Android 系统程序员开发指南

–based on Android 4.0.3-mr1 with linux kernel version 3.0.8

Revision 0.1

Jul 17, 2012

北京君正集成电路有限公司
Ingenic Semiconductor Co. Ltd

Important Notice

This documentation is provided for use with Ingenic products. No license to Ingenic property rights is granted. Ingenic assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement except as provided for by Ingenic Terms and Conditions of Sale.

Ingenic products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. Ingenic may make changes to this document without notice. Anyone relying on this documentation should contact Ingenic for the current documentation and errata.

北京君正集成电路有限公司
北京市海淀区东北旺西路 8 号中关村软件园一号楼
信息中心 A 座 108 室，100193
Tel: 86-10-82826661
Fax: 86-10-82825845
http://www.ingenic.cn

Revision History

Revision	Date	Author(s)	Description
0.1	Jul 17, 2012	Lutts	Creation

Contents

Part 1

bootloader

Chapter 1

bootloader

1.1 代码结构

```
xboot-h700 (git version: c0a252be21d3a16c55acea2df315682c561cacd6)
├── spl: Second Program Loader, 负责初始化 CPU(pll, ddr), 并加载 xboot
│   ├── common: mddr/ddr2 初始化, 串口初始化, 一些通用函数的实现
│   ├── init: 只有一个 init.S, 提供启动方式识别, 从汇编跳转到 C, 休眠唤醒判断
│   ├── msc: MSC 启动的核心入口, 调用 pll、ddr 初始化函数, 从 MMC/SD 上加载 xboot
│   ├── nand: Nand 启动的核心入口, 调用 pll、ddr 初始化函数, 从 Nand 加载 xboot
│   └── tools: 一些辅助性的 PC 端小工具, 由主 Makefile 调用
├── tools
│   └── rsa: 加解密小工具
├── boot
│   ├── board: 板级文件, 包括板级特有的 GPIO 初始化, 分区表的设置等
│   ├── common: 一些通用函数的实现
│   ├── init: 只包含一个 init.S, 是 x-boot 的入口, 实现了 ldscript 中的 start entry
│   ├── lcd: lcd 驱动, 实现了显示开机画面等功能
│   ├── libc: x-boot 不依赖 libc, 但代码中用到了一些 C 函数, 这里实现了这些 C 函数
│   ├── libtomcrypt: 加解密库
│   ├── logo: 存放 logo 文件
│   ├── msc: MSC 驱动, MMC/SD 启动时的入口点
│   ├── nand: Nand 驱动
│   ├── tomsfastmath: 加解密库用到的数学库
│   └── usb: usb 驱动, 主要用于 fastboot
└── include
    ├── asm:
    │   └── jz_mem_nand_configs: 存放 DDR 及 Nand 的配置文件
    ├── configs: 板级配置文件放在这个目录下, 如 h700.h
    ├── lcd_configs: LCD 的板级文件, 实现 lcd 的初始化, 点屏, 关屏.
    └── libc: libc 的头文件
```

1.1.1 Tips

1. spl 的入口函数是 spl_main, 位于 spl/msc 或 spl/nand 中和 CPU 相应的.c 文件中
2. boot 的入口函数是 boot_main, 位于 boot/common/boot_main.c 中
3. jz_serial.c 中实现了一些串口打印函数, 可用于调试时打印信息, 但在 spl 中要慎用, 编译完后要查看 spl 目录下的 msc-spl.bin 或 nand-spl.bin, 确保大小不超过 8K(8192B). 下面是常用的函数:

(a) void serial_putc (const char c): 打印一个字符

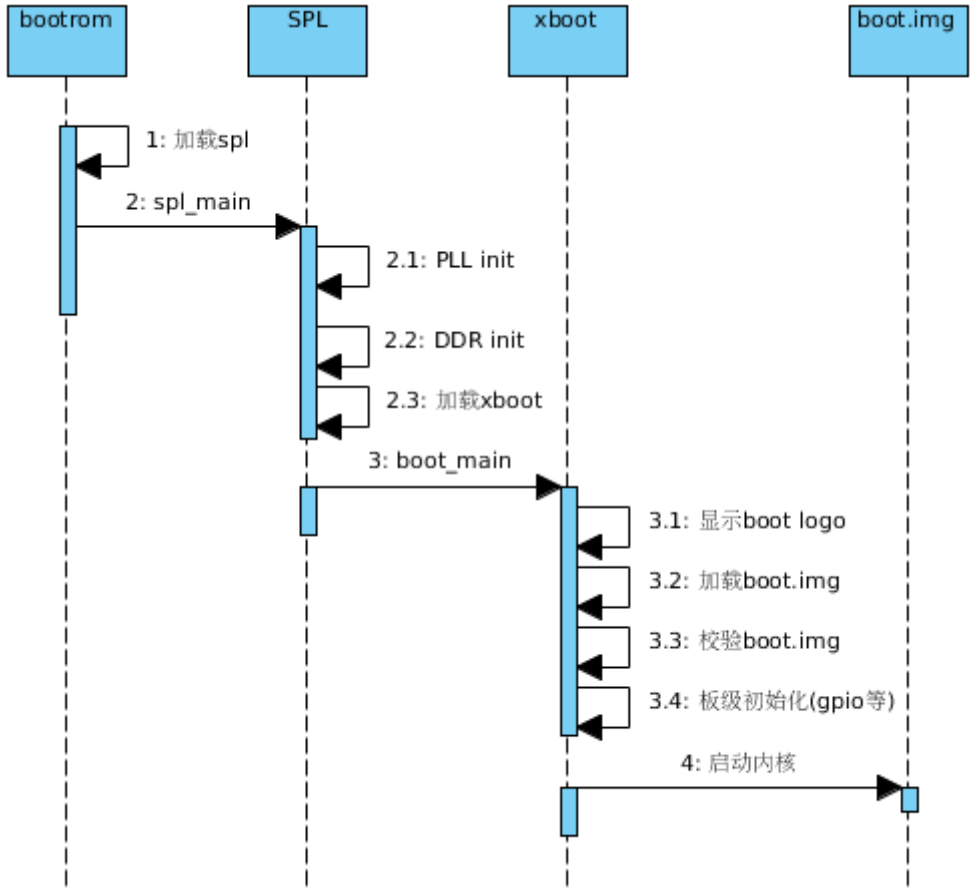
(b) void serial_puts (const char *s): 打印一个以 NULL 结尾的定附串

(c) void serial_put_hex(unsigned int d): 以十六进制打印 d

4. 调试及学习 bootloader 过程中要善用 while(1), 编译器很聪明, while(1) 之后相关的代码都会被优化掉, 从而可以暂时避开头疼的 8KB 问题, 又不用大把大把地删代码.

5. 有必要把主 Makefile 看一遍

下面是一个简单的 bootloader 启动过程, 描绘了启动过程中主要的步骤:



1.2 实例: 移植一个新板子

1. 修改主 Makefile, 增加 h700_msc_lvds_config, 拷贝自 o1_msc_config, 选 o1 是因为 o1 也是高清屏, 稍做修改, 修改后内容如下:

```
h700_msc_lvds_config:       unconfig
        @echo "#define CONFIG_MSC_X_BOOT" > include/config.h
        @echo "#define CONFIG_ANDROID_LCD_BBK_LVDS 1" >> include/config.h
        @echo "#define CONFIG_LCD_SYNC_MODE 1" >> include/config.h
        @echo "#define CONFIG_LCD_GET_PIXCLK_FROM_PLL1" >> include/config.h
        @echo "#define DEFAULT_BACKLIGHT_LEVEL  40" >> include/config.h
#       @echo "#define CONFIG_XBOOT_LOW_BATTERY_DETECT" >> include/config.h
```

```
8         @echo "#define I2C_BASE I2C1_BASE" >> include/config.h
9         @echo "#define CONFIG_JZ4770 1" >> include/config.h
10        @echo "#define CONFIG_JZ4770_H700 1" >> include/config.h
11        @echo "#define CONFIG_DDR2_DIFFERENTIAL 1" >> include/config.h
12        @echo "#define CONFIG_DDR2_DRV_CK_CS_FULL 1" >> include/config.h
13        @echo "#define CONFIG_MSC_TYPE_MMC" >> include/config.h
14        @echo "#define CONFIG_XBOOT_LOGO_FILE 1" >> include/config.h
15        @echo "#define CONFIG_XBOOT_LOGO_FILE_LEN 39448" >> include/config.h
16        @echo "Compile MSC boot image for soc jz4770 BBK H700 BOARD"
17        @./mkconfig jz4770 h700 msc
18        @echo "CONFIG_MSC_X_BOOT = y" > include/config.mk
19        @echo "CONFIG_JZ4770 = y" >> include/config.mk
20    #   @echo "CONFIG_POWER_MANAGEMENT = y" >> include/config.mk
21        @echo "CPU_TYPE=JZ4770" >> include/config.mk
22        @echo "CONFIG_JZ4770_H700 = y" >> include/config.mk
23        @echo "CONFIG_USE_DDR2 = y" >> include/config.mk
24        @echo "./_mklogo.sh bbk_800_600.rle" >> boot/mklogo.sh
25        @ln -s ${PWD}/boot/common/jz_serial.c ${PWD}/spl/common/jz_serial.c
26        @ln -s ${PWD}/boot/common/cpu.c ${PWD}/spl/common/cpu.c
27        @ln -s ${PWD}/boot/common/debug.c ${PWD}/spl/common/debug.c
28        @ln -s ${PWD}/boot/common/common.c ${PWD}/spl/common/common.c
```

这段代码写了两个配置文件, 一个是 include/config.h, 一个是 include/config.mk, config.h 由 C 代码使用, config.mk 由 Makefile 使用

对于各个选项的含义, 简单说明如下:

- CONFIG_MSC_X_BOOT: 从 MMC/SD 启动
- CONFIG_MSC_TYPE_SD/CONFIG_MSC_TYPE_MMC: eMMC 遵循 SD 协议或 MMC 协议
- CONFIG_ANDROID_LCD_BBK_LVDS: 这个宏决定了我们要使用的屏的型号, 稍后讲解 LCD 相关的配置时会看到他

  在设置这个宏时, 应该先到 include/lcd_configs 下去看看, 看是否已经有相关的支持了, 有的话就可以直接拿来用, 具体怎么用稍后讲解.
- DEFAULT_BACKLIGHT_LEVEL: bootloader 运行时的背光亮度
- CONFIG_XBOOT_LOW_BATTERY_DETECT: 是否在 bootloader 中进行低电检测
- CONFIG_DDR2_DIFFERENTIAL/CONFIG_DDR2_DRV_CK_CS_FULL: DDR 相关的配置, 视具体硬件而定, 4770 一般都是差分的, CK, CS 的驱动强度需要硬件工程师确定
- CONFIG_USE_DDR2: 使用 DDR2, 可能的选项还有 CONFIG_USE_MDDR 等
- CONFIG_XBOOT_LOGO_FILE/CONFIG_XBOOT_LOGO_FILE_LEN: 是否使用 logo, 如果使用, 由 CONFIG_XBOOT_LOGO_FILE_LEN 指定 logo 的文件大小 (字节数), logo 的位置放在 boot/logo 目录下, bbk_800_600.rle 是 logo 的文件名.

具体的含义请以代码为准.

完成主 Makefile 的修改后, 我们就可以使用以下命令来生成 config.h 和 config.mk 两个配置文件

```
$ make h700_msc_lvds_config
```

2. 准备板级文件。在 boot/board 下，从一个较为相似的板子拷贝一份，重命名为 h700, 这个目录下有两个文件，也要相应地重命名，并修改其中的 Makefile, 在 xxx_speicial.c 中，重新实现 board_-private_init，实现板级相关的 gpio 初始化等。修改 mbr.h，这里面主要是分区表的设置，要保证和内核中的分区表一致。

3. 增加 LCD 相关的代码

   - 修改 include/jz4770_android_lcd.h，参照已有代码添加相应屏的配置，如:

```
#if defined(CONFIG_ANDROID_LCD_BBK_LVDS)
#include "jz_bbk_lvds.h"
#endif
```

   - 其中 jz_bbk_lvds.h 位于 include/lcd_configs/目录下，在这个头文件里实现以下三个宏:
     - __lcd_special_pin_init
     - __lcd_special_on
     - __lcd_special_off
   - 在 boot/lcd/jz4770_lcd.c 中增加相应的时序配置，请参照已有代码，需要修改两处：
     - struct jz4760lcd_info jzfb
     - vidinfo_t panel_info
   - 背光设置，位于 include/jz4770_android_lcd.h 中，主要需要修改所使用的 PWM

4. 在 include/asm/jz_mem_nand_configs 下，看有没有相应的 DDR 配置文件及 Nand 配置文件 (从 Nand 启动时), 如果没有，参照已有文件新建一个，查询 DDR 手册，配置好相关的参数

5. 在 include/configs 下，从相似的板级文件拷贝一个，重命名为 h700.h, 一般需做以下修改:

   - 修改所包含的 DDR 头文件及 Nand 头文件 (从 Nand 启动时)
   - 修改 CONFIG_BOOTARGS, 主要需要修改内存大小以及所使用的串口
   - 找到 “defined(CONFIG_MSC_X_BOOT)” 处，修改分区大小及分区偏移
   - 在文件末尾，有很多 GPIO 相关的宏，修改他们，使之与板子对应

由于代码混乱，详细列出所有步骤有点难度，请以代码为准。

Part 2

Linux 基础入门

Chapter 2

关于 Linux 入门

2.1 破釜沉舟: 最好把 windows 卸载掉

学习 Linux 最好的方法就是天天用它，天天用它最好的方法是把你的 windows 卸载掉，只要你的 windows 还在，你就依然会时不时地依恋她。

Linux 有很多发行版，在`http://distrowatch.com/`上可以看到各个发行版的流行度，在进行 Android 开发时，官方推荐使用 Ubuntu，但现在 Ubuntu 越来越难用了，因此选用 Mint, Debian 或是 Gentoo 也许是不错的选择，也许在这些发行版上搭建环境可能会遇到这样那样的问题，但大家都是 Linux，既然 Ubuntu 可以，理论上大家都可以，只是有可能会稍微麻烦一点。

2.1.1 关于发行版

Gentoo 是一个很有意思的版本，他只比 LFS 高级一点点，刚安装时只是一个最小系统，所有软件安装都是下载源码编译安装 (始终是最新的软件)，这点和 ubuntu 直接安装二进制包不同。这样能带来高度的定制性。

Arch Linux 是比 Gentoo 高级一点点的发行版，它不从源码编译，直接安装二进制包，但软件更新速度比 ubuntu 快，始终是最新版的软件。

Gentoo 和 Arch Linux 都不适合入门者使用，她们相对 Ubuntu 来说不是那么稳定，Ubuntu 不追求最新的软件，而且大部分软件经过了仔细的测试。

也许 Mint 比 Ubuntu 更傻瓜，Mint 上集成了更多的常用软件，这导致它的安装盘必须是 DVD，一张 CD 容不下.

