat  
-r----- 1 root root 0 Feb  8 17:14 smaps  
-r--r--r-- 1 root root 0 Feb  8 17:09 stat  
-r--r--r-- 1 root root 0 Feb  8 17:14 statm  
-r--r--r-- 1 root root 0 Feb  8 17:10 status  
dr-xr-xr-x 3 root root 0 Feb  8 17:15 task  
-r--r--r-- 1 root root 0 Feb  8 17:14 wchan
```

1.1、cmdline — 启动当前进程的完整命令 ,但僵尸进程目录中的此文件不包含任何信息 ;

```
[root@rhel5 ~]# more /proc/2674/cmdline  
/usr/sbin/saslauthd
```

1.2、 cwd — 指向当前进程运行目录的一个符号链接 ;

1.3、environ — 当前进程的环境变量列表，彼此间用空字符（NULL）隔开；变量用大写字母表示，其值用小写字母表示；

```
[root@rhel5 ~]# more /proc/2674/environ
```

```
TERM=linuxauthd
```

1.4、exe — 指向启动当前进程的可执行文件（完整路径）的符号链接，通过/proc/N/exe可以启动当前进程的一个拷贝；

1.5、fd — 这是个目录，包含当前进程打开的每一个文件的文件描述符（file descriptor），这些文件描述符是指向实际文件的一个符号链接；

```
[root@rhel5 ~]# ll /proc/2674/fd
```

```
total 0
```

```
lrwx----- 1 root root 64 Feb 8 17:17 0 -> /dev/null
```

```
lrwx----- 1 root root 64 Feb 8 17:17 1 -> /dev/null
```

```
lrwx----- 1 root root 64 Feb 8 17:17 2 -> /dev/null
```

```
lrwx----- 1 root root 64 Feb 8 17:17 3 -> socket:[7990]
```

```
lrwx----- 1 root root 64 Feb 8 17:17 4 -> /var/run/saslauthd/saslauthd.pid
```

```
lrwx----- 1 root root 64 Feb 8 17:17 5 -> socket:[7991]
```

```
lrwx----- 1 root root 64 Feb 8 17:17 6 -> /var/run/saslauthd/mux.accept
```

1.6、limits — 当前进程所使用的每一个受限资源的软限制、硬限制和管理单元；此文件仅可由实际启动当前进程的 UID 用户读取；（2.6.24 以后的内核版本支持此功能）；

1.7、maps — 当前进程关联到的每个可执行文件和库文件在内存中的映射区域及其访问权限所组成的列表；

```
[root@rhel5 ~]# cat /proc/2674/maps
00110000-00239000 r-xp 00000000 08:02 130647      /lib/libcrypto.so.0.9.8e
00239000-0024c000 rwxp 00129000 08:02 130647      /lib/libcrypto.so.0.9.8e
0024c000-00250000 rwxp 0024c000 00:00 0
00250000-00252000 r-xp 00000000 08:02 130462      /lib/libdl-2.5.so
00252000-00253000 r-xp 00001000 08:02 130462      /lib/libdl-2.5.so
```

1.8、mem — 当前进程所占用的内存空间，由 open、read 和 lseek 等系统调用使用，不能被用户读取；

1.9、root — 指向当前进程运行根目录的符号链接；在 Unix 和 Linux 系统上，通常采用 chroot 命令使每个进程运行于独立的根目录；

1.10、stat — 当前进程的状态信息，包含一系统格式化后的数据列，可读性差，通常由 ps 命令使用；

1.11、statm — 当前进程占用内存的状态信息，通常以“页面”（page）表示；

1.12、status — 与 stat 所提供信息类似，但可读性较好，如下所示，每行表示一个属性信息；其详细介绍请参见 proc 的 man 手册页；

```
[root@rhel5 ~]# more /proc/2674/status
```

Name: saslauthd

State: S (sleeping)

SleepAVG: 0%

Tgid: 2674

Pid: 2674

PPid: 1

TracerPid: 0

Uid: 0 0 0 0

Gid: 0 0 0 0

FDSSize: 32

Groups:

VmPeak: 5576 kB

VmSize: 5572 kB

VmLck: 0 kB

VmHWM: 696 kB

VmRSS: 696 kB

.....

1.13、task — 目录文件，包含由当前进程所运行的每一个线程的相关信息，每个线程的相关信息文件均保存在一个由线程号（tid）命名的目录中，这类似于其内容类似于每个进程目录中的内容；（内核2.6版本以后支持此功能）

## 二、/proc目录下常见的文件介绍

### 2.1、/proc/apm

高级电源管理（APM）版本信息及电池相关状态信息，通常由apm命令使用；

### 2.2、/proc/buddyinfo

用于诊断内存碎片问题的相关信息文件；

### 2.3、/proc/cmdline

在启动时传递至内核的相关参数信息，这些信息通常由lilo或grub等启动管理工具进行传递；

```
[root@rhel5 ~]# more /proc/cmdline
ro root=/dev/VolGroup00/LogVol00 rhgb quiet
```

## 2.4、/proc/cpuinfo

处理器的相关信息的文件；

## 2.5、/proc/crypto

系统上已安装的内核使用的密码算法及每个算法的详细信息列表；

```
[root@rhel5 ~]# more /proc/crypto
```

name : crc32c

driver : crc32c-generic

module : kernel

priority : 0

type : digest

blocksize : 32

digestsize : 4

.....

## 2.6、/proc/devices

系统已经加载的所有块设备和字符设备的信息，包含主设备号和设备组（与主设备号对应  
的设备类型）名；

```
[root@rhel5 ~]# more /proc/devices
```

### Character devices:

1 mem

4 /dev/vc/0

4 tty

4 ttyS

.....

### Block devices:

1 ramdisk

2 fd

8 sd

.....

## 2.7、/proc/diskstats

每块磁盘设备的磁盘 I/O 统计信息列表；（内核 2.5.69 以后的版本支持此功能）

## 2.8、/proc/dma

每个正在使用且注册的 ISA DMA 通道的信息列表；

```
[root@rhel5 ~]# more /proc/dma
```

2: floppy

4: cascade

## 2.9、/proc/execdomains

内核当前支持的执行域（每种操作系统独特“个性”）信息列表；

```
[root@rhel5 ~]# more /proc/execdomains
```

0-0	Linux	[kernel]
-----	-------	----------

## 2.10、/proc/fb

帧缓冲设备列表文件，包含帧缓冲设备的设备号和相关驱动信息；

## 2.11、/proc/filesystems

当前被内核支持的文件系统类型列表文件 , 被标示为 nodev 的文件系统表示不需要块设备的支持 ; 通常 mount 一个设备时 , 如果没有指定文件系统类型将通过此文件来决定其所需文件系统的类型 ;

```
[root@rhel5 ~]# more /proc/filesystems
```

```
nodev    sysfs
```

```
nodev    rootfs
```

```
nodev    proc
```

```
iso9660
```

```
ext3
```

```
.....
```

```
.....
```

## 2.12、/proc/interrupts

X86 或 X86\_64 体系架构系统上每个 IRQ 相关的中断号列表 ; 多路处理器平台上每个 CPU 对于每个 I/O 设备均有自己的中断号 ;

```
[root@rhel5 ~]# more /proc/interrupts
```

```
CPU0
```

```
0:    1305421    IO-APIC-edge  timer
```

```
1:        61    IO-APIC-edge  i8042
```

185: 1068 IO-APIC-level eth0

.....

## 2.13、/proc/iomem

每个物理设备上的记忆体 ( RAM 或者 ROM ) 在系统内存中的映射信息 ;

[root@rhel5 ~]# more /proc/iomem

00000000-0009f7ff : System RAM

0009f800-0009ffff : reserved

000a0000-000bffff : Video RAM area

000c0000-000c7fff : Video ROM

.....

## 2.14、/proc/ioports

当前正在使用且已经注册过的与物理设备进行通讯的输入-输出端口范围信息列表 ; 如下面所示 , 第一列表示注册的 I/O 端口范围 , 其后表示相关的设备 ;

[root@rhel5 ~]# less /proc/ioports

0000-001f : dma1

0020-0021 : pic1

0040-0043 : timer0

0050-0053 : timer1

0060-006f : keyboard

.....

## 2.15、/proc/kallsyms

模块管理工具用来动态链接或绑定可装载模块的符号定义，由内核输出；（内核 2.5.71 以后的版本支持此功能）；通常这个文件中的信息量相当大；

```
[root@rhel5 ~]# more /proc/kallsyms
```

c04011f0 T \_stext

c04011f0 t run\_init\_process

c04011f0 T stext

.....

## 2.16、/proc/kcore

系统使用的物理内存，以 ELF 核心文件（core file）格式存储，其文件大小为已使用的物理内存（RAM）加上 4KB；这个文件用来检查内核数据结构的当前状态，因此，通常由 GDB 通常调试工具使用，但不能使用文件查看命令打开此文件；

## 2.17、/proc/kmsg

此文件用来保存由内核输出的信息，通常由/sbin/klogd 或/bin/dmsg 等程序使用，不要试图使用查看命令打开此文件；

## 2.18、/proc/loadavg

保存关于 CPU 和磁盘 I/O 的负载平均值，其前三列分别表示每 1 秒钟、每 5 秒钟及每 15 秒的负载平均值，类似于 uptime 命令输出的相关信息；第四列是由斜线隔开的两个数值，前者表示当前正由内核调度的实体（进程和线程）的数目，后者表示系统当前存活的内核调度实体的数目；第五列表示此文件被查看前最近一个由内核创建的进程的 PID；