选择 Mint 的原因可能更多的是 Ubuntu 最新版的界面越来越难用了，而 Mint 的 MATE 界面从经典的 GNOME 2 衍生，可操作性及用户友好性上更好。

Ubuntu 的失败在于它是由一个自私的公司开发，从历史来看，在 Linux 世界里，但凡是夹杂着公司的行为的，除了服务器版本，大多桌面版本都以失败告终。Linux 的力量来自社区的合作，这些人互不相识，但却有共同的目标，没有人成为主导，所有人只是为了目标而相互合作。

这就像人类的历史一样，一旦走上独裁，离失败也就不远了，可惜的是几乎所有人都希望成为主导，殊不知只有互相合作才是生存之道，也许世界上只有 Linux 社区这一方净土。

个人建议: 如果追求稳定，安装 ubuntu 10.04，如果既要稳定又要最新，安装 Ubuntu 12.04(或即将发布的 12.10)，如果追求性能及最新软件，安装 Gentoo 或 Arch。

2.2 请试试 LFS

OK，现在你有一个发行版了，你能进行日常的工作，但你仍然认为 Linux 很神秘，你想知道你的系统上都有什么文件，你想知道一个发行版是如何制造出来的，尝试一下 LFS/BLFS，她就像一个天文学家，向你讲述 Linx 宇宙大爆炸之后产生的各个碎片是如何形成今天的宇宙的。

LFS 网址: `http://www.linuxfromscratch.org/lfs/`。

LFS 只是一个基本的系统，只有一个命令行，更多的软件在 BLFS 中说明，其网址是: `http://www.linuxfromscratch.org/blfs/`

在学习 LFS 的过程中，我们的目标就是知其然，至于其所以然，是一个长期使用积累的过程，因此，学习 LFS 的方法只需要注意两点:

- 一个字不差地看
- 不明白的地方先跳过，照着做就行了

2.3 Linux 的世界观

有时间可以阅读 Richard Stallman 的一些演讲，还有这篇文章: `http://www.catb.org/esr/faqs/smart-questions.html`

2.4 Linux 资料

Linux 的世界很大，初学者很容易迷茫，这里送上几句个人建议，仅供参考:

- 尽量不要看中文书籍，你会觉得中文更容易懂，但实际上并非如此，很多翻译者都只是英语专业里 linux 最好的人。
- 看 Linux 相关的书时，建议先看个大概，然后 Read the fucking code，实在理解不了代码再去看书。这样做的原因是即便是 “最新” 的书，往往讲述的都是旧版本的内核，计划往往赶不上变化。
- 学习 Linux 最好的方法是先看个例子，然后再举一反三，再以这个例子为线索去挖掘例子背后的故事
- 不要花时间学你最近两个月都用不到的东西，哪怕你非常好奇，记住 “消费者导向”*o*
- 先扫一遍，用的时候再临时抱佛脚也未尝不是件好事
- 积累，积累，积累

2.4.1 Linux 系统使用

入门

所谓入门即掌握常用的系统管理命令，不要去买什么《Linux 入门教程》之类的书浪费钱，看几篇入门文章就 OK 了，不用去看那些长篇大论浪费时间。

进阶

入门了之后，作为一个 Linux 程序员，还需要掌握很多工具的使用, 有一本好书推荐给大家: <Unix Power Tools, Third Edition>，这是一本大部头的书，但是由很多小 tips 组成的，不需要像平常的书那样通读，作为工具书去看，或是在无聊时翻一翻。

我们还需要掌握 bash, 最好掌握 sed&awk，在 python 和 perl 之间二选一或者都学。

对于 bash，可以看几篇入门类的文章，能写个 Hello World, 会 if, else 之后，就去看 Advanced Bash Script(`http://tldp.org/LDP/abs/abs-guide.pdf`), 打算好好学习 bash 的话，把这本书整个扫一遍，知其然即可，在脑子里留下个印象，以后用到的时候再来查。

对于 sed&awk，不建议去网上看入门文章，直接看《sed & awk, 2nd Edition》即可，为了更好地应用 sed 和 awk，需要掌握正则表达式，可以看几篇入门的文章，打算好好学习的话，可以阅读 <Mastering Regular Expressions>。

至于 python 和 perl，书实在太多，Learning Python 和 Learning Perl 都是很好的书，适合入门。

2.4.2 内核学习

经典的书籍有:

《Understanding the Linux Kernel》:

《深入理解 LINUX 内核》(第 3 版) 指导你对内核中使用的最重要的数据结构、算法和程序设计诀窍进行一次遍历。通过对表面特性的探究，作者给那些想知道自己机器工作原理的人提供了颇有价值的见解。书中讨论了 Intel 特有的重要性质。相关的代码片段被逐行剖析。然而，《深入理解 LINUX 内核》(第 3 版) 涵盖的不仅仅是代码的功能，它解释了 Linux 以自己的方式工作的理论基础。

《Professional Linux Kernel Architecture》:

众所周知，Linux 操作系统的源代码复杂、文档少，对程序员的要求高，要想看懂这些代码并不是一件容易事。本书结合内核版本 2.6.24 源代码中最关键的部分，深入讨论 Linux 内核的概念、结构和实现。具体包括进程管理和调度、虚拟内存、进程间通信、设备驱动程序、虚拟文件系统、网络、时间管理、数据同步等方面的内容。本书引导你阅读内核源代码，熟悉 Linux 所有的内在工作机理，充分展现 Linux 系统的魅力。　　本书适合 Linux 的系统编程人员、系统管理者以及 Linux 爱好者学习使用。

《Linux Device Driver》:

《LINUX 设备驱动程序 (第 3 版)》已针对 Linux 内核的 2.6.10 版本彻底更新过了。内核的这个版本针对常见任务完成了合理化设计及相应的简化，如即插即用、利用 sysfs 文件系统和用户空间交互，以及标准总线上的多设备管理等等。要阅读并理解本书，您不必首先成为内核黑客；只要您理解 C 语言并具有 Unix 系统调用的一些背景知识即可。您将学到如何为字符设备、块设备和网络接口编写驱动程序。为此，《LINUX 设备驱动程序 (第 3 版)》提供了完整的示例程序，您不需要特殊的硬件即可编译和运行这些示例程序。《LINUX 设备驱动程序 (第 3 版)》还在单独的章节中讲述了 PCI、USB 和 tty(终端) 子系统。对期望了解操作系统内部工作原理的读者来讲，《LINUX 设备驱动程序 (第 3 版)》也深入阐述了地址空间、异步事件以及 I/O 等方面的内容。

《Linux 设备驱动开发详解》:

这是一本国人写的一本不错的书，符合国情，适合入门，虽然其中的内容和网上的一些资料有些雷同，但看得出作者真的花了不少功夫，个人认为就入门来说比 Linux Device Driver 更好，只是内容的深度稍有欠缺。

《Linux 设备驱动开发详解 (第 2 版)》是一本介绍 Linux 设备驱动开发理论、框架与实例的书，《Linux 设备驱动开发详解 (第 2 版)》基于 LDD6410 开发板，以 Linux2.6 版本内核为蓝本，详细介绍自旋锁、信号量、完成量、中断顶/底半部、定时器、内存和 I/O 映射以及异步通知、阻塞 I/O、非阻塞 I/O 等 Linux 设备驱动理论；字符设备、块设备、TTY 设备、I2C 设备、LCD 设备、音频设备、USB 设备、网络设备、PCI 设备等 Linux 设备驱动的架构和框架中各个复杂数据架构和函数的关系，并讲解了 Linux 驱动开发的大量实例，使读者能够独立

开发各类 Linux 设备驱动。《Linux 设备驱动开发详解 (第 2 版)》内容全面，实例丰富，操作性强，语言通俗易懂，适合广大 Linux 开发人员、嵌入式工程师参考使用。

《Understanding the Linux Virtual Memory Manager》：

买不到纸质的了，可以从`www.kernel.org/doc/gorman/pdf/understand.pdf`下载到。

是一本介绍 Linux 虚拟内存管理机制的书。如果你希望深入的研究 Linux 的内存管理子系统，仔细的研读这本书无疑是最好的选择。虽然代码介绍的是 Linux 2.4，但对于理解 linux 2.6 的内存管理也有很大的帮助。

《Understanding Linux Network Internals》：

一本讲解网络子系统实现的书，通过这本书，我们可以了解到 Linux 内核是如何实现复杂的网络功能的。

2.4.3 常用网站

- 内核官网: `http://kernel.org`
- 订阅邮件列表，所有的内核相关的邮件列表可以在`http://vger.kernel.org/vger-lists.html`找到，常用的邮件列表有 linux-kernel, linux-next, linux-mmc, linux-input, linux-usb, linux-pm 等。订阅了 linux-kernel 之后，一天有几百封邮件，请三思而后行。
- Linux Weekly News: `http://lwn.net/`
- Linux 网址导航: `http://linux.ubuntu.org.cn/`

2.5 编程工具选择

2.5.1 Vim 和 Emacs

在 Linux 上主要的编程工具是 vim 和 Emacs，很难说哪个更好。

个人认为 vim 比较适合编辑单个文件或做简单的编辑，Emacs 比较适合同时编辑多个文件 (一个工程)，之所以这么认为，是因为个人觉得为了简单地编辑单个文件而去 “劳烦”Emacs 是不值当的，而 Vim 在处理多个文件时显得比较笨拙，Emacs 熟练了之后，相对来说 Vim 的操作其实更复杂。建议最好两种工具都学习，一个为主，一个为辅。

Cscope, GNU Global, ctags, etags

浏览代码时，在代码间跳转是很常用的，使用 Cscope 等插件，我们能把 Vim 或 Emacs 打造成比 Source Insight 还强大的代码浏览工具。

- cscope: 适用于 C/C++，在 C/C++ 方面比 global 好
- GNU Global: 通吃 C/C++/JAVA 等，在 Emacs 里进行 Android 代码学习离不开他
- ctags: 不熟
- etags: 不熟

2.5.2 Source Insight

大部分人仍然倾向于 Source Insight，如果使用 Source Insight 编辑代码，请注意换行符，Linux 的换行符和 Windows 是不一样的。

2.5.3 Visual Studio

最好别用，会水土不服的。

Chapter 3

初识 linux 内核

“学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于内核，而又不高于内核的。内核源码本身就是最好的参考资料，其他任何经典或非经典的书最多只是起到个辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。”

“Read the fucking source code” ——Linus Torvalds

3.1 内核代码树介绍

3.1.1 目录结构

目录	描述
arch	特定于 CPU 架构的源代码, 其中 arch/mips 是 MIPS 架构相关的代码
block	块 I/O 层
crypto	加密 API
Documentation	内核源代码文档
drivers	设备驱动
firmware	设备固件集合地
fs	VFS 和各种文件系统 (fat, ext2, ext3, ext4)
include	内核头文件
init	内核启动和初始化
ipc	System V IPC 实现
kernel	核心子系统，如调度器等
lib	包含很多 Helper Functions, 供系统其他部分调用
mm	内存管理子系统
net	网络子系统
samples	示例，示范代码
scripts	用于生成内核的脚本
security	Linux 安全模块
sound	音频子系统
usr	早期的用户空间代码（即 initramfs）
tools	辅助 Linux 开发的一些小工具
virt	虚拟化基础设施

3.1.2 小窍门

- 内核的代码虽多, 并不是所有的都和我们相关, 在使用 source insight 或其他工具查看代码时, 只加入下面这些目录中的代码即可

arch/mips	block	crypto	fs	include
init	ipc	kernel	lib	mm
net	net	security	sound	tools
usr	drivers/base	drivers/mmc	drivers/i2c	drivers/media/video
drivers/misc	drivers/usb	drivers/net	drivers/power	drivers/input
drivers/gpu	drivers/staging/android			

建议在阅读内核代码前先阅读内核 Documentation 目录下的 CodingStyle。

- 内核中大量使用宏来生成一系列函数或变量，如果直接查找这些函数或变量，一般是查找不到的，此时可使用以下方法 (以查找 set_c0_cause 的定义为例):

  1. touch 一下引用了这些函数或变量的.c 文件 (这里是 traps.c)

     ```
     $ touch arch/mips/kernel/traps/c
     ```

  2. 使用如下命令编译内核

     ```
     $ make V=1 zImage | tee log
     ```

  3. 查看 log 文件，查找 xxx.c，你会看到一条以 xxx-gcc 开头的编译命令, 例如:

     ```
     mips-linux-gnu-gcc -Wp,-MD,arch/mips/kernel/.traps.o.d ..... arch/mips/kernel/traps.c
     ```

  4. 将编译命令修改一下，添加 -save-temps:

     ```
     mips-linux-gnu-gcc -save-temps -Wp,-MD,arch/mips/kernel/.traps.o.d ..... arch/mips/kernel/traps.c
     ```

     编译完成后会得到 traps.i 和 traps.s 两个文件，前者是 preprocess 后的文件，后者是汇编的中间文件

  5. 查找 set_c0_status，结果如下:

     ```
     # 1602 "/home/lutts/android/disk1/android-4.0-bbk/kernel/arch/mips/include/asm/mipsregs.h"
     static inline __attribute__((always_inline)) unsigned int set_c0_status(unsigned int set)
     ```

     由此可见 set_c0_status 定义在 arch/mips/include/asm/mipsregs.h 中。

  这个 V=1 然后 save-temp 大法适于大多数找不到定义的情况。

- 有时通过 -save-temps 方式仍然找不到函数定义的地方，这种定义往往是由于汇编 macro 生成的，使用 -save-temps 是不会展开汇编的 macro 的，此时有一个办法就是 grep 所有的.o 文件，以查找 handle_fpe 为例, 在内核根目录下执行以下命令:

  ```
  $ find . -name '*.o' -print | xargs grep 'handle_fpe'
  Binary file ./arch/mips/built-in.o matches
  Binary file ./arch/mips/kernel/built-in.o matches
  Binary file ./arch/mips/kernel/genex.o matches
  Binary file ./arch/mips/kernel/traps.o matches
  Binary file ./vmlinux.o matches
  ```

  此时会发现 handle_fpe 在 genex.o 和 traps.o 中出现了 (build-in.o 和 vmlinux.o 是特殊的中间文件，不和.c 对应，因此排除在外)，因此接下来我们 objdump 这两个.o 文件:

  ```
  $ mips-linux-gnu-objdump -d genex.o >genex.dump
  $ mips-linux-gnu-objdump -d traps.o >traps.dump
  ```

  查看 dump 文件，看哪里有 handle_fpe，在这个例子里是 genex.o，也就是说，handle_fpe 是定义在 genex.S 中的，在 genex.S 中，有以下代码:

```
.macro  __BUILD_HANDLER exception handler clear verbose ext
.align  5
NESTED(handle_\exception, PT_SIZE, sp)
.set    noat
SAVE_ALL
FEXPORT(handle_\exception\ext)
__BUILD_clear_\clear
.set    at
__BUILD_\verbose \exception
move    a0, sp
PTR_LA  ra, ret_from_exception
j       do_\handler
END(handle_\exception)
.endm
.macro  BUILD_HANDLER exception handler clear verbose
__BUILD_HANDLER \exception \handler \clear \verbose _int
.endm
...
BUILD_HANDLER fpe fpe fpe silent                /* #15 */
```

由此我们可以知道 handle_fpe 是通过 BUILD_HANDLER 定义的，其最终调用了 do_fpe，这个方法在 traps.c 中。

使用以下脚本可以将上述过程自动化:

```
#!/bin/bash

OBJ_FILES=‘find . -name ’*.o’ -print | xargs grep -l ”$1”‘

for file in $OBJ_FILES; do
        if echo $file | egrep ”(built-in|vmlinux)” >/dev/null 2>&1; ↩
            then
                continue;
        fi
        mips-linux-gnu-objdump -d $file >temp.dump
        if grep ”<$1>:” temp.dump >/dev/null 2>&1; then
                echo ”Found in $file”
        fi
done

rm temp.dump
```

3.1.3 内核编译及 linux build system

学习一个系统，最关键的着手点是理解其编译系统，其就像是一张地图一样，为您在陌生的世界里指明方向。理解 linux 的编译过程，会使得您从大局上对系统有更深入的理解，更能为以后分析代码带来方便，

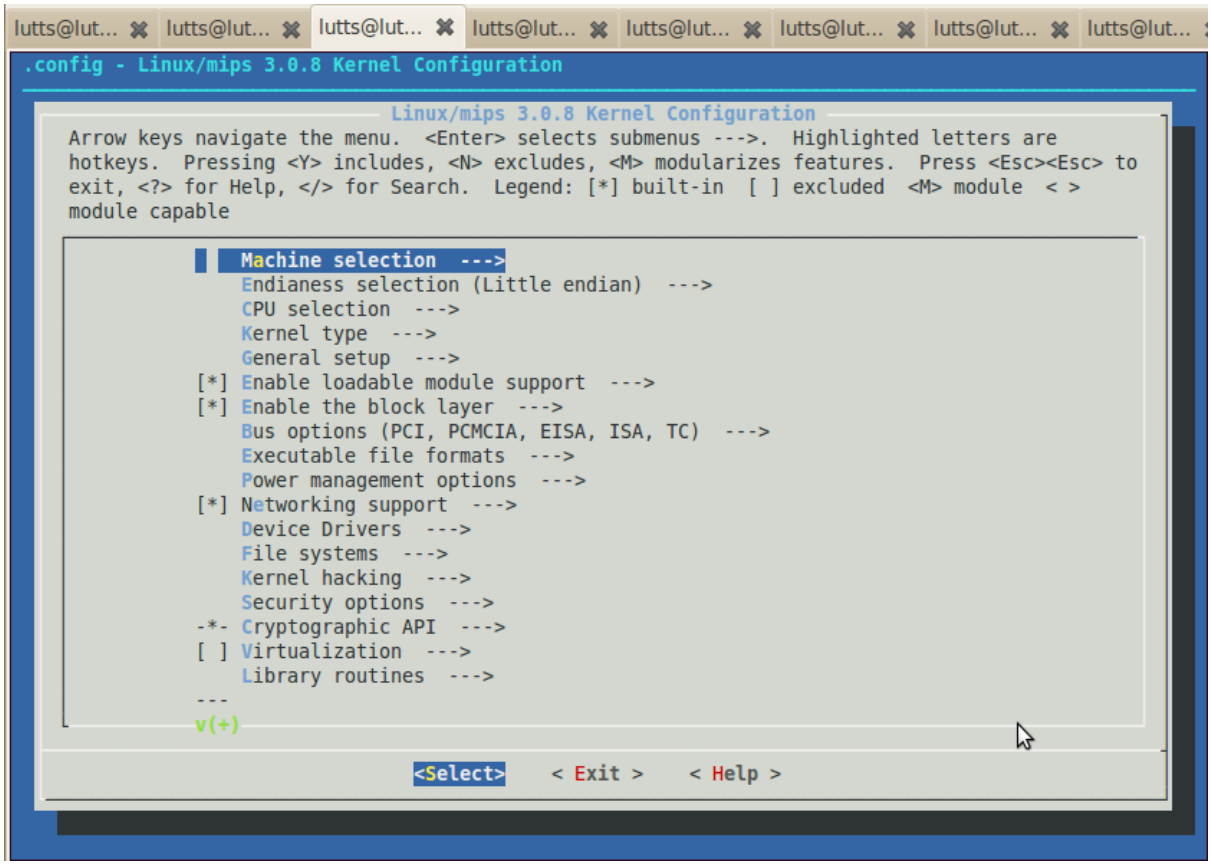
先来看一下 h700 的内核编译过程，在内核的根目录下执行以下命令:

```
$ make h700_defconfig
$ make
```

第一步 make h700_defconfig 用于在编译前配置内核，linux 内核在编译之前都需要先配置，以决定哪些模块，哪些代码需要编译进内核；配置完成后会在内核根目录下生成一个隐藏的.config 文件, 还会在 include/generated/(或 include/linux/autoconf.h) 目录下生成一些头文件。

第二步开始编译，可以指定 -j 参数进行多进程编译，比如 make -j4 会使用 4 个进程编译，根据经验 <cpu 核数 >+1—2 * <cpu 核数 > 是比较合适的值，但也需要考虑其他人, 因为一旦马力全开, 机器会很慢的。

还有一个常用的命令是 make menuconfig，这个命令会弹出一个类似下图的界面，提供手动配置内核的功能:



注: 喜欢图形界面的可以使用 make xconfig。

其他常用的命令有:

- make clean: 清除编译过程中生成的中间文件，但会保留.config 文件
- make mrproper: 使内核树回到 “干净的” 状态，任何中间文件，配置文件等都会删除，回到刚从库里 check 出来时的状态 (实际上没这么干净)。

  注: 使用 git 管理时，使用 “git clean -xfd .” 能真正回到刚 check 时的状态。
- make zImage: 这个命令会重新生成 vmlinux 及 zImage，zImage 是 boot.img 的一部分，有时你只是想编译一下内核看能不能编，并不需要生成 boot.img。
- make modules: 生成 config 为 m 的动态模块 (ko)

linux 内核配置与编译的核心是一个名叫 kbuild 的系统，详细的文档可以在内核根目录的 Documentation/kbuild 目录下找到。这里稍作讲解，并以一个实例讲解 Kconfig 和 Makefile 在分析代码时的作用。

linux kernel 的主 Makefile 位于源码的根目录下，感兴趣的可以耐着性子详细地分析内核 Makefile 的实现，这可是不可多得的学习 makefile 的好例子。

内核中的各个模块都有自己的 Makefile，还有一个 Kconfig 和其搭档。Kconfig 和 Makefile 可以看做是内核地图，在分析内核源码的过程中，往往需要借助他们来找到所对应的.c 文件

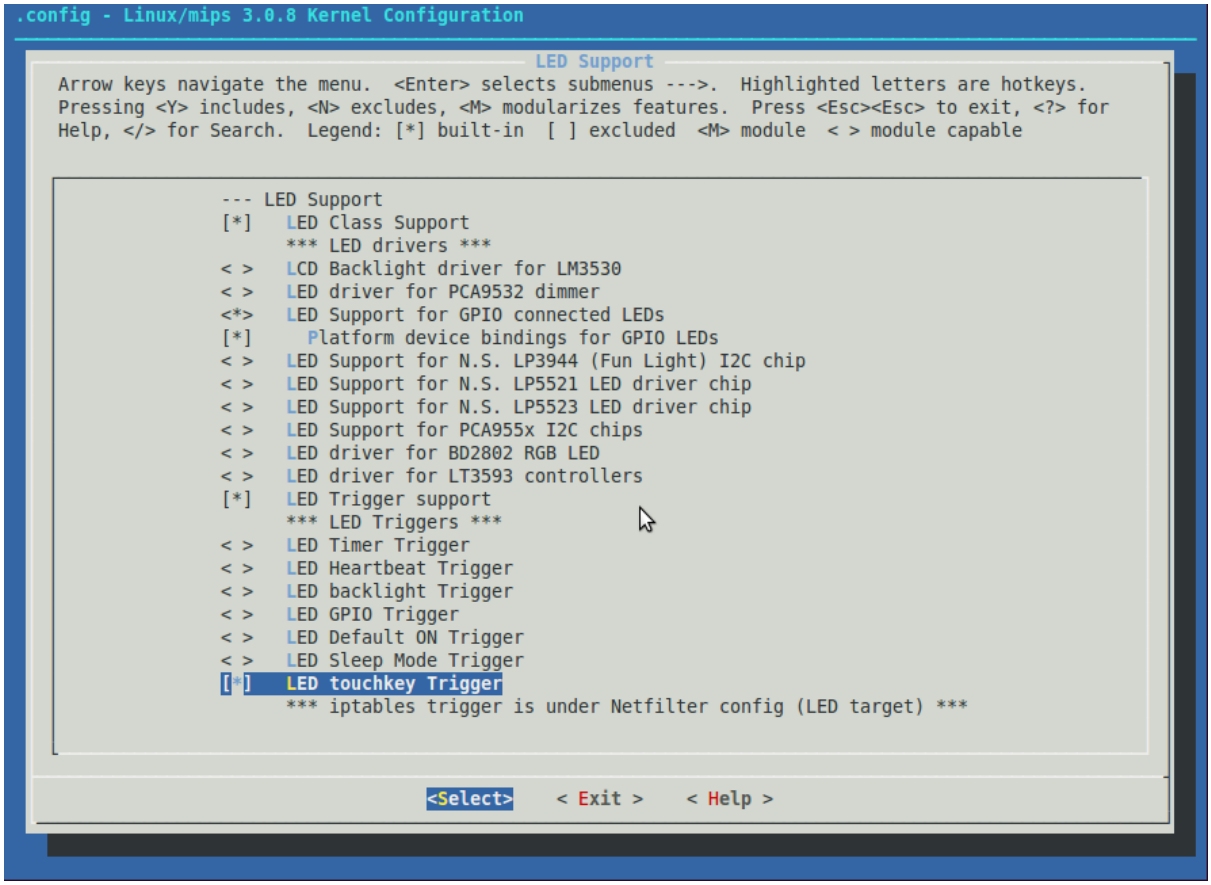
使用 Kconfig 和 Makefile 分析源码

分析代码前先分析相关的 Kconfig 和 Makefile, 能增加对代码理解的一种大局观, 还有很多时候，你会碰到如下情况:

1. 某某某告诉你为了完成这个功能需要在 config 里选择某个选项，除了这个, 没有说明其他信息
2. 在分析代码的过程中, 有多个.c 实现了同样的功能, 您需要确定现在编译的是哪个

这时候, 也需要借助 Kconfig 和 Makefile 来理清代码的结构。

以分析 linux LED 子系统为例, 来看一下目前我们所选择的项:



通过查看 Help，图中标 * 的选项对应的 CONFIG 如下:

- LED Support 本身 —> CONFIG_NEW_LEDS
- LED Class Support —> CONFIG_LEDS_CLASS

- LED Support for GPIO connected LEDs —> CONFIG_LEDS_GPIO
- Platform device bindings for GPIO LEDs —> CONFIG_LEDS_GPIO_PLATFORM
- LED Trigger support —> CONFIG_LEDS_TRIGGERS
- LED touchkey Trigger —> CONFIG_LEDS_TRIGGER_TOUCHKEY

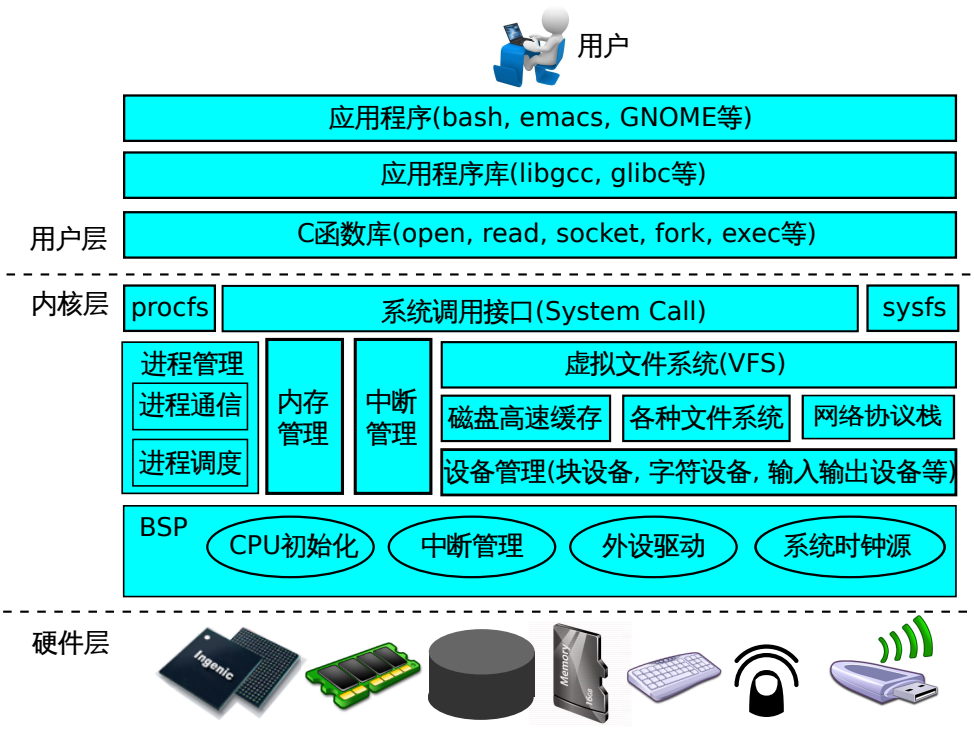
通过查看 Help，我们还可以知道这些选项位于 drivers/leds/Kconfig 中，我们进入 drivers/leds 目录，打开 Makefile 文件，查找上面那些 CONFIG 宏，可以查找出所涉及到的源文件如下:

- CONFIG_NEW_LEDS —> led-core.o —> led-core.c
- CONFIG_LEDS_CLASS —> led-class.o —> led-class.c
- CONFIG_LEDS_GPIO —> leds-gpio.o —> leds-gpio.c
- CONFIG_LEDS_GPIO_PLATFORM —> 没有对应的源文件，一般情况下表明其是用来选择编译某些函数宏。
- CONFIG_LEDS_TRIGGERS —> led-triggers.o —> led-triggers.c
- CONFIG_LEDS_TRIGGER_TOUCHKEY —> ledtrig-touchkey.o —> ledtrig-touchkey.c

由此我们就确定了增加 LED 支持所涉及到的源文件。

3.2 linux 系统架构

下图是 linux 的系统架构:



3.2.1 用户态与内核态的区别与联系

整个 **linux** 软件系统分为两层

- 用户态
- 内核态

对于这两者的区别，大致可归结为以下几点

- 最简单的解释就是: 内核运行在内核态，普通的用户程序运行在用户态.
- 从 MIPS CPU 的角度来看，用户态和内核态分别对应 MIPS 的两个 privilege level，用户态的程序不能执行某些指令, 内核态什么都可以做.
- 用户态为应用程序的执行引入了一个沙盒 (sandbox), 一个应用程序不可以直接访问另一个应用程序的内存及资源, 这样在一个应用程序崩溃或是干某些坏事的时候,(理论上) 就不会影响系统中的其他程序.

两者间的联系，即用户态与内核态的接口，大致可以归结为以下几个方面

- 用户态程序 (正常情况下) 通过 syscall 指令请求内核态完成相应的功能。用户态程序很少 “亲自” 执行 syscall 指令，C 函数库封装了内核的 syscall 调用，用户态通过调用 C 函数库的 API 接口执行特定的操作。

  每个 syscall 调用都有一个调用号，用户态程序看到的是 open、read 这样函数调用，C 函数库将他们 “翻译” 成系统调用号, 将函数参数进行适当的排放, 然后执行 syscall 指令, syscall 指令会触发一个 CPU 异常, 内核中的处理函数是 handle_sys, 它根据调用号将用户态的请求分发给适当的处理程序。

  虽然理论上可以自由添加新的系统调用, 但一般都不建议这样做, linux 现有的系统调用由 POSIX 标准进行了定义。

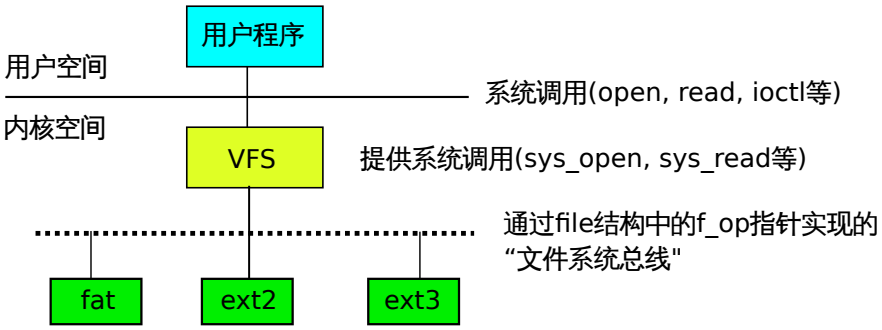
- procfs 和 sysfs，这是两个基于内存的虚拟文件系统，用于内核向用户态展示信息，同时允许用户态通过 procfs 和 sysfs 动态改变内核的行为，我们将在内核调试相关的章节详细描述 procfs，在 linux 设备驱动相关的章节描述 sysfs.