```
[root@rhel5 ~]# more /proc/loadavg
```

```
0.45 0.12 0.04 4/125 5549
```

```
[root@rhel5 ~]# uptime
```

```
06:00:54 up 1:06, 3 users, load average: 0.45, 0.12, 0.04
```

## 2.19、/proc/locks

保存当前由内核锁定的文件的相关信息，包含内核内部的调试数据；每个锁定占据一行，且具有一个唯一的编号；如下输出信息中每行的第二列表示当前锁定使用的锁定类别，POSIX

表示目前较新类型的文件锁，由 lockf 系统调用产生，FLOCK 是传统的 UNIX 文件锁，由 flock 系统调用产生；第三列也通常由两种类型，ADVISORY 表示不允许其他用户锁定此文件，但允许读取，MANDATORY 表示此文件锁定期间不允许其他用户任何形式的访问；

```
[root@rhel5 ~]# more /proc/locks
```

```
1: POSIX ADVISORY WRITE 4904 fd:00:4325393 0 EOF
```

```
2: POSIX ADVISORY WRITE 4550 fd:00:2066539 0 EOF
```

```
3: FLOCK ADVISORY WRITE 4497 fd:00:2066533 0 EOF
```

## 2.20、/proc/mdstat

保存 RAID 相关的多块磁盘的当前状态信息，在没有使用 RAID 机器上，其显示为如下状态：

```
[root@rhel5 ~]# less /proc/mdstat
```

```
Personalities :
```

```
unused devices: <none>
```

## 2.21、/proc/meminfo

系统中关于当前内存的利用状况等的信息，常由 free 命令使用；可以使用文件查看命令直接读取此文件，其内容显示为两列，前者为统计属性，后者为对应的值；

```
[root@rhel5 ~]# less /proc/meminfo
```

MemTotal: 515492 kB

MemFree: 8452 kB

Buffers: 19724 kB

Cached: 376400 kB

SwapCached: 4 kB

.....

## 2.22、/proc/mounts

在内核 2.4.29 版本以前 ,此文件的内容为系统当前挂载的所有文件系统 ,在 2.4.19 以后的内核中引进了每个进程使用独立挂载名称空间的方式 ,此文件则随之变成了指向 /proc/self/mounts ( 每个进程自身挂载名称空间中的所有挂载点列表 ) 文件的符号链接 ; /proc/self 是一个独特的目录 ,后文中会对此目录进行介绍 ;

```
[root@rhel5 ~]# ll /proc |grep mounts
```

lrwxrwxrwx 1 root root 11 Feb 8 06:43 mounts ->

self/mounts

如下所示，其中第一列表示挂载的设备，第二列表示在当前目录树中的挂载点，第三列表示当前文件系统的类型，第四列表示挂载属性（ro 或者 rw），第五列和第六列用来匹配 /etc/mtab 文件中的转储（dump）属性；

```
[root@rhel5 ~]# more /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / ext3 rw,data=ordered 0 0
/dev /dev tmpfs rw 0 0
/proc /proc proc rw 0 0
/sys /sys sysfs rw 0 0
/proc/bus/usb /proc/bus/usb usbfs rw 0 0
....
```

## 2.23、/proc/modules

当前装入内核的所有模块名称列表，可以由 lsmod 命令使用，也可以直接查看；如下所示，其中第一列表示模块名，第二列表示此模块占用内存空间大小，第三列表示此模块有多少实例被装入，第四列表示此模块依赖于其它哪些模块，第五列表示此模块的装载状态（Live：已经装入；Loading：正在装入；Unloading：正在卸载），第六列表示此模块在内核内存（kernel memory）中的偏移量；

```
[root@rhel5 ~]# more /proc/modules
autofs4 24517 2 - Live 0xe09f7000
hidp 23105 2 - Live 0xe0a06000
rfcomm 42457 0 - Live 0xe0ab3000
l2cap 29505 10 hidp,rfcomm, Live 0xe0aaa000
.....
```

## 2.24、/proc/partitions

块设备每个分区的主设备号 ( major ) 和次设备号 ( minor ) 等信息 , 同时包括每个分区所包含的块 ( block ) 数目 ( 如下面输出中第三列所示 ) ;

```
[root@rhel5 ~]# more /proc/partitions
```

```
major minor #blocks name
```

8	0	20971520	sda
8	1	104391	sda1
8	2	6907950	sda2
8	3	5630782	sda3
8	4	1	sda4
8	5	3582463	sda5

## 2.25、/proc/pci

内核初始化时发现的所有 PCI 设备及其配置信息列表，其配置信息多为某 PCI 设备相关 IRQ 信息，可读性不高，可以用 “/sbin/lspci -vb” 命令获得较易理解的相关信息；在 2.6 内核以后，此文件已为/proc/bus/pci 目录及其下的文件代替；

## 2.26、/proc/slabinfo

在内核中频繁使用的对象（如 inode、dentry 等）都有自己的 cache，即 slab pool，而 /proc/slabinfo 文件列出了这些对象相关 slab 的信息；详情可以参见内核文档中 slabinfo 的手册页；

```
[root@rhel5 ~]# more /proc/slabinfo

slabinfo - version: 2.1

# name          <active_objs> <num_objs> <objsize> <objperslab>
<pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : slabdata <ac
tive_slabs> <num_slabs> <sharedavail>

rpc_buffers      8      8   2048    2    1 : tunables   24    12     8 :
slabdata       4      4     0

rpc_tasks        8      20   192    20    1 : tunables  120    60     8 :
slabdata       1      1     0
```

```
rpc_inode_cache      6     9    448    9   1 : tunables  54    27    8 :  
slabdata      1     1     0  
.....  
.....  
.....
```

## 2.27、/proc/stat

实时追踪自系统上次启动以来的多种统计信息；如下所示，其中，

“cpu” 行后的八个值分别表示以 1/100 ( jiffies ) 秒为单位的统计值（包括系统运行于用户模式、低优先级用户模式，运系统模式、空闲模式、I/O 等待模式的时间等）；

“intr” 行给出中断的信息，第一个为自系统启动以来，发生的所有中断的次数；然后每个数对应一个特定的中断自系统启动以来所发生的次数；

“ctxt” 给出了自系统启动以来 CPU 发生的上下文交换的次数。

“btime” 给出了从系统启动到现在为止的时间，单位为秒；

“processes (total\_forks) 自系统启动以来所创建的任务的个数目；

“procs\_running”：当前运行队列的任务的数目；

“procs\_blocked”：当前被阻塞的任务的数目；

```
[root@rhel5 ~]# more /proc/stat
```

```
cpu 2751 26 5771 266413 2555 99 411 0
```

## 2.28、/proc/swaps

当前系统上的交换分区及其空间利用信息，如果有多个交换分区的话，则会每个交换分区的信息分别存储于`/proc/swap` 目录中的单独文件中，而其优先级数字越低，被使用到的可能性越大；下面是作者系统中只有一个交换分区时的输出信息：

```
[root@rhel5 ~]# more /proc/swaps
```

Filename	Type	Size	Used
Priority			
/dev/sda8	partition	642560 0	-1

## 2.29、/proc/uptime

系统上次启动以来的运行时间，如下所示，其第一个数字表示系统运行时间，第二个数字表示系统空闲时间，单位是秒；

```
[root@rhel5 ~]# more /proc/uptime
```

```
3809.86 3714.13
```

## 2.30、/proc/version

当前系统运行的内核版本号，在作者的 RHEL5.3 上还会显示系统安装的 gcc 版本，如下所示；

```
[root@rhel5 ~]# more /proc/version
```

```
Linux version 2.6.18-128.el5 (mockbuild@hs20-bc1-5.build.redhat.com) (gcc  
version 4.1.2 20080704 (Red Hat 4.1.2-44)) #1 SMP Wed Dec 17 11:42:39 EST 2008
```

### 2.31、/proc/vmstat

当前系统虚拟内存的多种统计数据，信息量可能会比较大，这因系统而有所不同，可读性较好；下面为作者机器上输出信息的一个片段；（2.6以后的内核支持此文件）