  注: procfs 被称为 “已是过去式但仍未被遗忘”，不建议再向 procfs 添加信息，应该转为使用 sysfs

3.2.2 内核核心组件

整个系统核心由以下部分组成：

1. 系统调用接口: 用户态与内核态的接口层
2. 虚拟文件系统 (VFS): 位于物理文件系统 (fat, ext2, ext3 等) 和应用层之间的一个抽象层，对用户层提供统一的系统调用接口，其结构如下:



3. 进程控制 (Process Management): 进程的调度、同步和通信
4. 中断管理 (Interrupt Management): 系统中断管理，在 linux 中，进程调度，时钟，定时器等都依赖中断
5. 内存管理 (Memory Management): 物理内存管理，虚拟内存管理等
6. 网络协议栈 (Networking)：实现各式各样的网络协议。
7. 模块管理 (Module Management): 几乎所有的设备驱动都以模块的形式出现在内核中
8. 设备管理 (Device Driver): 块设备驱动 (随机存取设备)、原始设备 (raw 设备，字符设备，裸设备)

从用户的角度来看，“文件”和“进程”是 Linux 内核中的两个最基本、最核心的概念，Linux 系统的所有操作都是以这两者为基础的。之所以说文件在 linux 中是一个最基本的概念，是因为不论是设备管理还是网络协议栈, 在用户态都表现为一个文件, 在 linux 中, ”everything is file”。一般来说，linux 系统中包括以下文件类型:

- 普通文件: 即我们经常看到的一个个文件
- 文件夹
- 设备文件，他们一般位于/dev 目录下
- /proc, sysfs 文件
- 管道文件

3.2.3 BSP(Board Support Package)

BSP 是嵌入式系统中介于硬件平台和系统核心之间的中间层软件, 主要目的是为了屏蔽底层硬件的多样性, 根据操作系统的要求完成对硬件的直接操作, 向操作系统提供底层硬件信息。BSP 具有硬件相关性和操作系统相关性的特点, 其主要作用包括:

- 初始化 CPU、内存、中断等相关的寄存器，对协处理器 (如 FPU) 进行正确的配置
- 初始化底层硬件, 为操作系统提供底层硬件信息

后面我们会安排专门的章节来详细讲述君正 BSP 的实现。

3.3 需要掌握的基本概念 –Kernel Infrastructure

3.3.1 基本数据类型

在内核中, 可以使用 C 语言标准的数据类型, 如 int, unsigned int, long long int 等，但使用这些类型可能会带来移植的问题，因为在不同的 CPU 架构上，这些类型的大小及对齐方式可能是不一样的，为了解决这个问题，内核为我们定义了一些“架构无关”的数据类型:

- 用户态: __s8, __u8, __s16, __u16, __s32, __u32, __s64, __u64
- 内核态: 除了用户态的类型外，还有 s8, u8, s16, u16, s32, u32, s64, u64，建议在内核态只使用不带双下划线 (”__”) 的.

相关的头文件是 kernel/include/asm-generic/int-ll64.h

需要注意的是, linux 内核不支持浮点, 如果要使用浮点, 最好使用除法来表示, 使用除法在精度上可能无法达到要求, 实际使用时要注意。

还需要注意的是在我们的 32 位内核中对 64 位的乘除法没有内建的支持, 如果要使用 64 的乘除法, 需要通过一些变通的方式。

内核中还有另外一个重要的数据类型是链表, 包括 list, klist, hlist(哈希链表), 在内核中大量使用他们, 一定要熟练掌握他们的使用.

内核中有一个 container_of 宏，一定要理解其实现并熟练使用!

因为内核不依赖于 libc，因此在内核里不能调用 libc 中的函数，因此，内核开发者在 lib 目录下为我们提供了一些等价的实现, 如 strtoul 在内核里的实现是 simple_strtoul，实现位于 lib/vsprintf.c 中，在这个文件里还实现了 sprintf, snprintf 等 “著名” 的操作, 一些操作字符串的函数如 strchr 等在 lib/string.c 中也有等价的实现，有兴趣不仿多熟悉 lib 下代码。

内核开发者为我们提供了很多 “车轮”，初学者一般都会重复造车轮，因此有时间的话一定要多读内核的代码，拓宽对内核的理解。

3.3.2 Time，Delays，and Deferred Work

Time

内核使用 jiffies 来保存自开机以来系统的时间, jiffies 是一个相对的概念, 在配置内核时可以指定一秒包含多少个 jiffies(由 CONFIG_HZ 指定), jiffies 越大, 理论上时间精度越高, 但不是越大越好, 精度大可能会带来其他的系统开销. 目前我们的 kernel 中一秒包含 250 个 jiffies(CONFIG_HZ=250), 也就是说, 一个 jiffie 相当于 4ms 的时间.

围绕着 jiffies, 内核提供了大量的辅助函数, 可以参见 include/linux/jiffies.h 头文件.

关于 jiffies 有一点需要注意: 在 CPU 中断关闭的情况下, jiffies 是不会增加的, 也就是说, 在 CPU 中断关闭的情况下, 不能依赖于 jiffies 来进行判断, 否则会限入死循环, 因为你要的条件永远也等不到.

Delays

内核提供了两类延时函数, 一类是忙等, 一类会让出 cpu. 忙等的有:

1. ndelay(unsigned long nsec): 纳秒
2. udelay(unsigned long usec): 微秒

3. mdelay(unsigned long msec): 毫秒

要注意, 内核没有提供秒级的等待函数, 要等待秒级的时间时, 可以给 mdelay 传一个较大的参数, 比如 5 秒就是 5*1000=5000ms

会让出 cpu 的等待有:

1. msleep(unsigned int millisecs): 毫秒, 不可中断, 也就是说一定会睡 (至少)millisecs 长的时间
2. msleep_interruptible(unsigned int millisecs): 毫秒, 可中断, 也就是说如果有信号发往这个进程, 即便还没睡够, 也会醒来.
3. ssleep(unsigned int seconds): 秒

内核中还提供等待某个事件的机制, 所谓的事件可能是某个变量由 0 变 1 之类的, 相关的方法有:

```
void init_waitqueue_head (wait_queue_head_t *q);
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);
void wake_up(wait_queue_head_t *q)
void wake_up_interruptible(wait_queue_head_t *q)
void wake_up_interruptible_all(wait_queue_head_t *q)
```

Deferred work

在 Linux 系统中，中断的处理分为两部分，分别称为上半部 (Top Half) 和下半部 (Bottom Half)。在硬中断的处理中，应该尽可能快，只做必要的工作，否则很容易导致中断事件的丢失。费时的任务都由中断下半部完成。

内核提供了很多机制用于下半部的调度, 目前的机制有:

1. 软中断: 它是所有后半部机制的基础, 中断下半部的实现基于软中断机制，系统中对时间要求很严格 (如网络及块设备) 时会直接使用软中断，目前系统中的软中断有:

```
/* include/linux/interupt.h */
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};
```

这里的定义顺序决定了相应软中断的优先级，软中断的实现位于 kernel/softirq.c 中，实现机制类似于 MIPS CPU 对中断向量表，核心元素包括:

(a) 软中断状态寄存器 (irq_cpustat_t irq_stat[NR_CPUS] ____cacheline_aligned;)

(b) 软中断向量表 (static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_-smp;)

(c) 软中断守护进程 (ksoftirqd)

如果 irq_stat 的值是 0001000010，则 TIMER_SOFTIRQ 和 TASKLET_SOFTIRQ 置 1，表示对应的 softirq 需要处理，调用软中断向量表里相应的 action。

熟悉 MIPS 中断处理的对这个可能有种似曾相识的感觉。

软中断是一种很底层的机制，不到万不得已我们不用添加自己的 softirq，事实上，七八年了，softirq 都一直是老样子。

2. tasklet: 也是基于软中断实现, 这个不常用, 在网络协议栈中用得较多, tasklet 也运行在软中断上下文, 因此也不可以 sleep, tasklet 只能在提交它的 CPU 上运行，同一个 tasklet 在同一时刻只能在一个 cpu 上运行，因此相对于其他 softirq 而言，无需考虑并行处理所带来的线程安全问题。

3. 内核定时器 (struct timer_list): 基于软中断实现，在指定的一段时间过后执行某项工作, 内核定时器运行在软中断上下文, 因此执行工作时不可以 sleep。内核定时器也只在提交它的 CPU 上运行。

4. 工作队列: 有非 delay 和 delay 的工作队列,

   - 非 delay 的工作队列: 在稍后 (时间不确定) 执行某项工作
   - delay 的工作队列: 在延迟一段时间后, 再稍后 (时间不确定) 执行某项工作

   工作队列和内核定时器的一个显著的区别就是: 工作队列具有随意性, 而定时器具体强制性. 在新的 Linux 内核里，工作队列其实由每 CPU 一个的内核线程管理，运行在进程上下文中，因此可以休眠，但也因为其依赖进程调度器，何时运行具有不确定性，实时性不能得到保证。

   在多核系统上，每个 CPU 都有一个工作队列，工作任务也只在提交它的 CPU 上运行。

TODO: 通过阅读代码进一步理解这三种机制和 CPU 的亲和性及并发特性。

这三种 Defer work 的方式需要很好地掌握他们的用法以及他们的区别.

Deferred Work 机制一般用于中断下半部的处理，我们来大致了解一下 linux 的中断处理流程，我相信这将帮助大家更好地理解 Deferred work 的处理时机:

中断产生时，由中断向量表里的 handle_int 处理，其实现位于 arch/mips/kernel/genex.S 中 (下面的代码中去掉了和我们 CPU 无关的部分):

```
NESTED(handle_int, PT_SIZE, sp)
        SAVE_ALL
        CLI
        TRACE_IRQS_OFF

        LONG_L  s0, TI_REGS($28)
        LONG_S  sp, TI_REGS($28)
        PTR_LA  ra, ret_from_irq
        PTR_LA  v0, plat_irq_dispatch
        jr      v0
        END(handle_int)
```

给 ra 赋值为 ret_from_irq，这样当 plat_irq_dispatch 返回时，将调用 ret_from_irq, plat_irq_dispatch 与具体的 CPU 相关，决定具体的 IRQ(如 gpio 中断), 最后调用 do_IRQ, 位于 arch/mips/kernel/irq.c 中:

```
void __irq_entry do_IRQ(unsigned int irq)
{
        irq_enter();
        check_stack_overflow();
        if (!smtc_handle_on_other_cpu(irq))
                generic_handle_irq(irq);
        irq_exit();
}
```

其中的 generic_handle_irq 将最终调用我们的中断处理程序，在 irq_exit() 之前就可以理解为中断的上半部，这一部分因为整个 CPU 的中断是禁止的，因此要求处理速度尽可能地快，以免影响系统响应，大部分工作都在 irq_exit() 中进行处理:

```
void irq_exit(void)
{
        account_system_vtime(current);
        trace_hardirq_exit();
        sub_preempt_count(IRQ_EXIT_OFFSET);
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();

        rcu_irq_exit();
#ifdef CONFIG_NO_HZ
        /* Make sure that timer wheel updates are propagated */
        if (idle_cpu(smp_processor_id()) && !in_interrupt() && !need_resched())
                tick_nohz_stop_sched_tick(0);
#endif
        preempt_enable_no_resched();
}
```

在这里处理了软中断 (invoke_softirq，真正的处理在 do_softirq 中)，然后启用了抢占，但并未启动调度器, do_IRQ 返回时将返回到 ret_from_irq:

```
FEXPORT(ret_from_irq)
        LONG_S  s0, TI_REGS($28)
FEXPORT(__ret_from_irq)
        LONG_L  t0, PT_STATUS(sp)               # returning to kernel mode?
        andi    t0, t0, KU_USER
        beqz    t0, resume_kernel
......
resume_kernel:
        local_irq_disable
        lw      t0, TI_PRE_COUNT($28)
        bnez    t0, restore_all
need_resched:
        LONG_L  t0, TI_FLAGS($28)
        andi    t1, t0, _TIF_NEED_RESCHED
        beqz    t1, restore_all
        LONG_L  t0, PT_STATUS(sp)               # Interrupts off?
        andi    t0, 1
        beqz    t0, restore_all
        jal     preempt_schedule_irq
        b       need_resched
```

在 need_resched 中启用进程调度。

其实软中断不只在中断下半部被执行，这一点往往是最容易忽视的地方，通过查找 do_softirq/__do_softirq 在哪些地方被调用，可以知道软件断被处理的时机, 在 3.0.8 内核中，有以下几处:

- invoke_softirq(): 这个是在硬件中断处理完毕时调用的
- local_bh_enable(): 这个很隐蔽，因为这意味着诸如 spin_unlock_bh 之类的调用都会解发软中断的调用
- netif_rx_ni(): 这个调用很自私，为了能快速地收包，“强行” 调度软中断执行。
- run_ksoftirqd(): 这是 ksoftirqd 的执行体，ksoftirqd 用于在出现大量软中断时进行辅助处理。

引入 **ksoftirq** 内核线程的原因

对于软中断，内核会选择在几个特殊时机进行处理。而在中断处理程序返回时处理是最常见的。软中断被触发的频率有时可能很高，更不利的是，处理函数有时还会重复触发自已，那么就会导致用户空间进程无法获得足够的处理时间，因而处于饥饿状态。单纯的对重新触发的软中断采取不立即处理的策略，也无法让人接受。

最初的解决方案：

1. 只要还有被触发并等待处理的软中断，本次执行就要负责处理，重新触发的软中断也在本次执行返回前被处理。这样做可以保证对内核的软中断采取即时处理的方式，关键在于，对重新触发的软中断也会立即处理。当负载很高的时候，此时若有大量被触发的软中断，而它们本身又会重复触发。系统可能会一直处理软中断根本不能完成其他任务。
2. 不处理重新触发的软中断。在从中断返回的时候，内核和平常一样，也会检查所有挂起的软中断并处理他们。但是，任何自行重新触发的软中断不会马上处理，它们被放到下一个软中断执行时机去处理。而这个时机通常也就是下一次中断返回的时候。可是，在比较空闲的系统中，立即处理软中断才是比较好的做法。尽管它能保证用户空间不处于饥饿状态，但它却让软中断忍受饥饿的痛苦，而根本没有好好利用闲置的系统资源。

改进：

最终在内核中实现的方案是不会立即处理处理重新触发的软中断。而作为改进，当大量软中断出现的时候，内核会唤醒一组内核线程来处理这些负载。这些线程在最低的优先级上运行（nice 值是 19），这能避免它们跟其他重要的任务抢夺资源。但它们最终肯定会被执行，所以这个折中方案能够保证在软中断负担很中的时候用户程序不会因为得不到处理时间处于饥饿状态。相应的，也能保证”过量“的软中断终究会得到处理。

每个处理器都有一个这样的线程。所有线程的名字都叫做 ksoftirq/n，区别在于 n，它对应的是处理器的编号。在一个双 CPU 的机器上就有两个这样的线程，分别叫做 ksoftirqd/0 和 ksoftirqd/1。为了保证只要有空闲的处理器，它们就会处理软中断，所以给每个处理器都分配一个这样的线程。

3.3.3 内核中的上下文

上面的内容中提到了上下文这个概念, linux 内核中大致来说有四种上下文:

- 进程上下文: 这种上下文包括内核线程执行体, 用户进程调用 syscall 进入到内核时的执行体, 在这个上下文, 允许进程调度, 从而允许 sleep

- 中断上下文: 顾名思义, 就是硬件中断处理函数的执行体, 这种上下文里的操作应该尽可能的快, 防止在中断中占用太多时间, 在这个上下文不允许调度, 从而不允许 sleep.

  中断上下文也可认为是中断上半部, 这是相对于后面的下半部来说的

- 软中断上下文: 软中断是为了减轻中断上下文的负担而生的, 在系统中其优先级仅次于硬件中断, 仅可被硬件中断打断.

  至此, 我们可以看到内核在处理中断时的大致流程: 硬件中断 –> 硬件中断处理函数的工作是迅速地做一些清理工作, 保留现场供下半部使用 –> 处理软中断 –> 常见的有 tasklet 和内核定时器 –> 处理完软中断后, 恢复正常进程调度 (一般情况下是恢复之前被中断打断的进程)

- 原子上下文: 这不是一种严格意义上的纯粹上下文, 它包括 (但可能不限于):

  – 中断上下文

  – 软中断上下文

  – spinlock 保护的临界区

  事实上, 任何禁止抢断的地方都被视为原子上下文. 需要注意, 禁止中断和禁止抢断是两个不同的概念.

从上面的描述可以看出, 中断处理是一件很繁烦的工作, 因此应该尽量避免产生过多的中断, 也许你认为你的中断 handler 处理的足够快, 但由中断带来的蝴蝶效应不可忽视。

我们还要说明一下通常我们所说的 sleep 的意思: sleep 应被理解为让出 CPU, 只要让出 CPU, 就视为 sleep 了, 有以下两种情况会让出 cpu:

1. 主动 sleep, 可以通过调用 msleep, schedule, schedule_timeout, wait_event 等函数实现
2. 被动 sleep, 通常由以下情况导致:

   (a) 进程所需的资源得不到满足, 比如内存紧张, cpu 需要挂起当前进程去进行内存 shrink

   (b) 时间片用完

3.3.4 内存分配

因为内核编译时不依赖于 libc, 因此很多 libc 中的函数在内核中是无法使用的, 包括大家熟悉的 malloc/free 等.

内核中有很多种分配内存的方法, 下面讲述比较常见的几种:

kmalloc 系列

```
void *kmalloc(size_t size, gfp_t flags)
void *kcalloc(size_t n, size_t size, gfp_t flags)
void *kzalloc(size_t size, gfp_t flags)
void * krealloc(const void *, size_t, gfp_t);
void kfree(const void *);
void kzfree(const void *);
```

使用时需 #include <linux/slab.h>

这个系列的函数分配到的内存在物理上是连续的.
kmalloc 分配内存时不会将分配到的内存清为 0, kcalloc 和 kzalloc 都会将分配到的内存清为 0
kfree 用于释放分配到的内存, kzfree 会在释放前将内存清为 0
可以使用 kmalloc 分配任意大小的内存, 而不是网上或是有的书上说的不能分配大的.
可以看到所有的分配函数都有一个 flags 参数, 常用的有:

- GFP_KERNEL: 这个是最常用的, 大多数分配都会使用这个 flags, 要注意的是, 使用 GFP_KERNEL 分配内存时可能会 sleep, 换句话说, 可能会有进程切换, 因此不能用于原子上下文.
- GFP_ATOMIC: 和 GFP_KERNEL 差不多, 区别有:
  1. 可以用于在原子上下文中分配内存, 因为它不会 sleep
  2. 内存紧张时会失败, 因为它不会启动内存 shrink

关于上下文的概念请看Section 3.3.3中的说明.

__get_free_page 系列

常用的有:

1. __get_free_page(gfp_t flags): 分配一页
2. __get_free_pages(gfp_t flags, unsigned int order): 分配 2^order 页, 可以使用 get_order(size) 来得到 size 对应的 order.
3. free_page(unsigned long addr): 释放__get_free_page 申请的页
4. free_pages(unsigned long addr): 释放__get_free_pages 申请的页

这一系列函数分配到的内存在物理是连续的.
要特别注意不要把 free_page 和 free_pages 弄混了.

vmalloc 系列

和上面两个方法不同之处在于, vmalloc 系统分配的是虚拟地址连接的内存, 这样分配的内存无法保证物理上连续, 但分配成功的可能性较大, 需要特殊需求时, 应该尽量用 vmalloc 分配内存.
这一系列的函数和用户态的 malloc 系列类似, 参数也几乎相同.
这一系列常用的方法有:

1. void *vmalloc(unsigned long size):
2. void *vzalloc(unsigned long size)
3. void vfree(const void *addr)
4. void *ioremap(unsigned long offset, unsigned long size);
5. void iounmap(void * addr);

ioremap 一般用于驱动中将寄存器映射到虚拟地址空间.
vmalloc 和 ioremap 的共同点: 两者都会建立页表
vmalloc 和 ioremap 的区别在于, ioremap 不真正地分配内存

3.3.5 内存映射

在说明内存映射前, 我们来看一下用户态和内核态是如何进行数据交换的.

前面我们说过, 用户态无法直接访问内核态的内存, 同样的, 内核态一般情况下也不可以直接访问用户态的内存, 因此, 有两个函数应运而生:

- long copy_to_user(void __user *to, onst void *from, unsigned long n)
- long copy_from_user(void *to, const void __user * from, unsigned long n)

前者用于将内核态的数据拷贝到用户态提供的内存空间里, 后都用于将用户态的数据拷贝到内核态的内存空间里.

对于少量的数据, 这样做是没有问题的, 但对于大量的数据, 比如显存, 这样反复拷贝就很不高效了, 因此内核提供了内存映射机制, 用于将内核态的内存映射到用户空间, 映射完成后, 用户态就可以直接访问这块内存了, 用户态的接口是:

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

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

在内核态, 需要实现 file_operations 中的 mmap 回调函数, 实现一般都大同小异, 可以从内核中其他 mmap 实现拷贝即可. 唯一需要注意的是映射的方式是 cache 的还是 uncached, 比如 LCD, 因为需要用 DMA 搬运, 因此需要是 uncache 的.

cache 还是 uncache 是由内核态的 mmap 回调函数决定的, 用户态无法决定.

对于 mmap 还有其他一些高级的用法, 比如将整个文件映射到用户态, 从而可以像访问内存一样访问文件.

3.3.6 高端内存

<TBC>

3.3.7 Concurrency(并发) and Race Conditions(竞态)

并发 (Concurrency) 指的是多个执行单元同时、并行被执行.

并发的执行单元对共享资源的访问很容易导致竞态 (Race Condition).

访问共享资源代码区域称为临界区 (critical section), 临界区必须用某种互斥机制加以保护.

造成竞态的主要原因:

- SMP systems can be executing your code simultaneously on different processors
- 进程调度与抢占
- 中断 (硬中断, 软中断) 与进程之间的竞争

linux 上的并发与竞态控制机制有 (依” 流行度” 排名):

- spinlock(自旋锁)
- Semaphore and Mutexes(信号量与互斥)

- 原子操作
- Completions(完成量)
- seqlock
- RCU
- 中断屏蔽

＊＊＊＊＊ TODO: 理解多核与单核 CPU 上并发与竞态的区别＊＊＊＊＊

spinlock

使用 spinlock 能保证同一时间只有一个进程进入临界区, 如果一个 spinlock 已经被占用, 其他试图占用这个锁的进程将会在锁上忙等, 直到锁被其他进程释放.

常见用法:

```
#include <linux/spinlock.h>
spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* static Initialize */

/* dynamic initialize
 *
 * spinlock_t mylock;
 * spin_lock_init(&mylock);
 */

spin_lock(&mylock);

/* ... Critical Section code ... */

spin_unlock(&mylock); /* Release the lock */
```

上而的自旋锁只能保证进程与进程之间互斥, 不能保证进程与中断之间的互斥, 于是 linux 还提供了以下版本:

与硬中断互斥

- void spin_lock_irqsave(spinlock_t *lock, unsigned long flags: 保存当前的中断状态 (禁止 or 启用), 禁止中断
- void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);: 和 spin_lock_irqsave 是一对的
- void spin_lock_irq(spinlock_t *lock);: 禁止中断
- void spin_unlock_irq(spinlock_t *lock);: 和 spin_lock_irq 是一对的

irq 和 irqsave 的版本的区别就在于 irqsave 会保存当前的中断状态, 如果调用 spin_lock_irqsave 之前中断就是关着的, irqrestore 之后仍然是关着的. spin_lock_irq 无法保证这一点.

如果你有十足的把握在获取 spinlock 之前中断是开着的, 就可以用 spin_lock_irq 版本, 这样能省几条指令。spin_unlock_irq 会打开中断，如果在中断禁止的情况下调用了 spin_lock_irq/spin_unlock_irq，则会错误地提前打开中断，这是一种常见的错误。

如果你不确定获取 spinlock 之前中断的状态, 则使用 spin_lock_irqsave 版本.

spin_lock_irq 的用法和 spin_lock 是一样的, 下面是 spin_lock_irqsave 的用法:

```c
#include <linux/spinlock.h>
spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* static Initialize */

/* dynamic initialize
 *
 * spinlock_t mylock;
 * spin_lock_init(&mylock);
 */

unsigned long flags;

spin_lock_irqsave(&mylock, flags);

/* ... Critical Section code ... */

spin_unlock_irqrestore(&mylock, flags); /* Release the lock */
```

与软中断的互斥

- void spin_lock_bh(spinlock_t *lock): 禁止软中断
- void spin_unlock_bh(spinlock_t *lock): 和 spin_lock_bh 是一对的

使用 spinlock 需要注意的地方:

- 在 spinlock 所保护的临界区内不能 sleep
- 同一个锁建议自始自终使用同一个类型的方法, 回顾一下, 我们有四种类型:

  1. 纯 spin_lock
  2. irqsave
  3. irq
  4. bh

  同一个锁不建议一会用 irqsave 的, 一会用 bh 的, 一会又是纯 spin_lock, 如果出现这种情况, 一般情况下说明程序存在问题, 需要重构代码, 如果还出现这种情况, 考虑一下是否需要定义多个锁.

  之所以这样建议, 不是说 linux 的 spinlock 实现不允许这样做， 而是这样容易出错，指不定会把自己也弄晕。

- lock 和 unlock 最好不要跨方法, 也最好不要跨进程, lock 和 unlock 应该总是成双成对地出现在临界区的前后.

  所谓的跨方法是指在 A() 中调用 lock, 在 B() 调用 unlock, 而 A 和 B 这俩并不总是成对出现.

  所谓的跨进程是指在进程 T1 中调用 lock, 在进程 T2 中调用 unlock

  当出现这种情况时, 大部分情况下都说明您对临界区的定义存在问题, 您没有正确识别出您的临界区.

  如果一定需要跨进程或是跨方法, 请反复检查程序的运行路径, 确保如下两点:

  1. lock 和 unlock 能配上对, 否则可能会导致死锁.
  2. 不会 sleep

- 临界区的执行时间应该尽可能短并且是可确定的, 特别是使用 irq/irqsave 版本时
- 不要忘记在出错处理时 unlock 获得的锁, 这是一个常见的错误:

```
spin_lock(&mylock);

......

if (error) {
        spin_unlock(&mylock);
        return;
}

......

spin_unlock(&mylock); /* Release the lock */
```

- 所有的 spinlock 方法都是接受一个指针类型的 spinlock_t 变量, 一种常见的错误就是直接写成了 spin_lock(mylock), 在单核 CPU 上, 这种错误不会导致问题, 他在多核 CPU 上, 这样做可能会导致系统崩溃, 或者是其他一些稀奇古怪的问题.

Mutex

在使用 spinlock 时问题就是不能 sleep, 而且临界区的执行时间不能太长, 如果一定需要 sleep, 或者执行时间不确定或者很长, 需考虑使用 Mutex, 但 Mutex 本身会 sleep, 因此不能用于原子上下文. (世间之事总是不能十全十美)

Mutex 使用一种事件等待机制实现, 而不是 spinlock 的忙等机制, 得不到 Mutex 的进程会 sleep, 直到有人释放 Mutex.

常见用法如下:

```
#include <linux/mutex.h>

/* Statically declare a mutex. To dynamically
   create a mutex, use mutex_init() */
static DEFINE_MUTEX(mymutex);

/* Acquire the mutex. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, mutex_lock() puts the calling thread to sleep.
 */
mutex_lock(&mymutex);

/* ... Critical Section code ... */

mutex_unlock(&mymutex);      /* Release the mutex */
```

常用的 mutex 方法有:

- DEFINE_MUTEX(name): (静态) 定义一个名字为 name 的 mutex
- void mutex_init(struct mutex *lock): (动态) 初始化一个 mutex
- void mutex_lock(struct mutex *lock): 申请占用 mutex, 不可被 signal 中断, 即忽略发往这个进程的 signal, 可能会导致进程永远杀不死

- int mutex_lock_interruptible(struct mutex *lock): 申请占用 mutex, 可被中断
- int mutex_lock_killable(struct mutex *lock): 申请占用 mutex, 可被 SIGKILL 唤醒
- int mutex_trylock(struct mutex *lock): 尝试占用 mutex, 返回 1 表示成功占用, 返回 0 表示被其他人占用了
- void mutex_unlock(struct mutex *lock): 释放 mutex

信号量 (Semaphore)

这里所说的信号量在学校的时候所说的 P/V 操作, 在打算使用 Semaphore 时, 请思考一下您真正需要的是 mutex, 从理论上讲, mutex 就是资源数为 1 的 Semaphore, 但 linux 针对 mutex 的实现进行了优化, 而且大部分时间我们其实需要的是 mutex(此时唯一的资源就是临界区).

Semaphore 和 mutex 的一些属性上是相同的, 比如说会 sleep, 不能用于原子上下文, 拥有信号量期间可以 sleep 等等.

基本用法如下:

```
#include <asm/semaphore.h>  /* Architecture dependent
                               header */

/* Statically declare a semaphore. To dynamically
   create a semaphore, use init_MUTEX() */
static DECLARE_MUTEX(mysem);

down(&mysem);    /* Acquire the semaphore */

/* ... Critical Section code ... */

up(&mysem);      /* Release the semaphore */
```

相关的函数有

```
DEFINE_SEMAPHORE(name); // 静态创建一个名字为 name 的未被 lock 的信号量, 资源数为 1, 建议使用 DEFINE_MUTEX
void sema_init(struct semaphore *sem, int val); // 动态初始化一个信号量, 不要在信号量被占用的时候初始化信号量
void down(struct semaphore *sem); // 占用信号量 (P), 不可被中断, 即忽略发往这个进程的 signal, 可能会导致进程永远杀不死
int down_interruptible(struct semaphore *sem); // 占用信号量 (P), 可被中断
int down_killable(struct semaphore *sem); // 占用信号量 (P), 可被 SIGKILL 唤醒
int down_trylock(struct semaphore *sem); // 尝试占用信号量, 返回 1 表示已经被他人占用, 返回 0 表示成功占用信号量, 这个方法可以在原子上下文使用
int down_timeout(struct semaphore *sem, long jiffies); // 占用信号量, 如果在 jiffies 时间内还抢不到, 则退出, 返回 0 表示成功占用, 返回 -ETIME 表示抢占失败
void up(struct semaphore *sem); // 释放信号量 (V)
```

原子操作及免锁算法

锁是有代价的, 无论是 spinlock 还是 mutex, 都有较大的系统开销, 因此对于一些简单的操作 (比如 i++), linux 为我们提供了称为原子操作的方法, 这些方法一般都需要 CPU 的配合, 在 MIPS 上是借助于 LL(load linked) 和 SC(store condition) 来实现的. 这些原子操作可以分为两类:

所谓原子, 可以将其理解为不管实际指令数是多少, 执行是" 不会" 被打断, 看起来就像是一条指令.

整型原子操作

例如 atomic_inc(i), 相当于 i++, 但是是原子的

具体请参见其他参考书籍，当然, 最好的书是内核原代码. 相关的实现在 include/asm-generic/atomic.h, MIPS 优化过的一些操作位于 arch/mips/include/asm/atomic.h

位原子操作

比如 set_bit(int nr, volatile unsigned *addr), 可以将从 addr 起第 nr 位" 原子" 地置为 1, nr 可以是一个大于 32 的数.

相关实现位于 include/asm-generic/bitops/atomic.h, MIPS 优化过的代码位于 arch/mips/include/asm/bitops.h

读写锁 (Reader-Writer lock)

自旋锁不关心临界区究竟是怎样的操作, 不管是读还是写, 都一视同仁. 即便多个执行单元同时读临界资源也会被锁住. 实际上, 对共享资源并发访问时, 多个执行单元同时读取它是不会有问题的, 读写锁就是为了解决这种情况的. 读写锁允许同时读, 但不允许同时写, 写时禁止读, 写操作优先级高于读操作.

基本使用方法如下:

```
#include <linux/rwlock.h>
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

// 读者
read_lock(&myrwlock);     /* Acquire reader lock */
/* ... Critical Region ... */
read_unlock(&myrwlock);   /* Release lock */

// 写者
write_lock(&myrwlock);    /* Acquire writer lock */
/* ... Critical Region ... */
write_unlock(&myrwlock);  /* Release lock */
```

读写锁是从 spinlock 派生的, 因此和 spinlock 一样, 也有 irq/irqsave/bh 这些变种, 其用法和原则也是一样的, 这里不再详述.