```
[root@rhel5 ~]# more /proc/vmstat
```

```
nr_anon_pages 22270
```

```
nr_mapped 8542
```

```
nr_file_pages 47706
```

```
nr_slab 4720
```

```
nr_page_table_pages 897
```

```
nr_dirty 21
```

```
nr_writeback 0
```

```
.....
```

### 2.32、/proc/zoneinfo

内存区域（zone）的详细信息列表，信息量较大，下面列出的是一个输出片段：

```
[root@rhel5 ~]# more /proc/zoneinfo
```

Node 0, zone	DMA
--------------	-----

pages free	1208
------------	------

min	28
-----	----

low 35  
high 42  
active 439  
inactive 1139  
scanned 0 (a: 7 i: 30)  
spanned 4096  
present 4096  
  
nr\_anon\_pages 192  
  
nr\_mapped 141  
  
nr\_file\_pages 1385  
  
nr\_slab 253  
  
nr\_page\_table\_pages 2  
  
nr\_dirty 523  
  
nr\_writeback 0  
  
nr\_unstable 0  
  
nr\_bounce 0  
  
protection: (0, 0, 296, 296)  
  
pagesets  
  
all\_unreclaimable: 0  
  
prev\_priority: 12

```
start_pfn:      0
```

```
.....
```

### 三、/proc/sys 目录详解

与/proc 下其它文件的“只读”属性不同的是，管理员可对/proc/sys 子目录中的许多文件内容进行修改以更改内核的运行特性，事先可以使用“ls -l”命令查看某文件是否“可写入”。写入操作通常使用类似于“echo DATA > /path/to/your/filename”的格式进行。需要注意的是，即使文件可写，其一般也不可以使用编辑器进行编辑。

#### 3.1、/proc/sys/debug 子目录

此目录通常是一空目录；

#### 3.2、/proc/sys/dev 子目录

为系统上特殊设备提供参数信息文件的目录，其不同设备的信息文件分别存储于不同的子目录中，如大多数系统上都会具有的/proc/sys/dev/cdrom 和/proc/sys/dev/raid（如果内核编译时开启了支持 raid 的功能）目录，其内存储的通常是系统上 cdrom 和 raid 的相关参数信息文件。

# 实验 24 中断的基础知识

## 24.1 本章导读

中断的用处非常广泛，本实验先给大家介绍一些基础知识，因为以前学习单片机或者纯粹的上位机软件可能并不会涉及到其中的一些基础知识，有了这些基础知识就容易理解后面中断相关的代码。

### 24.1.1 工具

#### 24.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 24.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）

### 24.1.2 预备课程

无

### 24.1.3 视频资源

本节配套视频为“视频 24\_中断的基础知识”

## 24.2 学习目标

本章需要学习以下内容：

中断的概念

中断源的概念

中断向量号等等

## 24.3 中断的基础知识

### 24.3.1 什么是中断？

中断是指 CPU 在执行程序的过程中，出现突发事件去处理，CPU 需要停止当前程序的执行，转去处理突发事件，处理完成之后再返回原程序部分。

### 24.3.2 什么是中断源？

引发中断的原因

### 24.3.3 硬件中断和软件中断

硬件中断一般指外设发出的中断请求以及内部硬件产生的中断（计算溢出，除数为 0，掉电等）

软件中断，典型的是中断处理程序的下半部操作。

#### 24.3.4 硬件中断的分类

内部中断：内部硬件产生的中断（例如：除数为 0）

外部中断：外设产生的中断（重点）

#### 24.3.5 外部中断的触发方式

上升沿触发和下降沿触发

电平触发

#### 24.3.6 中断优先级

系统根据中断事件的重要性和紧迫程度，将中断源分为若干个等级，优先级高的先执行。

#### 24.3.7 中断处理函数

中断产生之后执行的一段代码。

#### 24.3.8 中断向量号

中断源的识别标志，是跳往中断程序的“入口地址”。

#### 24.3.9 中断向量和非中断向量

硬件提供中断处理函数的地址

软件通过判断之后，提供中断处理函数的最终地址

### 24.3.10 向量中断和非向量中断的判断方法

一般一个中断号对应一个中断函数就是向量中断（独立按键）

多个中断函数共用一个中断号（矩阵键盘）

### 24.3.11 中断处理程序架构

操作系统中会产生很多中断，如果每一个中断都全部处理完之后再向后执行，是不可能的，所以就将中断处理程序分解为上半部和下半部。

例如给 PC 插入 U 盘会产生中断，接收之后，硬件会马上响应，中断操作会很快执行上半部分，然后就向上半部分通知系统调用对应的驱动程序。后面调用驱动的这个过程可以称之为下半部分。

上半部一般是和硬件紧密相关的代码，下半部一般是耗时的一些操作。

# 实验 25 中断之独立按键

## 25.1 本章导读

在学习了前面中断的知识，就可以尝试来写一个独立按键的驱动。本实验除了介绍 Datasheet 部分、中断函数以及中断的例子，还介绍了几个常用的调试方法，这些调试方法目前就现学现用，到了后面再给大家做一个总结。

### 25.1.1 工具

#### 25.1.1.1 硬件工具

- 1 ) iTOP4412 开发板
- 2 ) U 盘或者 TF 卡
- 3 ) PC 机
- 4 ) 串口