其他锁机制

linux 还提供了 seqlock, 和读写锁差不多, 这个锁最 “轻”，但也更" 忙", 只有在非常确定读的时候多，写的时候少时才可使用，感兴趣的可以查阅相关资料。

还有 RCU(Read-Copy-Update), 这个锁在内核中大量使用，其用法较为简单，但要理解其机制可不是一时半会能说明白的，感兴趣的可以从访问以下资源

- LWN 上和 RCU 相关的文件
- `http://en.wikipedia.org/wiki/Read-copy-update`
- RCU 作者的网站: `http://www.rdrop.com/~paulmck/RCU/`

Completion

Completion 和 Semaphore 在用法上差不多，仅仅是语义上以及方法名上不些许不同。

Semaphore 应被视为一种信号量的通用机制，而 Mutex 和 Completion 是 Semaphore 在互斥和事件完成通知上的优化实现。

互斥强调资源的唯一性, 而事件完成体现在有多个进程等待在一件事情的完成。他们都是 Semaphore 的语义延伸，侧重于不同的方面进行了优化。

相关的方法有:

```
DECLARE_COMPLETION(work)
DECLARE_COMPLETION_ONSTACK(work)
void init_completion(struct completion *x)
void wait_for_completion(struct completion *);
int wait_for_completion_interruptible(struct completion *x);
int wait_for_completion_killable(struct completion *x);
unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout);
long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout);
long wait_for_completion_killable_timeout(struct completion *x, unsigned long timeout);
bool try_wait_for_completion(struct completion *x);
bool completion_done(struct completion *x);
void complete(struct completion *);
void complete_all(struct completion *);
```

相信有了之前的讲解，仅看方法名大家就能猜到这些方法的意义。

Completion 和 Wait Queue(wait_event 系列) 的区别在于: 后者会在 sleep 之前和之后检查条件是否成立，如果 sleep 之前条件就成立，就不会进休眠，如果 sleep 之后条件不成立而且没超时，就会接着睡。在大部分时候，应优先考虑 wait_event 系列

中断屏蔽

避免竞态的一种简单的方法是进入临界区之前屏蔽系统的中断. 从而

- 当前进程不会被中断打断, 从而不会和中断竞争
- linux 的进程调度依赖于中断, 中断一关, 也就没有进程调度了, 从而避免其他进程的抢占

注意: 不要长时间屏蔽中断.

3.3.8 中断处理

本节所谓的中断处理只指中断处理的上半部，而中断的重点在于下半部的实现，基本上，在上半部我们只用到两个函数:

- int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev), 对其参数说明如下:
  - irq: irq 号，在 arch/mips/jz4770/include/mach/chip-intc.h 中定义，对应 4770 spec 上 INTC 的 ICSR0 和 ICSR1 两个寄存器，以及 DMA 等二级中断源。
  - handler: 中断处理函数，typedef irqreturn_t (*irq_handler_t)(int irq, void *dev)，其中 irq 为中断号, dev 即调用 request_irq 时指定的 dev 参数, 典型的中断处理函数实现如下:

```
irqreturn_t xxx_handler(int irq, void *dev) {
        // 1. 将 dev 转换为特定于驱动的实际数据结构
        struct xxx_priv *priv = (struct xxx_priv *)dev;

        // 2. 硬件层面必要的处理，比如查询状态寄存器，清状态等
        intr_status = REG32(XXX_STATUS);
```

```
7         if (intr_status & XXX_YYY) {
8                 /* do something */
9         }
10
11        // 3. 保存中断发生时的现场, 使用适当的 Deferred work 机制将费时的处理转入中断下半部
12
13        return IRQ_HANDLED; // 最后要返回这个, 表示中断处理完毕, 一般地, 在我们这种嵌入式系
              统里, 总是返回 IRQ_HANDLED
14 }
```

之所以有 irq 参数, 是因为有时候多个中断会共享一个处理函数。

– flags: 传 0 即可，如果想为系统的随机数生成器做贡献, 可以传 IRQF_SAMPLE_RANDOM

– name: 中断的名称, 在 cat /proc/interrupts 时显示, 在中断管理代码中直接保存了传入的指针, 没有复制一份拷贝, 因此不要在栈上分配 name 指向的空间, 可使用全局变量或是固定字符串。

– dev: 任意私有数据，没有时传 NULL。

- void free_irq(unsigned int irq, void *dev_id): 释放 irq，其中 irq 和 dev_id 需和 request_irq 时一致。

在我们的系统中，还有以下函数，用于动态申请 DMA 通道:

```
1 int jz_request_dma(int dev_id, const char *dev_str,
2                    irqreturn_t (*irqhandler)(int, void *),
3                    unsigned long irqflags,
4                    void *irq_dev_id)
```

其中的 dev_id 是模块 ID，在 arch/mips/jz4770/include/mach/dma.h 中的 DMA_ID_开头的宏定义, 比如 DMA_ID_I2C0 表示 I2C 的第 1 个控制器。

dev_str 即 request_irq 的 name。

中断处理的学习重点:

- 理解 CPU 及各个外设模块的中断产生机制及对中断处理的要求
- 理解下半部各种机制的特性及使用，选择适当的下半部机制

3.3.9 错误处理

由于内核代码不依赖于 libc，因此没有 errno 之类的东东，内核的错误处理完全依赖于函数的返回值。内核定义了一些常用的返回值定义:

error code

内核的 error code 定义主要集中在以下三个文件:

- include/asm-generic/errno-base.h
- include/asm-generic/errno.h
- include/linux/errno.h

最后一个包含了前两个，因此在实际应用时只需要 “#include <linux/errno.h” 即可

返回值为指针时的错误返回

当一个函数返回值为指针时，只是返回 NULL 有时无法表述具体的错误原因，内核为我们提供了以下方法来辅助这种情形:

```
#define MAX_ERRNO       4095

#define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)

static inline void * __must_check ERR_PTR(long error)
{
        return (void *) error;
}

static inline long __must_check PTR_ERR(const void *ptr)
{
        return (long) ptr;
}

static inline long __must_check IS_ERR(const void *ptr)
{
        return IS_ERR_VALUE((unsigned long)ptr);
}

static inline long __must_check IS_ERR_OR_NULL(const void *ptr)
{
        return !ptr || IS_ERR_VALUE((unsigned long)ptr);
}

static inline int __must_check PTR_RET(const void *ptr)
{
        if (IS_ERR(ptr))
                return PTR_ERR(ptr);
        else
                return 0;
}
```

他们都定义在 include/linux/err.h 中，虽然看上去只是简单的类型转换，但他们的魔力在于:

1. 可以使用简单的类型转换的原因在于，内核返回的指针一般是指向页面的边界 (4K 边界)，即

   ptr & 0xfff == 0

   这样 ptr 的值不可能落在（0xfffff000，0xffffffff）之间，而一般内核的出错代码也是一个小负数，在 -4095 到 0 之间，转变成 unsigned long，正好在（0xfffff000，0xffffffff) 之间。这就是 IS_ERR_-VALUE 实现的原理，依据这个原理来判断内核函数的返回值是一个有效的指针，还是一个出错代码。

   这也意味着，如果你的指针值本身就会位于 (0xfffff000,0xffffffff) 之间，你就不能使用上面这些方法。

2. 它们都声明了__mast_check，这样函数调用者将被强制检查返回值，否则编译器会报错。

3.4 调试机制及调试方法

3.4.1 打印信息

linux 内核使用 printk 打印信息，printk 在内核中的地位相当于 C 库中的 printf，但 printk 有一些 printf 没有的特性 (我想这就是它不叫 printf 的原因吧)。

printk 使用方法

```
printk(日志级别"format string", ...);
```

要注意在日志级别与 format string 除空白外不允许有其他字符 (看了日志级别的定义您就明白了), 下面是日志级别的定义:

```
#define KERN_EMERG      "<0>"   /* system is unusable                   */
#define KERN_ALERT      "<1>"   /* action must be taken immediately     */
#define KERN_CRIT       "<2>"   /* critical conditions                  */
#define KERN_ERR        "<3>"   /* error conditions                     */
#define KERN_WARNING    "<4>"   /* warning conditions                   */
#define KERN_NOTICE     "<5>"   /* normal but significant condition     */
#define KERN_INFO       "<6>"   /* informational                        */
#define KERN_DEBUG      "<7>"   /* debug-level messages                 */
```

定义于 include/linux/printk.h 中。

日志级别是可选的，不指定级别时，内核会使用一个默认的级别，这个级别可以通过 kernel hacking->Default message log level 设置，默认是 4。

日志级别决定了哪些打印会最终输出，默认情况下，任何小于 KERNEL_DEBUG 的消息都会被输出。

可以在运行过程中修改默认日志级别以及阈值，可通过以下命令查看当前的级别信息:

```
# cat /proc/sys/kernel/printk
7 4 1 7
```

这四个值对应 kernel/printk.c 中的 console_printk 数组；第一个 7 是打印阈值，第二个是默认消息级别，第三个是通过 sysctl 接口改变日志级别时允许设置的最小值，第 4 个暂无用处。

通过如下命令可以修改上面的数值:

```
# echo "4" >/proc/sys/kernel/printk
# cat /proc/sys/kernel/printk
4 4 1 7
# echo "4 2" >/proc/sys/kernel/printk
# cat /proc/sys/kernel/printk
4 2 1 7
```

我们还可以通过在内核命令行参数中指定 ignore_loglevel 来取消打印阈值的限制，使得所有级别的信息都输出。

因为我们可以灵活控制打印信息，建议平时开发时保留重要的打印信息，视信息的重要程度使用不同的级别

查看打印信息

日志级别只是控制打印是否显示 (比如串口上)，但并不表示不打印，可以通过 dmesg 命令查看当前所有的内核打印信息。使用 “dmesg -c” 可以清空当前内核打印信息。

控制打印与否

从上面的描述可以看出，日志级别只是决定是否显示到终端，并不表示不打印，但有时候，我们可能需要完全禁止打印，可参考以下方法:

在公共头文件里定义一个宏:

```
#ifdef XXX_DEBUG
#define PDEBUG(fmt, args...) printk(KERN_DEBUG "xxx: " fmt, ## args)
#endif
```

在相应的 Makefile 里:

```
EXTRA_CFLAGS += -DXXX_DEBUG
```

当然，您也可以将 XXX_DEBUG 就放在公共的头文件里。

printk 使用注意事项

- 一次性输入给 printk 的数据不能超过 1024 字节，这是由 printk.c 中的 static char printk_buf[1024]; 数组决定的。
- printk 会禁止抢断及中断，也就是说 printk 在输出的时候不会被任何人打断，因此在打印大量信息时，可能会在 printk 中耗费很长的时间，需要特别注意。一般情况下，一个 20–30 字符的输出会耗费 10ms 左右的时间。

  注: 虽说 printk 一般情况下会花费较长时间，但您不能依靠这个来做延时，因为 printk 有可能很快就会退出。这种情况出现在有多个 printk 同时输出时，只有其中一个 printk 负责向终端输出
- printk 是个函数调用，编译器的优化不会跨函数，因此有时候会发现加一个打印就好了，此时有两种可能性:
  - 由于 printk 的延时导致
  - 由于优化被取消导致，你可以将加了打印和未加打印的反汇编结果进行对比，也可以将打印去掉，并且在编译时去掉优化选项，来确认是否由于编译优化导致的问题。
- 死机时，可能上一条打印还没完就死机了，此时可以在打印加一些延时，确保打印输出。

关于 printk 的更多信息可参考 Documentation/CodingStyle 的 “Chapter 13: Printing kernel messages”。

3.4.2 proc 和 sysfs 文件系统

sysfs 和设备驱动联系较为紧密，因此留待讲解 Linux 设备驱动模型时再详细说明。

procfs 向用户展示的是系统中进程以及其他核心部件 (中断管理，内存管理等) 的相关信息，下面对一些重要的目录及文件进行说明:

/proc/PID:

这里的 PID 就是那些全为数字的目录，目录名即进程的 ID，目录中包含很多文件，较重要的有:

- cmdline: 启动这个进程的命令行, 对于内核线程，这个文件的内容为空
- exe: 批向进程对应的可执行文件的软链接
- cwd: 指向进程当前工作目录的软链接

- fd: 这个目录下包含进程打开的所有文件，文件名 file descriptor 号，链接到具体的文件
- fdinfo: 这个目录包含进程打开的文件的信息，包括当前文件的偏移和 flag
- maps: 这个文件包含进程使用 mmap 映射的文件信息，还包括 heap 和 stack 的信息
- mem: 表示进程的虚拟内存空间 (Virtual Memory)，由此可获知进程当前的运行信息 (不能直接 cat，可使用 ptrace 访问)
- root: 当前进程的根目录，一般是 "/"，如果是在 chroot 环境下运行，就是 chroot 后的目录
- status: 进程当前状态及内存使用情况等
- wchan: Waiting Channel，在进程处于 sleep 状态时，指示进程当前在哪个函数中等待。
- task: 子进程信息
- oom_*: 和 Android 的 Out-of-Memory killer 相关

一般情况下我们不直接查看这些信息，使用 ps 命令就足够了

/proc/cpuinfo:
cpu 的信息

/proc/cmdline:
内核命令行

/proc/interrupts:
中断信息

/proc/meminfo:
内存使用信息

/proc/mounts:
当前的文件系统挂载信息

/proc/partitions:
分区信息

/proc/slabinfo:
slab 分配器相关的信息

/proc/version:
内核版本信息

3.4.3 解析内核崩溃信息

这里我们故意制造一个空指针错误来说明如何解析内核崩溃信息 (在 linux 的世界，我们称之为 Oops)。

首先来看我们制造出来的 Oops:

```
CPU 0 Unable to handle kernel paging request at virtual address 00000000, epc == 802d6120, ra == 802d6114
Oops[#1]:
Cpu 0
zero[$0]=00000000   at[$1]=00000001   v0[$2]=deadbeaf   v1[$3]=8d121620
 a0[$4]=8d121618   a1[$5]=8d024000   a2[$6]=bd646000   a3[$7]=000001f4
 t0[$8]=8d029d70   t1[$9]=00000002  t2[$10]=00000008  t3[$11]=10000400
 t4[$12]=0000000d  t5[$13]=00000010  t6[$14]=804575c0  t7[$15]=ffffffff
 s0[$16]=00000001  s1[$17]=8d121800  s2[$18]=80580000  s3[$19]=8d029d74
 s4[$20]=8d121820  s5[$21]=00000007  s6[$22]=8d029d70  s7[$23]=00000000
 t8[$24]=00000018  t9[$25]=00000000  k0[$26]=8d029d78  k1[$27]=80131678
 gp[$28]=8d028000  sp[$29]=8d029d58  s8[$30]=00000000  ra[$31]=802d6114
Hi    : 00000000
Lo    : 00001388
epc   : 802d6120 mma7660_probe+0x17c/0x424
   Not tainted
ra    : 802d6114 mma7660_probe+0x170/0x424
Status: 10000403    KERNEL EXL IE
Cause : 0080000c
This is TLB exception (store)
BadVA : 00000000
PrId  : 2ed1024f (Ingenic XBurst)
Modules linked in:
Process swapper (pid: 1, threadinfo=8d028000, task=8d024000, tls=00000000)
Stack : 00000000 8d018e40 00000008 8d029d90 80570000 8d121820 8d120007 00e06a08
        8d1237b0 8d121820 80517780 8d121800 8d121820 805177a8 8d2eca00 80534fe8
        00000000 802d0b04 8d2eca00 00000000 8d054688 801327c4 80570000 80570000
        8d121820 8023ed4c 8d2eca80 8d029dc8 8d2ea7e0 801d8b4c 8d029e10 8d121854
        8d121820 805177a8 8023eea4 8023ef60 00000000 801d942c 8d2e9b80 00000000
        ...
Call Trace:
[<802d6120>] mma7660_probe+0x17c/0x424
[<802d0b04>] i2c_device_probe+0x12c/0x164
[<8023ed4c>] driver_probe_device+0xa0/0x1f8
[<8023ef60>] __driver_attach+0xbc/0xc4
[<8023e334>] bus_for_each_dev+0x68/0xd0
[<8023d954>] bus_add_driver+0xc8/0x2dc
[<8023f738>] driver_register+0x88/0x184
[<802d0e94>] i2c_register_driver+0x34/0xcc
[<80010440>] do_one_initcall+0x40/0x1f8
Code: 04400003  3c02dead  3442beaf <ac020000> 8e448854  00002821  24840004  0c013bb0  00003021
android_work: sent uevent USB_STATE=CONNECTED
---[ end trace 90933faa0efa032f ]---
android_work: sent uevent USB_STATE=DISCONNECTED
Kernel panic - not syncing: Attempted to kill init!
```

CPU寄存器信息

epc, ra, status, cause, BadVA, 都是熟悉的身影

进程信息及当前的栈内容

对栈的解析，打印出了函数调用过程

格式: [<内存地址>] 地址对应的函数名+函数内字节偏移/函数总字节数

当前的指令

这个是混进来的其他信息，亲，把他忽略

从打印信息来看，当前的栈顶是 “[<802d6120>] mma7660_probe+0x17c/0x424”, 也就是说，问题出在地址 0x802d6120 处, 方法 mma7660_probe 中，mma7660_probe 总计有 0x424 字节，出错的位置位于 0x17c 偏移处。

那么接下来反编译内核 (也可以只反编译 mma7660.o，但反编译整个内核我们可以不用做一次加法去算出错的指令位置)

```
mips-linux-gnu-objdump -d vmlinux >k.dump
```

查找 k.dump，找到 mma7660_probe 的实现, 如下:

```
802d5fa4 <mma7660_probe>:
802d5fa4:       27bdffb8        addiu   sp,sp,-72
802d5fa8:       afb1002c        sw      s1,44(sp)
802d5fac:       afbf0044        sw      ra,68(sp)
......
802d6114:       04400003        bltz    v0,802d6124 <mma7660_probe+0x180>
802d6118:       3c02dead        lui     v0,0xdead
802d611c:       3442beaf        ori     v0,v0,0xbeaf
```

```
802d6120:       ac020000        sw      v0,0(zero)   ;<--- 出错位置
802d6124:       8e448854        lw      a0,-30636(s2)
802d6128:       00002821        move    a1,zero
802d612c:       24840004        addiu   a0,a0,4
......
802d63bc:       248477d8        addiu   a0,a0,30680
802d63c0:       080b57fd        j       802d5ff4 <mma7660_probe+0x50>
802d63c4:       8fbf0044        lw      ra,68(sp)
```

接下来，我们需要将汇编指令对应到 C 代码上，可以使用 gnu as 的 listing 功能来实现:

```
$ touch drivers/i2c/chips/mma7660.c
$ make V=1 zImage
$ mips-linux-gnu-gcc -g -Wa,-adhls ... drivers/i2c/chips/mma7660.c >mma7660.temp
```

前两步是为了得到编译 mma7660.c 的 gcc 命令行，第三步给 as 传递了 “-adhls” 选项，并且使用 -g 编译, 默认输出是 stdout，因此需要重定向到 mma7660.temp。查看 mma7660.temp，查找 “sw $2,0($0)”，其中 “$2” 即 v0, “$0” 即 zero, 可以找到以下内容:

```
 1285                   .loc 1 423 0
 1286 05b8 ADDE023C     li $2,-559087616
 1287                   $LVL109:
 1288 05bc AFBE4234     ori $2,$2,0xbeaf
 1289 05c0 000002AC     sw $2,0($0)
```

也就是说，这句指令对应源码的第 423 行，让我们来看一下代码:

```
static volatile unsigned int *bad = NULL;

static int mma7660_set_mode(u8 mode)
{
        u8 buffer[2];
        int ret;

        buffer[0] = MMA7660_REG_MODE;
        buffer[1] = mode;
        ret = mma7660_i2c_txdata(buffer, 2);
        if (ret < 0)
                return ret;

        *bad = 0xdeadbeaf;

        return 0;
}
```

423 行处正是我们故意加入的实验语句。

3.4.4 Oops 信息内幕

其实不是什么内幕，只是一些小 tips 而已。

1. oops 信息是在 arch/mips/kernel/traps.c 中打印的，感兴趣的话可以查看其实现，有时候，如果默认打印的信息不够，或是需要打印自己的一些信息，都需要来改动 traps.c 中相应的实现;
2. 除了 oops 信息，有时我们还会看到一些警告信息 (WARNINGS)，这些一般都是由于 WARN_ON 宏输出的，其最终是由 warn_slowpath_common() 输出的，如果需要打印更多信息，需要手工做一些修改

3. sysrq 是个不错的东东，虽然在嵌入式系统上看似派不同用场，但其实现代码为我们提供了许多“窥视” 内核的实例，其代码位于 drivers/tty/sysrq.c，在这里你可以看到如何打印内核的进程信息、内存信息等等。
4. 崩溃时打印的信息不总是可靠的 (虽然在大部分情况下是可靠的)，特别是在中断环境出出错的时候，因此不要为 Oops 信息不对而太感惊讶。

3.4.5 几个重要的打印函数

- dump_stack(): 打印堆栈信息，对我们来说最重要的时知道当前函数是从哪里调过来的。
- void panic(const char * fmt, ...): 其实 Oops 信息就是 panic 打印的，我们也可以在需要的地方调用 panic
- WARN(condition, message): 在 condition 成立的时候打印 message 及类似乎 Oops 的信息，只是打印 Warning，系统 (不出意外的话) 照常运行
- WARN_ON(condition): 与 WARN 区别是不需要指定一个自己的 message
- BUG_ON(condition): 在 WARN_ON 的区别是系统会 panic，停止运行
- BUG(): 对 panic 的封装，在 panic 之前会打印当前的函数名及行号，以及 C 文件名

3.4.6 watch

在内核里没有一个统一加 watch 的地方，因此在需要 watch 的时候需要手工在代码的某处添加，相关的方法可参照 arch/mips/kernel/watch.c

3.4.7 kprobe

<TBC>

Chapter 4

linux 设备驱动基础

4.1 内核模块

内核模块就像是动态库一样，可以动态地从内核中卸载和加载，但内核模块也可以静态地编进内核里，也许将内核模块称为一个内核态 Application 更为贴切。

既然如此，就让我们来看一下著名的 “Hello World!” 是如何用内核模块实现的:

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

使用到模块机制的文件都需要包含这个头文件

```
static char *whom = "Lutts";
static int howmany = 1;

module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

可以给模块传递参数

参数会以文件的形式同时会出现在/sys/module/hello/parameters
在这里参数对所有用户都是只读的

```
static int __init hello_init(void) {
    int i = 0;
    for (i = 0; i < howmany; i++) {
        printk("Hello %s\n", whom);
    }

    return 0;
}

static void __exit hello_exit(void) {
    int i = 0;

    for (i = 0; i < howmany; i++) {
        printk("Goodbye %s\n", whom);
    }
}
```

模块初始化完后会释放函数指令所占用的内存

返回0表示加载成功，否则加载失败

执行完后释放内存(非立即)

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

相当于main函数，模块的入口，模块加载时调用

卸载模块时被调用，通常做一些和module_init相反的清除操作

```
MODULE_DESCRIPTION("Hello Example");
MODULE_AUTHOR("Lutts Wolf <slcao@ingenic.cn");
MODULE_LICENSE("GPL");
```

模块信息，在嵌入式系统上，Description和Author其实都不重要，但license非常重要，GPL的模块不能调用非GPL的函数

将这些内容存为 hell.c, 放在 drivers/char/目录下，修改 drivers/char/Makefile, 在最后增加 “obj-m +=

hello.o", 然后在内核根目录下执行 "make" 即可编译模块，生成 drivers/char/hello.ko 文件, 将这个 ko 文件 push 到 Android 上，执行以下命令加载模块

```
# busybox insmod hello.ko whom='Lutts Cao' howmany=10
```

即可在串口中打印 10 行 "Hello Lutts Cao"。

4.1.1 模块参数

当前内核支持的模块参数类型可查看 include/linux/moduleparam.h, 下面是 3.0.8 内核支持的类型:

- module_param: 支持标准类型的参数，具体类型请参见其注释，其中的 charp 是指字符指针，要注意。
- module_param_cb: 支持 get/set 回调
- module_param_string: 字符数组 (注意要留出一个 NULL)
- module_param_array: 数组，传参时使用逗号分隔数组元素

4.1.2 导出符号

这个也类似于动态库的导出符号，没有导出的符号是不可以被其他模块调用的，同样的，内核中没有导出的符号在模块中也不能调用。导出符号一般使用以下两个宏:

- EXPORT_SYMBOL
- EXPORT_SYMBOL_GPL

4.1.3 模块信息

最重要的就是 license，具体支持的 license 参见 include/linux/module.h 中 MODULE_LICENSE 的定义处。

要特别注意这个 license，他决定了模块能调用哪些 EXPORT 了的 SYMBOL，有时候你发现一个函数你明明定义了，也 EXPORT 了，但就是在别的模块用不了，此时一般都是 license 问题

4.1.4 模块编译

上面的例子中使用的模块就位于内核代码树中，此时需要修改相应的 Kconfig 和 Makefile（例子中我为了省事没有使用 Kconfig)。我们的模块代码其实不一定非要放在内核源代码目录下，可以放在别的地方，但此时的 Makefile 稍有不同, 可参照下面的模板:

```
1  ifneq ($(KERNELRELEASE),)
2
3  #EXTRA_CFLAGS := -I.
4  obj-m += xxx.o
5
6  else
7  KERNELDIR ?= < 内核所在的目录>
8  PWD := $(shell pwd)
9
10 default:
```