#### 25.1.1.2 软件工具

- 1 ) 虚拟机 VMware
- 2 ) Ubuntu12.04.2
- 3 ) 超级终端（串口助手）
- 4 ) 源码文件夹 “itop4412\_irq”

## 25.1.2 预备课程

24\_中断的基础知识

## 25.1.3 视频资源

本节配套视频为“视频 25\_中断之独立按键”

## 25.2 学习目标

本章需要学习以下内容：

中断的硬件知识

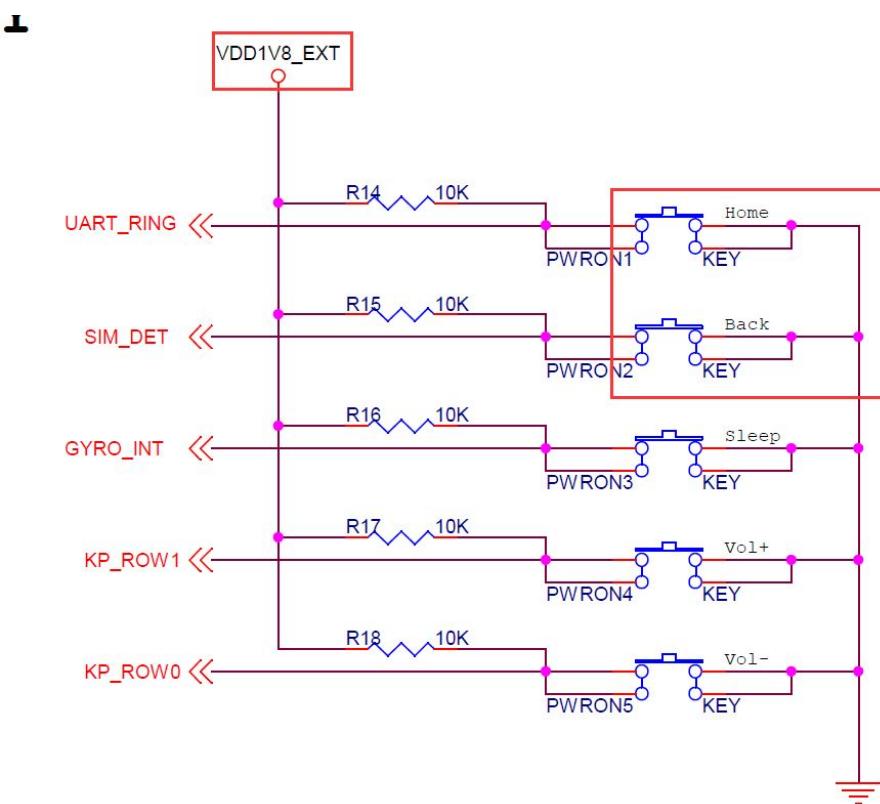
外部中断 datasheet 阅读

申请中断函数和中断处理函数

驱动程序的几个调试方法

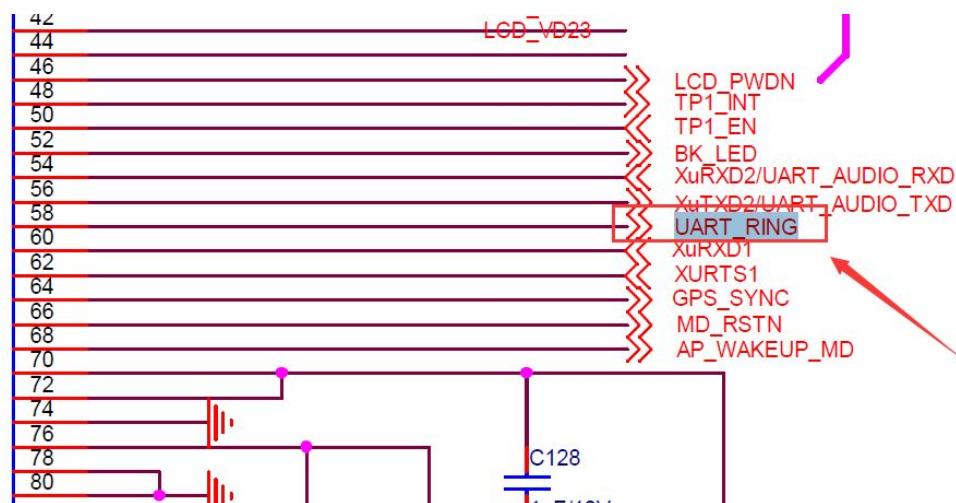
## 25.3 中断的硬件知识和外部中断 datasheet 阅读

底板上有 5 个在一起的独立按键，以 HOME 和 BACK 为例，先来看一下原理图，如下图所示。

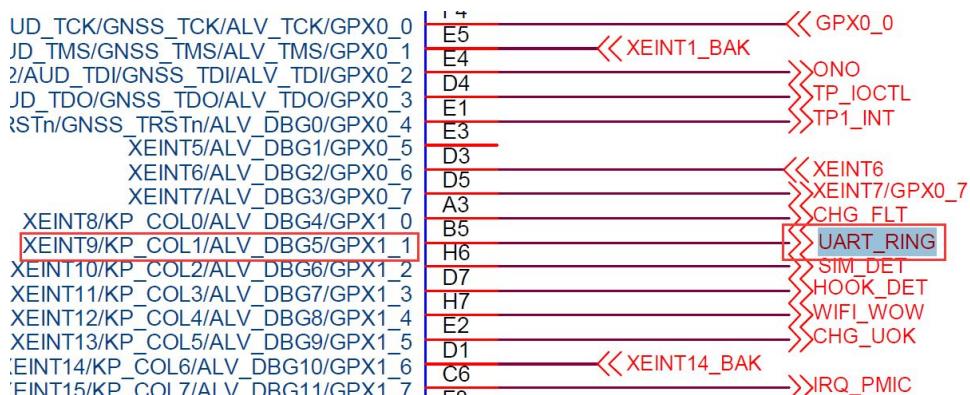


如上图所示，通过硬件连接可知，默认状态给中断的 IO 是高电平。所以可以判断后面对中断触发方式要为下降沿触发。

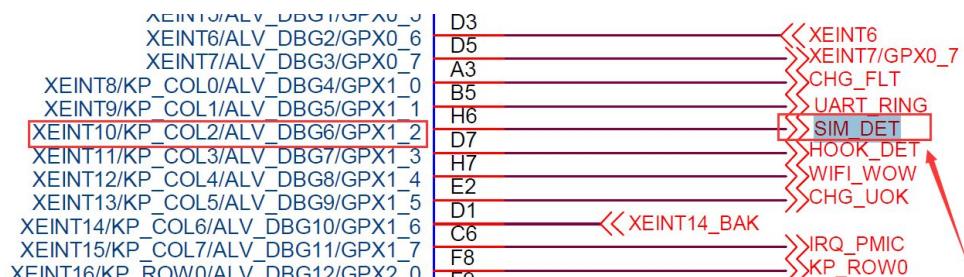
如下图所示，查找一下这个两个网络 UART\_RING 和 SIM\_DET。



然后打开核心板原理图文档，查找“UART\_RING”网络，如下图所示，对应的 GPIO 为“EXYNOS4\_GPX1(1)”，中断号为“XEINT9”。



另外一个网络，如下图所示，对应的 GPIO 为“EXYNOS4\_GPX1(2)”中断号为“XEINT10”。



然后打开 4412 的 Datasheet ,搜索“GPX1CON” ,如下图所示 ,下图以 SCP 的 Datasheet 为例 , POP 的也是类似。

#### 6.2.3.198 GPX1CON

- Base Address: 0x1100\_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000\_0000

Name	Bit	Type	Description	Reset Value
GPX1CON[7]	[31:28]	RW	0x0 = Input 0x1 = Output 0x2 = WAKEUP_INT1[7] 0x3 = KP_COL[7] 0x4 = Reserved 0x5 = ALV_DBG[11] 0x6 to 0xE = Reserved 0xF = EXT_INT41[7]	0x00
			0x0 = Input 0x1 = Output	

如下图所示，找到其中对应的“GPX1CON[1]”和“GPX1CON[2]”，如下图所示。

			<small>UAI - L&amp;I_INTERRUPT[2]</small>	
GPX1CON[2]	[11:8]	RW	0x0 = Input, 0x1 = Output, 0x2 = WAKEUP_INT1[2] 0x3 = KP_COL[2]	0x00

SAMSUNG ELECTRONICS

6-257



Preliminary product information describes products that are in development, for which full characterization data and associated errata are not yet available. Specifications and information herein are subject to change without notice.

*Samsung Confidential*

Exynos 4412\_UM

6 General Purpose Input/Output (GPIO) Control

Name	Bit	Type	Description	Reset Value
			0x4 = Reserved 0x5 = ALV_DBG[6] 0x6 to 0xE = Reserved <b>0xF = EXT_INT41[2]</b>	
GPX1CON[1]	[7:4]	RW	0x0 = Input 0x1 = Output 0x2 = WAKEUP_INT1[1], 0x3 = KP_COL[1], 0x4 = Reserved, 0x5 = ALV_DBG[5], 0x6 to 0xE = Reserved <b>0xF = EXT_INT41[1]</b> <small>0x0 - Input</small>	0x00

如上图所示，可以看到将管脚配置为外部中断模式需要设置为0xF。

最后再带大家看一个部分，在Datasheet的56章，找到Debouncing Filter，这是4412对于按键自动防抖的部分，如下图所示，可以看到有自带的防抖。

### 56.2 Debouncing Filter

Supports debouncing filter for keypad interrupt of any key input. The filtering width is approximately 62.5 usec ("FCLK" two-clock, when the FCLK is 32 kHz). The keypad interrupt (key pressed or key released) to the CPU in software scan mode is an ANDed signal of the all row input lines after filtering.

[Figure 56-2](#) illustrates the internal debouncing filter operation.

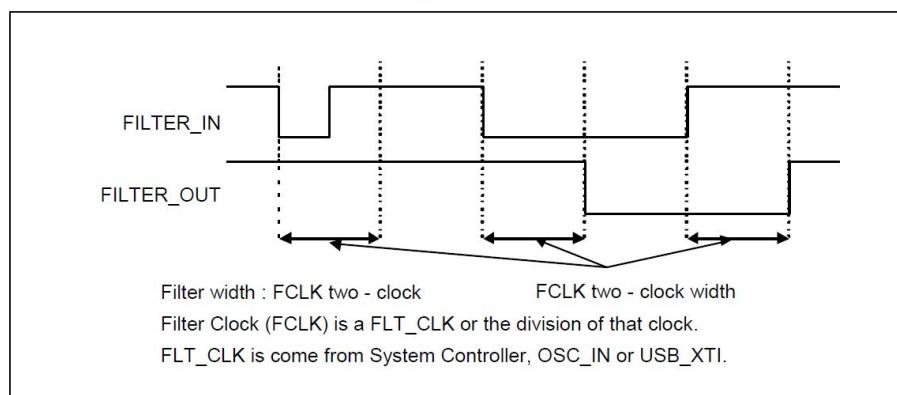


Figure 56-2 Internal Debouncing Filter Operation

## 25.4 中断相关函数简介

Linux 中的中断在使用前，都需要申请。中断申请函数是 “request\_irq” ，在头文件 “include/linux/interrupt.h” 中，如下图所示。

```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
} __cacheline_internodealigned_in_smp;

extern irqreturn_t no_action(int cpl, void *dev_id);

#ifndef CONFIG_GENERIC_HARDIRQS
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);

static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

extern int __must_check
request_any_context_irq(unsigned int irq, irq_handler_t handler,
                       unsigned long flags, const char *name, void *dev_id);

extern void exit_irq_thread(void);
#else
```

132,1 17%

如上图所示，中断申请函数 request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags,const char \*name, void \*dev)有下面几个参数。

参数 unsigned int irq : irq 是中断号

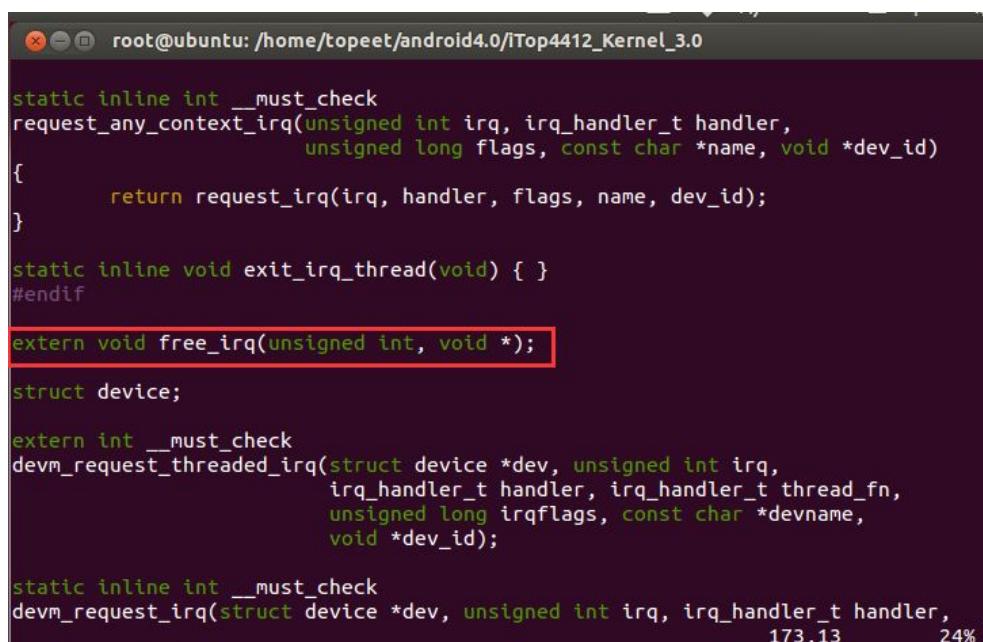
参数 irq\_handler\_t handler : handler 是向系统登记的处理函数

参数 unsigned long flags : irqflags 是触发标志位

参数 const char \*name : devname 是中断名称，可以通过注册之后可以通过 “cat /proc/interrupts” 查看

参数 void \*dev : dev\_id 是设备

和上面中断申请函数对应的就是中断释放函数 free\_irq，卸载驱动的时候需要调用，如下图所示，也是在头文件 “include/linux/interrupt.h” 中。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
static inline int __must_check
request_any_context_irq(unsigned int irq, irq_handler_t handler,
                       unsigned long flags, const char *name, void *dev_id)
{
    return request_irq(irq, handler, flags, name, dev_id);
}

static inline void exit_irq_thread(void) { }

#ifndef CONFIG_SMP
extern void free_irq(unsigned int, void *);
#endif

struct device;

extern int __must_check
devm_request_threaded_irq(struct device *dev, unsigned int irq,
                          irq_handler_t handler, irq_handler_t thread_fn,
                          unsigned long irqflags, const char *devname,
                          void *dev_id);

static inline int __must_check
devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
```

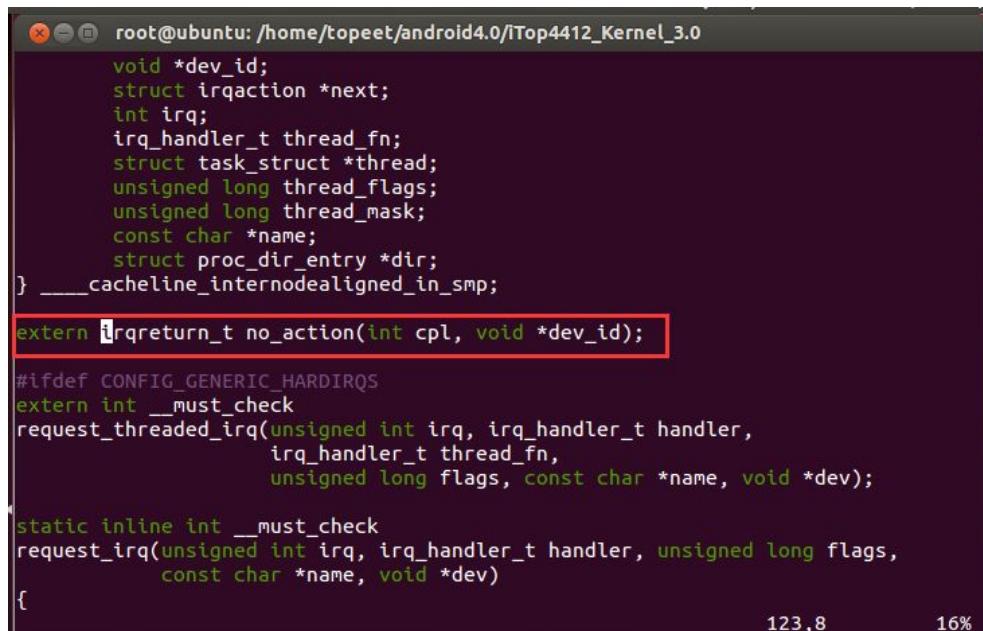
如上图所示中断释放函数 extern void free\_irq(unsigned int, void \*);的参数如下。

参数 1 : irq 是中断号

参数 2 : dev\_id 是设备

产生中断之后，会调用中断处理函数 irqreturn\_t，这个函数也是在头文件

“include/linux/interrupt.h” 中如下图所示。



The screenshot shows a terminal window with the following kernel source code:

```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
void *dev_id;
struct irqaction *next;
int irq;
irq_handler_t thread_fn;
struct task_struct *thread;
unsigned long thread_flags;
unsigned long thread_mask;
const char *name;
struct proc_dir_entry *dir;
} __cacheline_internodealigned_in_smp;

extern irqreturn_t no_action(int cpl, void *dev_id);

#ifndef CONFIG_GENERIC_HARDIRQS
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);
#endif

static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
           const char *name, void *dev)
{
```

123,8 16%

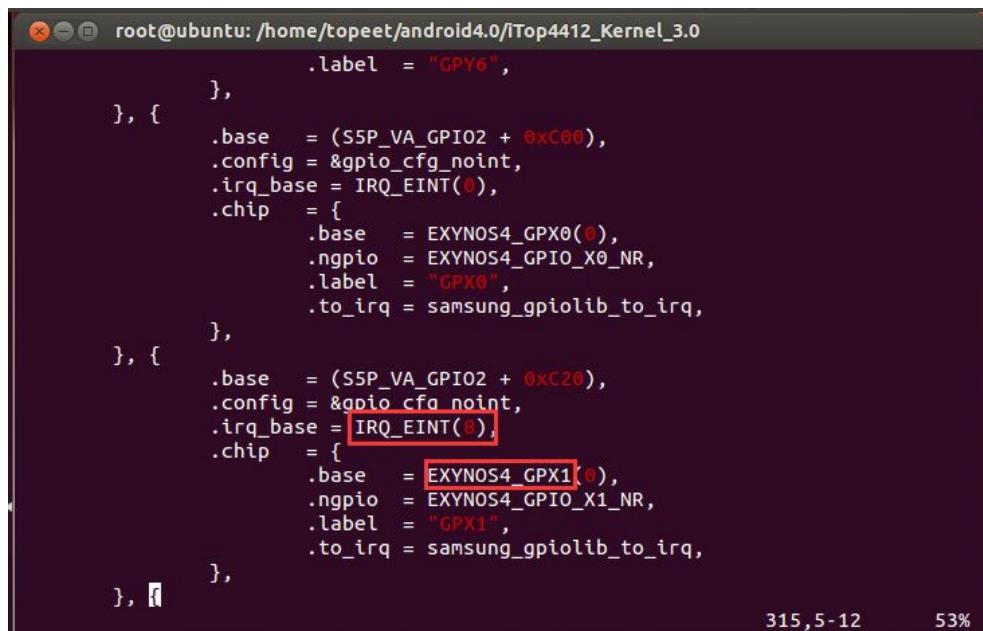
如上图所示，该函数为 extern irqreturn\_t no\_action(int cpl, void \*dev\_id);

中断函数类型为 irqreturn\_t

参数 int cpl : 中断号

参数 void \*dev\_id : 设备

在初始化文件 “drivers/gpio/gpio-exynos4.c” 中，如下图所示，



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
        .label = "GPY6",
    },
},
{
    .base   = (S5P_VA_GPIO2 + 0xC00),
    .config = &gpio_cfg_noint,
    .irq_base = IRQ_EINT(0),
    .chip   = {
        .base   = EXYNOS4_GPX0(0),
        .ngpio  = EXYNOS4_GPIO_X0_NR,
        .label   = "GPX0",
        .to_irq = samsung_gpiolib_to_irq,
    },
},
{
    .base   = (S5P_VA_GPIO2 + 0xC20),
    .config = &gpio_cfg_noint,
    .irq_base = IRQ_EINT(8),
    .chip   = {
        .base   = EXYNOS4_GPX1(0),
        .ngpio  = EXYNOS4_GPIO_X1_NR,
        .label   = "GPX1",
        .to_irq = samsung_gpiolib_to_irq,
    },
},
```

315,5-12 53%

选取的两个中断管脚都属于“GPX1”，所以如上图所示，中断编号的基础数值是IRQ\_EINT(8)，那么GPX1CON[0]和GPX1CON[1]则对应着中断号IRQ\_EINT(9)和IRQ\_EINT(10)。

如果想调用其他外部中断也可以通过这种方式来查阅中断号，不过前提是这个中断没有被占用。

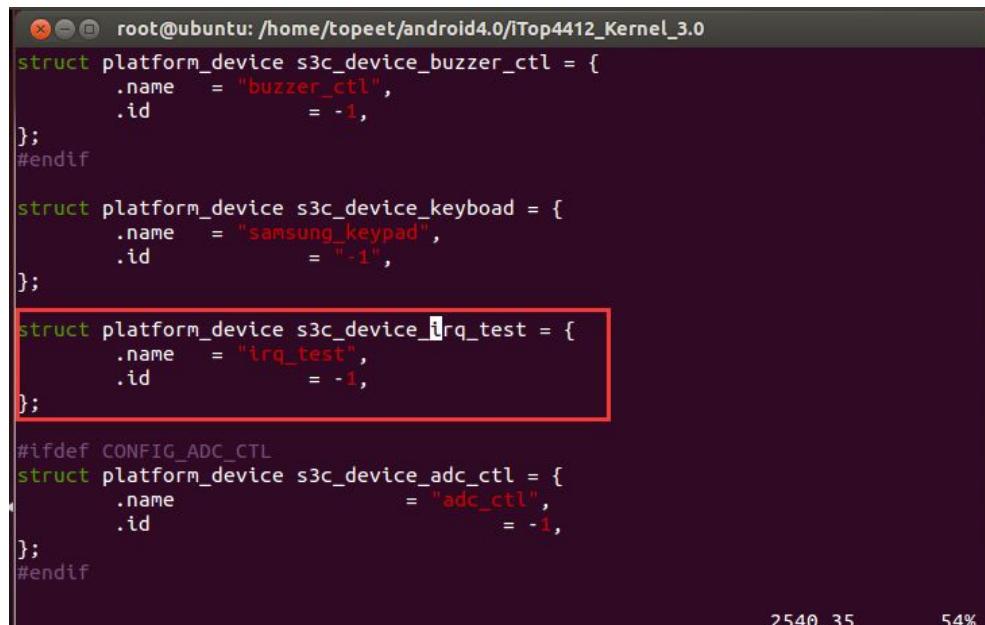
## 25.5 实验操作

外部中断的实际应用一般是集成在一些类似声卡、显卡、其他总线设备中，实际的应用也就是调用前面提到的头文件和函数，在驱动初始化的时候申请中断，然后针对具体的驱动写中断函数处理函数。例如声卡中调音量的按键，就是将按键部分的代码集成到声卡中，检测到按键就可以对应的调低调高音量。

这里注册一个简单的字符驱动，这样的话会便于大家的理解。要完成的功能就是按键中断产生之后打印数据。

如下图所示，首先注册设备，如下图所以，在平台文件

“arch/arm/mach-exynos/mach-itop4412.c” 中添加注册设备的代码。



The screenshot shows a terminal window with the title "root@ubuntu: /home/topeet/android4.0/iTop4412\_Kernel\_3.0". The code in the terminal is as follows:

```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
struct platform_device s3c_device_buzzer_ctl = {
    .name    = "buzzer_ctl",
    .id      = -1,
};

#endif

struct platform_device s3c_device_keyboard = {
    .name    = "samsung_keypad",
    .id      = "-1",
};

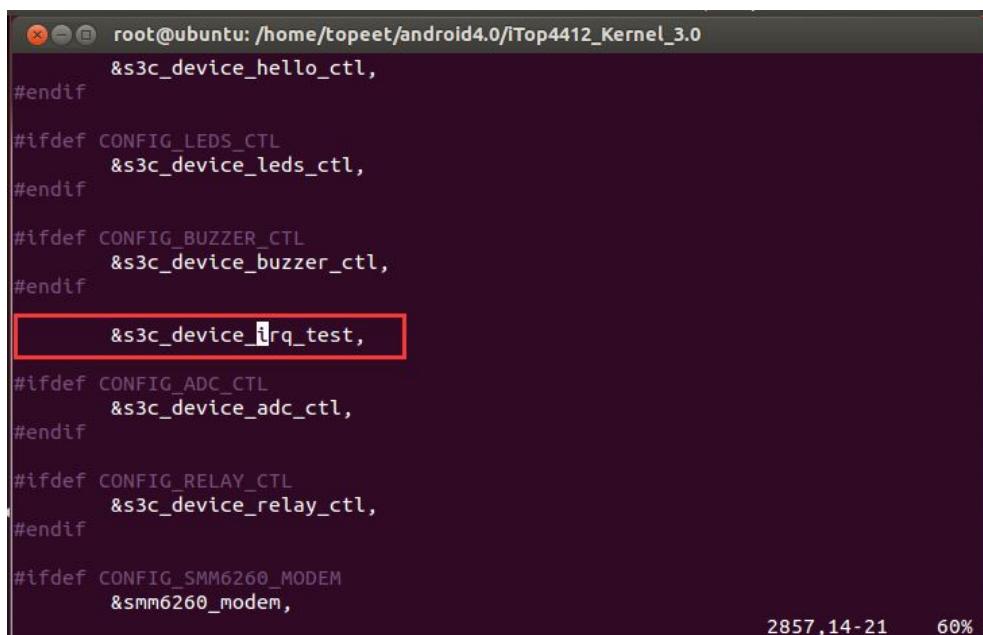
struct platform_device s3c_device_irq_test = {
    .name    = "irq_test",
    .id      = -1,
};

#ifndef CONFIG_ADC_CTL
struct platform_device s3c_device_adc_ctl = {
    .name    = "adc_ctl",
    .id      = -1,
};
#endif
```

The line "struct platform\_device s3c\_device\_irq\_test = {" is highlighted with a red rectangle.

如上图所示，为了简单，可以也可以不定义宏变量，强制注册设备。

如下图所示，添加设备调用的代码。



```
root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
    &s3c_device_hello_ctl,
#endif

#ifndef CONFIG_LEDS_CTL
    &s3c_device_leds_ctl,
#endif

#ifndef CONFIG_BUZZER_CTL
    &s3c_device_buzzer_ctl,
#endif

    &s3c_device_irq_test, &s3c_device_irq_test,

#ifndef CONFIG_ADC_CTL
    &s3c_device_adc_ctl,
#endif

#ifndef CONFIG_RELAY_CTL
    &s3c_device_relay_ctl,
#endif

#ifndef CONFIG_SMM6260_MODEM
    &smm6260_modem,
```

2857, 14-21 60%

然后打开 menuconfig 配置文件，将使用这两个中断的驱动卸载掉。

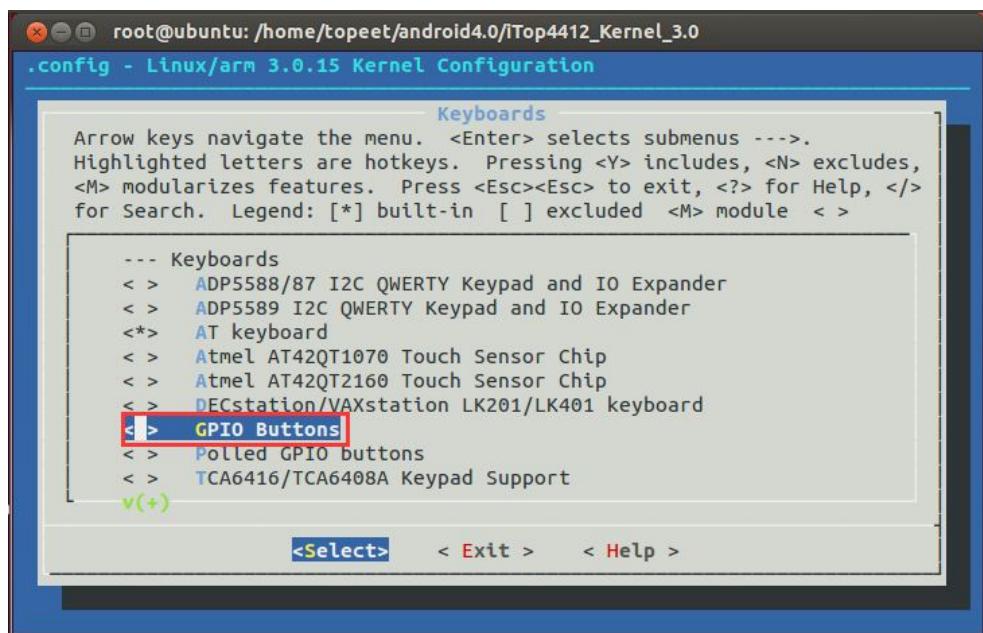
Device Drivers --->

Input device support --->

Keyboards --->

取消 GPIO Buttons --->

如下图所示。



重新设置之后，将内核重新编译，并将心生成的二进制 zImage 文件烧写到开发板中替换原来的内核。

然后将驱动的基本框架完成。

如下图所示，头文件部分，以后写 4412 的驱动可以将这些头文件一股脑的添加到代码前面。

```
/*以后写驱动可以讲头文件一股脑的加载代码前面*/
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <mach/gpio.h>
#include <plat/gpio-cfg.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>
#include <mach/regs-gpio.h>
#include <asm/io.h>
#include <linux/regulator/consumer.h>
#include <linux/delay.h>

/*中断函数头文件*/
#include <linux/interrupt.h>
#include <linux/irq.h>
```

然后是驱动模块的入口函数和出口函数，如下图所示。

```
static struct platform_driver irq_driver = {
    .probe = irq_probe,
    .remove = irq_remove,
    .suspend = irq_suspend,
    .resume = irq_resume,
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
    },
};

static void __exit irq_test_exit(void)
{
    platform_driver_unregister(&irq_driver);
}

static int __init irq_test_init(void)
{
    return platform_driver_register(&irq_driver);
}

module_init(irq_test_init);
module_exit(irq_test_exit);

MODULE_LICENSE("Dual BSD/GPL");
```

如下图所示，是 platform\_driver 的 irq\_resume、irq\_suspend、irq\_remove 函数，在 irq\_remove 函数中使用中断释放函数。

```
static int irq_remove (struct platform_device *pdev)
{
    free_irq(IRQ_EINT(9), pdev);
    free_irq(IRQ_EINT(10), pdev);

    return 0;
}

static int irq_suspend (struct platform_device *pdev, pm_message_t state)
{
    DPRINTK("irq suspend:power off!\n");
    return 0;
}

static int irq_resume (struct platform_device *pdev)
{
    DPRINTK("irq resume:power on!\n");
    return 0;
}
```

接着重点介绍一下初始化函数 irq\_probe，如下图所示。首先对中断 IO 进行检测，是否被占用了，处理方式一般就是申请 IO，看是否成功，申请成功之后就将 GPIO 配置为上拉模式，然后调用 gpio\_free 将其释放。

```
static int irq_probe(struct platform_device *pdev)
{
    int ret;
    char *banner = "irq_test Initialize\n";

    printk(banner);

    ret = gpio_request(EXYNOS4_GPX1(1), "EINT9");
    if (ret) {
        printk("%s: request GPIO %d for EINT9 failed, ret = %d\n", DRIVER_NAME,
               EXYNOS4_GPX1(1), ret);
        return ret;
    }

    s3c_gpio_cfgpin(EXYNOS4_GPX1(1), S3C_GPIO_SFN(0xF));
    s3c_gpio_setpull(EXYNOS4_GPX1(1), S3C_GPIO_PULL_UP);
    gpio_free(EXYNOS4_GPX1(1));
```

然后对另一个中断 IO 进行初始化，如下图所示。

```

s3c_gpio_cfgpin(EXYNOS4_GPX1(2), S3C_GPIO_SFN(0xF));
s3c_gpio_setpull(EXYNOS4_GPX1(2), S3C_GPIO_PULL_UP);
gpio_free(EXYNOS4_GPX1(2));

ret = request_irq(IRQ_EINT(9), eint9_interrupt,
                  IRQ_TYPE_EDGE_FALLING /*IRQ_TRIGGER_FALLING*/, "eint9", pdev);
if (ret < 0) {
    printk("Request IRQ %d failed, %d\n", IRQ_EINT(9), ret);
    goto exit;
}

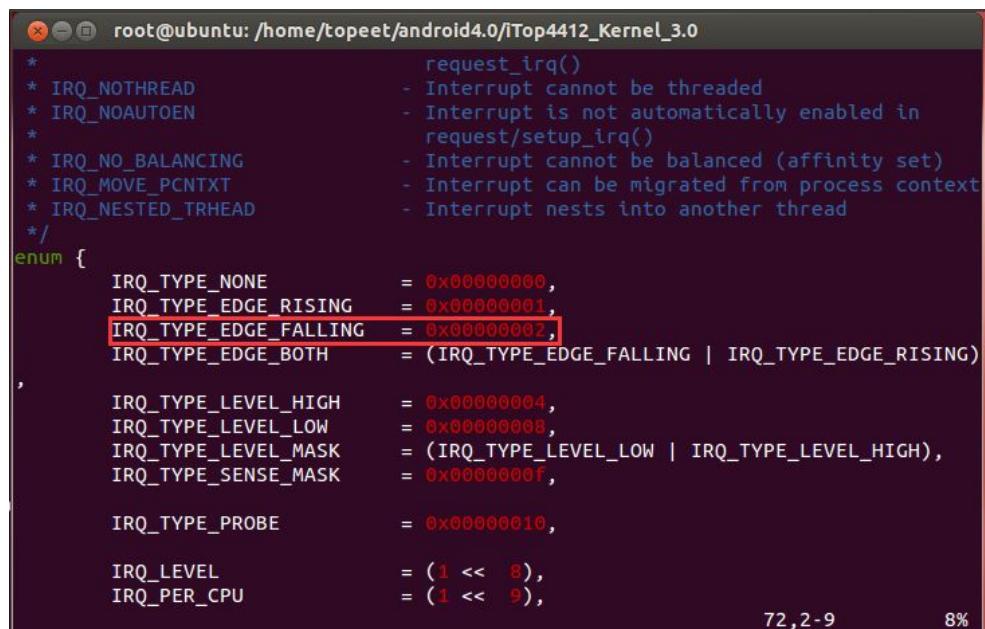
ret = request_irq(IRQ_EINT(10), eint10_interrupt,
                  IRQ_TYPE_EDGE_FALLING /*IRQ_TRIGGER_FALLING*/, "eint10", pdev);
if (ret < 0) {
    printk("Request IRQ %d failed, %d\n", IRQ_EINT(10), ret);
    goto exit;
}

return 0;

exit:
return ret;
}

```

如上图所示，对 IO 初始化之后，调用中断申请函数 `request_irq`，申请这两个中断。其中用到了参数 “`IRQ_TYPE_EDGE_FALLING`” ，这个代表下降沿触发，这个宏定义在头文件 “`include/linux/irq.h`” 中，如下图所示。



```

root@ubuntu:/home/topeet/android4.0/iTop4412_Kernel_3.0
*           request_irq()
* IRQ_NO_THREAD          - Interrupt cannot be threaded
* IRQ_NOAUTOEN          - Interrupt is not automatically enabled in
*                         request/setup_irq()
* IRQ_NO_BALANCING      - Interrupt cannot be balanced (affinity set)
* IRQ_MOVE_PCNTXT       - Interrupt can be migrated from process context
* IRQ_NESTED_TRHEAD     - Interrupt nests into another thread
*/
enum {
    IRQ_TYPE_NONE          = 0x00000000,
    IRQ_TYPE_EDGE_RISING   = 0x00000001,
    IRQ_TYPE_EDGE_FALLING  = 0x00000002,
    IRQ_TYPE_EDGE_BOTH     = (IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING),
    ,
    IRQ_TYPE_LEVEL_HIGH    = 0x00000004,
    IRQ_TYPE_LEVEL_LOW     = 0x00000008,
    IRQ_TYPE_LEVEL_MASK    = (IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH),
    IRQ_TYPE_SENSE_MASK    = 0x0000000f,
    ,
    IRQ_TYPE_PROBE         = 0x00000010,
    ,
    IRQ_LEVEL              = (1 << 8),
    IRQ_PER_CPU             = (1 << 9),

```

接着就可以定义中断处理函数，如下图所示，对这两个中断分别定义 `irqreturn_t` 函数。

```
static irqreturn_t eint9_interrupt(int irq, void *dev_id) {
    printk("%s(%d)\n", __FUNCTION__, __LINE__);
    return IRQ_HANDLED;
}

static irqreturn_t eint10_interrupt(int irq, void *dev_id) {
    printk("%s(%d)\n", __FUNCTION__, __LINE__);
    return IRQ_HANDLED;
}
```

如上图所示，上面使用了一句比较特殊的代码 “`printk("%s(%d)\n", __FUNCTION__, __LINE__);`” 这句代码在调试的过程中非常有用，就是打印当前所在的函数以及对应的行，在后面测试的时候就可以看到其效果。

另外将打印函数做一个简单的处理，在调试过程中少不了用到打印函数，在调试完了之后，要一个一个的去删除打印信息会很麻烦，这里将打印函数做一个宏定义。如果定义了宏变量 “`IRQ_DEBUG`” ，那么运行驱动的时候就会打印 `DPRINTK` 就会生效。当然还有一些很重要的信息，例如在初始化中，有些信息可能需要每次运行的时候都打印，如下图所示。

```
/*中断函数头文件*/
#include <linux/interrupt.h>
#include <linux/irq.h>

#define IRQ_DEBUG
#ifndef IRQ_DEBUG
#define DPRINTK(x...) printk("IRQ_CTL DEBUG:" x)
#else
#define DPRINTK(x...)
#endif

#define DRIVER_NAME "irq_test"
```

如下图所示，简单修改一下编译文件 Makefile，如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_irq.c这个文件编译成中间文件itop4412_irq.o
obj-m += itop4412_irq.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

修改完成之后，在 Ubuntu 系统下使用命令 “mkdir irq\_test” ，新建文件夹 “irq\_test” ，

然后将修改好的驱动文件 “itop4412\_irq.c” 、 Makefile 文件拷贝到文件夹 “irq\_test” 中，如下图所示。

```
root@ubuntu:/home/topeet/irq_test
root@ubuntu:/home/topeet# mkdir irq_test
root@ubuntu:/home/topeet# cd irq_test/
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  Makefile
```

使用编译命令“make” 编译驱动，如下图所示。

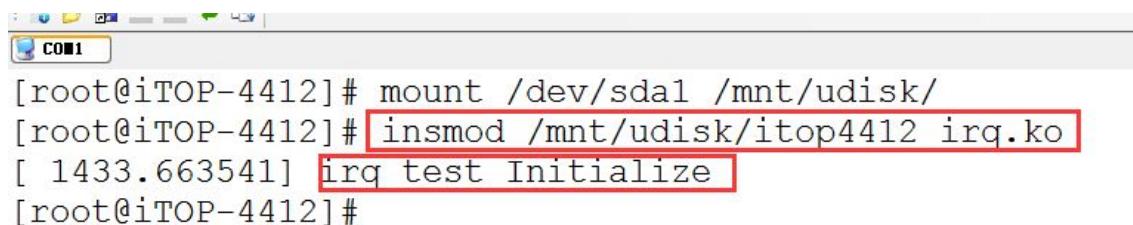
```
root@ubuntu:/home/topeet/irq_test
root@ubuntu:/home/topeet# mkdir irq_test
root@ubuntu:/home/topeet# cd irq_test/
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  Makefile
root@ubuntu:/home/topeet/irq_test# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/irq_test modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/irq_test/itop4412_irq.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/irq_test/itop4412_irq.mod.o
  LD [M] /home/topeet/irq_test/itop4412_irq.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  itop4412_irq.mod.c  itop4412_irq.o  modules.order
itop4412_irq.ko  itop4412_irq.mod.o  Makefile        Module.symvers
root@ubuntu:/home/topeet/irq_test#
```

将生成的驱动模块“itop4412\_irq.ko”拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令“mount /dev/sda1 /mnt/udisk/”加载 U 盘，如下图所示。

```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
```

使用命令 “insmod /mnt/udisk/itop4412\_irq.ko” 加载驱动模块，如下图所示。



```
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/itop4412_irq.ko
[ 1433.663541] irq test Initialize
[root@iTOP-4412]#
```

如上图所示，打印了初始化代码，没有打印任何错误。

接着使用命令 “cat /proc/interrupts” 查看申请的中断。

```
[root@iTOP-4412]# cat /proc/interrupts
CPU0      CPU1      CPU2      CPU3
24:       58        0        0        0    s3c-uart   s5pv210-uart
26:      119        0        0        0    s3c-uart   s5pv210-uart
98:       0        0        0        0      GIC      s3c-p1330.0
99:       0        0        0        0      GIC      s3c-p1330.1
100:      0        0        0        0      GIC      s3c-p1330.2
107:      0        0        0        0      GIC      s3c2410-wdt
108:      0        0        0        0      GIC      s3c2410-rtc alarm
121:       9        0        0        0      GIC      mct_comp_irq
123:     42509        0        0        0      GIC      s3c2440-i2c.1
125:       1        0        0        0      GIC      s3c2440-i2c.3
126:       60        0        0        0      GIC      s3c2440-i2c.4
127:       0        0        0        0      GIC      s3c2440-i2c.5
129:       6        0        0        0      GIC      s3c2440-i2c.7
134:     49064        0        0        0      GIC      ehci_hcd:usb1
135:       41        0        0        0      GIC      s3c-udc
139:       0        0        0        0      GIC      mmc1
140:       52        0        0        0      GIC      mmc2
141:     3687        0        0        0      GIC      mmc0
160:       0        0        0        0      GIC      samsung-rp
281:       0        0        0        0  COMBINER  s3cfb
352:       1        0        0        0      exynos-eint
359:       0        0        0        0      exynos-eint  s3c-sdhci.2
361:       0        0        0        0      exynos-eint  eint9
362:       0        0        0        0      exynos-eint  eint10
367:       1        0        0        0      exynos-eint  s5m87xx-irq
370:       0        0        0        0      exynos-eint  switch-gpio
428:       0        0        0        0      s5m8767  rtc-alarm0
      n        n        n        n      Timer broadcast interrupts
```

使用命令 “rmmod itop4412\_irq” 卸载中断的驱动，如下图所示。

```
1P15:          U          U          U          U      CPU backtrace
LOC:    35800      54125      37631      40424 Local timer interrupts
Err:      0
[root@iTOP-4412]# lsmod
itop4412_irq 2142 0 - Live 0xbff00000
[root@iTOP-4412]# rmmod itop4412_irq
[root@iTOP-4412]#
```

接着再次使用命令 “cat /proc/interrupts” 查看申请的中断，如下图所示已经没有了。

```
141:    3932      0      0      0      GIC mmc0
160:      0      0      0      0      GIC samsung-rp
281:      0      0      0      0      COMBINER s3cfb
352:      1      0      0      0      exynos-eint
359:      0      0      0      0      exynos-eint s3c-sdhci.2
367:      1      0      0      0      exynos-eint s5m87xx-irq
370:      0      0      0      0      exynos-eint switch-gpio
428:      0      0      0      0      s5m8767 rtc-alarm0
IPIO:      0      0      0      0      Timer broadcast interrupts
IPI1:   9237    12294    9163    8835 Rescheduling interrupts
IPI2:     97      94     102    105 Function call interrupts
IPI3:     60      40      78      53 Single function call interrupts
IPI4:      0      0      0      0      CPU stop interrupts
IPI5:      0      0      0      0      CPU backtrace
...       ...     ...     ...     ... Total system interrupt counts
```

接着使用命令 “insmod /mnt/udisk/itop4412\_irq.ko” 加载驱动模块，测试功能。然后

按几下 HOME 和 BACK 按键，会出现类似下面的打印信息。

```
[root@iTOP-4412]# insmod /mnt/udisk/itop4412_irq.ko
[ 1732.895078] irq_test Initialize
[root@iTOP-4412]# [ 1736.489298] eint9_interrupt(32)
[ 1736.632151] eint9_interrupt(32)
[ 1737.393576] eint10_interrupt(39)
[ 1738.210704] eint10_interrupt(39)
[ 1739.196217] eint10_interrupt(39)
```

再回过头去看一下源码，如下图所示，32 行和 39 行正是对应的那两处打印的代码，分别在函数 eint9\_interrupt 和 eint10\_interrupt 中。

```
30 static irqreturn_t eint9_interrupt(int irq, void *dev_id) {
31
32     printk("%s(%d)\n", __FUNCTION__, __LINE__);
33
34     return IRQ_HANDLED;
35 }
36
37 static irqreturn_t eint10_interrupt(int irq, void *dev_id) {
38
39     printk("%s(%d)\n", __FUNCTION__, __LINE__);
40
41     return IRQ_HANDLED;
42 }
43
```

最后再使用命令“cat /proc/interrupts”查看申请的中断，如下图所示，已经检测到这两个中断分别触发了几次。

126:	60	0	0	0	GIC	s3c2440-i2c.4
127:	0	0	0	0	GIC	s3c2440-i2c.5
129:	6	0	0	0	GIC	s3c2440-i2c.7
134:	60220	0	0	0	GIC	ehci_hcd:usb1
135:	41	0	0	0	GIC	s3c-udc
139:	0	0	0	0	GIC	mmc1
140:	52	0	0	0	GIC	mmc2
141:	4443	0	0	0	GIC	mmc0
160:	0	0	0	0	GIC	samsung-rp
281:	0	0	0	0	COMBINER	s3cfb
352:	1	0	0	0	exynos-eint	
359:	0	0	0	0	exynos-eint	s3c-sdhci.2
361:	2	0	0	0	exynos-eint	eint9
362:	3	0	0	0	exynos-eint	eint10
367:	1	0	0	0	exynos-eint	s5m87xx-irq
370:	0	0	0	0	exynos-eint	switch-gpio
428:	0	0	0	0	s5m8767	rtc-alarm0
IPIO:	0	0	0	0	Timer broadcast	interrupts
IPI1:	10236	13922	10705	9993	Rescheduling	interrupts
IPI2:	111	108	115	118	Function call	interrupts
IPI3:	83	44	95	66	Single function call	interrupts
TPT4:	0	0	0	0	CPU stop	interrupts

## 联系方式

北京迅为电子有限公司致力于嵌入式软硬件设计，是高端开发平台以及移动设备方案提供商；基于多年的技术积累，在工控、仪表、教育、医疗、车载等领域通过 OEM/ODM 方式为客户创造价值。

iTOP-4412 开发板是迅为电子基于三星最新四核处理器 Exynos 4412 研制的一款实验开发平台，可以通过该产品评估 Exynos 4412 处理器相关性能，并以此为基础开发出用户需要的特定产品。

本手册主要介绍如何使用 iTOP-4412 开发板学习 Linux 驱动，旨在帮助用户快速学习 Linux 驱动。

如需‘平板电脑’产品级方案支持，请访问迅为平板方案网‘[www.topeet.com](http://www.topeet.com)’。我司将有能力为您提供全方位的技术服务，保证您产品设计无忧！

本手册将持续更新，并通过多种方式发布给新老用户，希望迅为电子的努力能给您的学习和开发带来帮助。

迅为电子

2015 年 9 月