```
11          $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
12
13  clean:
14          rm -rf *.o *.ko *.mod.c .*.cmd \
15                  Module.symvers modules.order
16  endif
```

位于内核代码树之个的模块不需要 Kconfig.

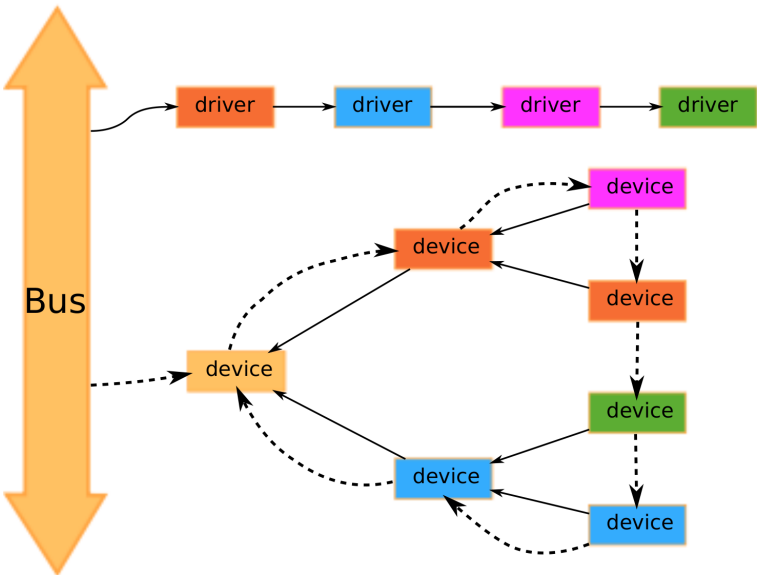
4.2 Linux 设备驱动模型

4.2.1 Linux 设备驱动模型

从驱动开发人员的角度，Linux 设备驱动模型建立在 bus-driver-device 模型上，这个模型中与 sysfs 打交道的则是 kobject-kobj_type-kset, Figure 4.1展示了 Linux 驱动模型中各个元素的关系。

图中使用了面向对象的方式来理解 kobject 等，和实际代码并不对应，Linux 内核是用 C 语言写成的，但却往往包含很多面向对象的思想，体现在具体实现上，一般都使用组合的方法来实现类似于 C++ 的继承。

下图是 bus-driver-device 的基本组织形式:



所有的 device 以树的形式进行组织，树叶往往表示具体的设备，树干上的各个节点一般都是为了方便组织 device，比如/sys/devices/platform 这个 device 就是 platform 模型为了统一各个 platform device 而建立的。

所有的 drivers 和 devices 都组织在 bus 上的 drivers 和 devices 链表上

下面我们来说明 linux 设备模型中各个对象的含义:

bus

bus 即总线，管理着 device 和 driver, 在 device 和 driver 间进行配对等。

- 它有一个 device 列表

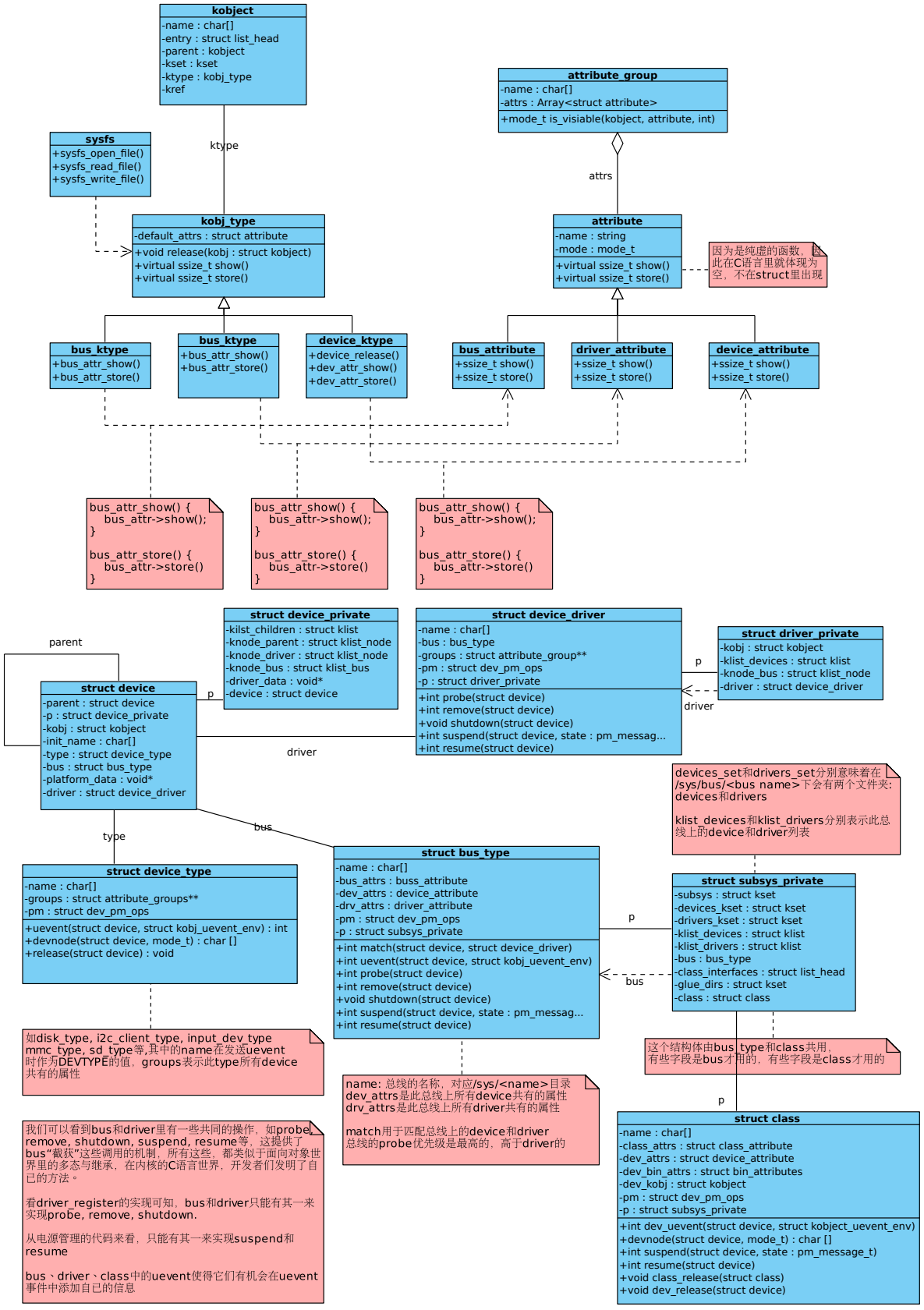


Figure 4.1: Linux 设备模型数据结构

- 它有一个 driver 列表
- 当 device 或 driver 挂接到总线上时，在 device 和 driver 之间进行配对 (probe)
- 它有自已的属性
- 它管理着同一总线上所有 device 所共有的属性
- 它管理着同一总线上所有 driver 所共有的属性

driver

driver 是一个软件抽象, 代表着对设备的操作，负责探测、控制 device。

- 它是驱动开发的主要工作
- 它有一个 device 列表
- 它有自已的属性
- 它管理着同一 driver 中所有 device 所共有的属性
- 有新设备加入式，负责初始化设备 (probe)
- 对上层提供接口 (file_operations): 打开，关闭，ioctl, 读，写等

device

device 对应具体的物理设备 (也可以是软件虚拟的)，代表着资源的拥有者，这意味着

- 一般都有寄存器
- 可能支持 DMA，DMA 的起始地址，终止地址
- 可能是个中断源
- 因为一个 driver 可能管理多个 device，因此每个 device 都有些“私房钱”(自已的状态等), 可以通过 dev_set_drvdata 设置，使用 dev_get_drvdata 获取.

kobject

kobject 和 kobj_type 构成一个面向对象的类，表示 sysfs 中的一个目录，kobject 是静态属性的集合:

- 目录名即其 name 值，parent 即其父目录，如果 parent 为 NULL, 则父目录由 kset 指定，如果 parent 和 kset 都为 NULL，父目录为/sys
- 可能属于某个 kset,
- 可能属于某个 kobj_type
- 维护引用计数 (kref)

kobj_type

- kobj_type 处理 sysfs 目录下文件的读与写，这种读和写大部分时候只是一个代理 (delegate 设计模式)，真正的读写由各 attribute 具体实现。
- 负责相同 type 的 kobject 的 release
- 维护一些共有属性。

这种 delagate 模式的应用使得 sysfs 不必关心 attribute 的具体实现，bus, driver, device 等都有自己的 attribute 实现方式

attribute

以文件形式呈现在 sysfs 目录下，

- name 表示文件名，mode 表示权限
- show 和 store 一般情况下是由 kobj_type 代理的，不直接和 sysfs 打交道

实现 attribute 的 show 和 store 需要注意的地方:

- show 方法的 buffer 是一个固定大小为 PAGE_SIZE 字节的 buffer，实现时注意不要超过大小
- store 方法不能返回 0，如果在传进来的 buffer 里没有你感兴趣的数据，应该返回一个负数 (如 -EINVAL)。

kset

kset 的作用在目前的代码里有两点:

1. 类似于一个扁平族谱，可以通过其列表找到家族中每一个成员
2. 在 kobject 没有指定 parent 时，kset 就是它的 parent。

class

class 对应一类设备，如鼠标、rtc 等，类似于 kset，不同之处是 kset 更多关注的是 sysfs 的文件组织结构上，更能反映设备在物理上的连接关系，而 class 强调其逻辑功能，对系统上层隐藏其物理连接情况，是看问题的两个不同方面。

1. 与 sysfs 接口方面，其主要功能体现在 get_device_parent() 方法中决定设备的父目录，用于在 device 的 parent 未旨定 class 时将 device 按 class 组织在一起 (不过现有代码似乎没考虑 device 和 parent class 不一样时的情形)
2. class 的逻辑功能特性及系统上层软件联系较为紧密，class 一般情况下都被实现为一个中间层 (subsystem)，为上层软件提供服务，对下简化驱动编程。具体可参见 drivers/rtc/class.c 和 drivers/input/input.c 中关于 input 和 rtc 两个 class 的实现。
3. class 仅对 device 而言 (需在 device_register 之前指定 class), 对驱动及总线不适用。

4.2.2 uevent

uevent 的核心是 kobject_uevent 以及 dev_uevent，其中 dev_uevent 只对 devices 目录下的设备起作用并由 kobject_uevent 通过 uevent_ops 调用。

kobject_uevent 一般由 device_add/device_del 调用，其他需要向用户态发送设备状态信息的地方也可以使用。

uevent 通过 netlink 向用户层报告设备的添加及删除信息，在 Android 系统里，vold 负责处理这些信息。

想一看 uevent 究竟的朋友可以在 PC 机上使用 “udevadm monitor --property” 查看, 看了之后，再对照 kobject_uevent 及 dev_uevent，你就会明白了，这里就不再多说了。

在 android 上，打开 vold 的 debug 也能看到类似的信息。

4.2.3 sysfs 及 sysfs API

sysfs 是一个基于内存的文件系统，是内核向用户态展示信息的接口，和 procfs 相比，sysfs 的目录组织结构更合理，同时也反映了内核内部数据结构 (主要是 kobject) 的组织结构。sysfs 文件系统中的文件一般都可以简单地使用 cat 命令查看，但不要乱写 *o*。

针对 sysfs 的编程主要有两方面:

用户态呈现形式

sysfs 文件系件一般挂载在/sys 目录下，根目录一般包含以下文件夹:

- block: 块设备，目前我们主要的块设备有 iNand 及 SD/TF 卡等，分别对应 mmcblk0, mmcblk1 等。
- bus: 总线, 一般包含 hid，i2c, mmc, platform, scsi, sdio, serio, usb 等，在嵌入式 SoC 平台上，大部分设备位于 platform 总线上。
- class
- dev: 这个目录下的子目录是以设备的 major:minor 命名的，指向 devices 下的相应设备, 根据 major:minor 查找设备可以来这里查。
- devices: 这个是主要的目录，包含系统所有的设备。
- firmware: 没什么用
- fs: 系统当前用到的文件系统相关的信息
- kernel: 内核相关的信息
- module: 子目录以模块名命令，在这里可以查找到系统所有的模块信息，大部分时候，你到这个目录来是因为你需要操作模块的 module_param，位于 < 模块名 >/parameters 目录下。
- power: 电源管理相关的信息。

由于篇幅限制，在这里不详细介绍 sysfs 的整个目录结构。

内核态编程接口

sysfs 文件系统中提供了四类文件的创建与管理，分别是目录，普通文件，软链接文件，二进制文件。在实际应用中，较常用的是普通文件和目录。使用 sysfs 的优点在于不用写 ioctl 就能在运行时调整驱动的功能，使用 ioctl 方式往往还需要写一个短小的 C 语言程序，需要打开、操作、关闭设备，还要编译等，非常复杂，为了完成一些非常简单的 yes or no 操作去写个程序是不值当的，而使用 sysfs 没有这个问题，一般情况下只需要向相应的文件值 echo 一个值就搞定了。

sysfs 模块提供给外界的接口都放在 include/linux/sysfs.h 中，下面我们来简要描述我们常用的数据结构及接口 (去掉了一些不相关的代码)。

```
struct attribute {
        const char              *name;
        mode_t                  mode;
};

struct attribute_group {
        const char              *name;
        mode_t                  (*is_visible)(struct kobject *,
                                              struct attribute *, int);
        struct attribute        **attrs;
};

struct bin_attribute {
        struct attribute        attr;
        size_t                  size;
        void                    *private;
        ssize_t (*read)(struct file *, struct kobject *, struct bin_attribute *,
                        char *, loff_t, size_t);
        ssize_t (*write)(struct file *,struct kobject *, struct bin_attribute *,
                         char *, loff_t, size_t);
        int (*mmap)(struct file *, struct kobject *, struct bin_attribute *attr,
                    struct vm_area_struct *vma);
};
```

bin_attribute 不是很常用，bin_attribute 在 sysfs 中体现为一个二进制的文件。attribute 一般体现为文本文件 (由实现决定)。

attribute_group 的 name 属性如果指定了，则会新建一个目录，其所属属性将位于这个目录之下；is_visible 一般不实现，默认都是可见的。

不建议直接操作 attribute 结构，内核为我们定义了一些宏:

```
#define __ATTR(_name,_mode,_show,_store) { \
        .attr = {.name = __stringify(_name), .mode = _mode },   \
        .show   = _show,                                        \
        .store  = _store,                                       \
}

#define __ATTR_RO(_name) { \
        .attr   = { .name = __stringify(_name), .mode = 0444 }, \
        .show   = _name##_show,                                 \
}

#define __ATTR_NULL { .attr = { .name = NULL } }
```

__stringify 是内核提供的一个宏，看一下其实现就明白了:

```
#define __stringify_1(x...)     #x
#define __stringify(x...)       __stringify_1(x)
```

我们来看一下如何使有这些宏 (摘自 drivers/base/platform.c):

```
static ssize_t modalias_show(struct device *dev, struct device_attribute *a,
                             char *buf)
{
        struct platform_device  *pdev = to_platform_device(dev);
        int len = snprintf(buf, PAGE_SIZE, "platform:%s\n", pdev->name);

        return (len >= PAGE_SIZE) ? (PAGE_SIZE - 1) : len;
}

static struct device_attribute platform_dev_attrs[] = {
        __ATTR_RO(modalias),
        __ATTR_NULL,
};
```

不多做解释了，接下来让我们看一下和 bus_attribute, driver_attribute, device_attribute 相关的数据结构 (位于 include/linux/device.h 中):

```
struct bus_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct bus_type *bus, char *buf);
        ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
};

#define BUS_ATTR(_name, _mode, _show, _store)   \
struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)

extern int __must_check bus_create_file(struct bus_type *,
                                        struct bus_attribute *);
extern void bus_remove_file(struct bus_type *, struct bus_attribute *);


struct driver_attribute {
        struct attribute attr;
        ssize_t (*show)(struct device_driver *driver, char *buf);
        ssize_t (*store)(struct device_driver *driver, const char *buf,
                         size_t count);
};

#define DRIVER_ATTR(_name, _mode, _show, _store)        \
struct driver_attribute driver_attr_##_name =           \
        __ATTR(_name, _mode, _show, _store)

extern int __must_check driver_create_file(struct device_driver *driver,
                                        const struct driver_attribute *attr);
extern void driver_remove_file(struct device_driver *driver,
                               const struct driver_attribute *attr);

struct device_attribute {
        struct attribute        attr;
```

```
        ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                        char *buf);
        ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                         const char *buf, size_t count);
};

#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)

extern int __must_check device_create_file(struct device *device,
                                           const struct device_attribute *entry);
extern void device_remove_file(struct device *dev,
                               const struct device_attribute *attr);
extern int __must_check device_create_bin_file(struct device *dev,
                                           const struct bin_attribute *attr);
extern void device_remove_bin_file(struct device *dev,
                                   const struct bin_attribute *attr);
```

我们最常用的是 device_attribute，仍然地，不建议直接操作这个结构，而应该使用内核为我们提供的 device_attribute 宏。让我们来看一下 device_attribute 的使用 (摘自 drivers/i2c/chips/mma7660.c):

```
static ssize_t mma7660_attr_irq_balance_show(struct device *dev,struct device_attribute *attr, char *buf)
{
        return sprintf(buf, "en = %d dis = %d\n",
                        irq_en_count, irq_dis_count);
}

static DEVICE_ATTR(irq_balance, S_IRUSR | S_IRGRP | S_IROTH, mma7660_attr_irq_balance_show, NULL);


static ssize_t mma7660_attr_show_regs(struct device *dev, struct device_attribute *attr, char *buf) {
        .......
        return count;
}

static ssize_t mma7660_attr_set_regs(struct device *dev, struct device_attribute *attr,
                 const char *buffer, size_t count) {
        ......

        return count;
}

static DEVICE_ATTR(regs, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH, mma7660_attr_show_regs, mma7660_attr_set_regs);

static struct attribute *mma7660_attributes[] = {
        &dev_attr_irq_balance.attr,
        &dev_attr_regs.attr,
        NULL
};

static const struct attribute_group mma7660_attr_group = {
```

```
        .attrs = mma7660_attributes,
};
```

再次回到 include/linux/sysfs.h 中，说明一些常用函数的含义。

```
int __must_check sysfs_create_dir(struct kobject *kobj);
void sysfs_remove_dir(struct kobject *kobj);
int __must_check sysfs_rename_dir(struct kobject *kobj, const char *new_name);
int __must_check sysfs_move_dir(struct kobject *kobj,
                                struct kobject *new_parent_kobj);
```

sysfs_create_dir() 创建一个 kobject 对应的目录，目录名就是 kobj->name。

sysfs_remove_dir() 删除 kobj 对应的目录。删除一个目录也会相应地删除目录下的文件及子目录。

sysfs_rename_dir() 修改 kobj 对应目录的名称。

sysfs_move_dir() 将 kobj 对应的目录移到 new_parent_kobj 对应的目录下。

```
int __must_check sysfs_create_file(struct kobject *kobj,
                                   const struct attribute *attr);
int __must_check sysfs_create_files(struct kobject *kobj,
                                   const struct attribute **attr);
int __must_check sysfs_chmod_file(struct kobject *kobj,
                                  const struct attribute *attr, mode_t mode);
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
void sysfs_remove_files(struct kobject *kobj, const struct attribute **attr);
```

sysfs_create_file() 和 sysfs_remove_files() 在 kobj 对应的目录下创建 attr 对应的属性文件 (s)。

sysfs_chmod_file() 修改 attr 对应的属性文件的读写权限。

sysfs_remove_file() 和 sysfs_remove_files() 在 kobj 对应的目录下删除 attr 对应的属性文件 (s)。

```
int __must_check sysfs_create_bin_file(struct kobject *kobj,
                                       const struct bin_attribute *attr);
void sysfs_remove_bin_file(struct kobject *kobj,
                           const struct bin_attribute *attr);
```

sysfs_create_bin_file() 在 kobj 目录下创建 attr 对应的二进制属性文件。

sysfs_remove_bin_file() 在 kobj 目录下删除 attr 对应的二进制属性文件。

```
int __must_check sysfs_create_link(struct kobject *kobj, struct kobject *target,
                                   const char *name);
int __must_check sysfs_create_link_nowarn(struct kobject *kobj,
                                          struct kobject *target,
                                          const char *name);
void sysfs_remove_link(struct kobject *kobj, const char *name);

int sysfs_rename_link(struct kobject *kobj, struct kobject *target,
                      const char *old_name, const char *new_name);

void sysfs_delete_link(struct kobject *dir, struct kobject *targ,
                      const char *name);
```

sysfs_create_link() 在 kobj 目录下创建指向 target 目录的软链接，name 为软链接文件名称。

sysfs_create_link_nowarn() 与 sysfs_create_link() 功能相同，只是在软链接文件已存在时不会出现警告。

sysfs_remove_link() 删除 kobj 目录下名为 name 的软链接文件

sysfs_rename_link() 重命名 kobj 目录下指向 target 的 old_name 为 new_name

sysfs_delete_link() 删除 kobj 目录下指向 target 的 name(根据 name 是不是就足够了呢，为什么还有一个 targ 参数)。

```
int __must_check sysfs_create_group(struct kobject *kobj,
                                    const struct attribute_group *grp);
int sysfs_update_group(struct kobject *kobj,
                       const struct attribute_group *grp);
void sysfs_remove_group(struct kobject *kobj,
                        const struct attribute_group *grp);
int sysfs_add_file_to_group(struct kobject *kobj,
                        const struct attribute *attr, const char *group);
void sysfs_remove_file_from_group(struct kobject *kobj,
                        const struct attribute *attr, const char *group);
int sysfs_merge_group(struct kobject *kobj,
                       const struct attribute_group *grp);
void sysfs_unmerge_group(struct kobject *kobj,
                       const struct attribute_group *grp);
```

sysfs_create_group() 在 kobj 目录下创建一个属性集合，并显示集合中的属性文件。如果文件已存在，会报错。

sysfs_update_group() 在 kobj 目录下创建一个属性集合，并显示集合中的属性文件。文件已存在也不会报错。sysfs_update_group() 也用于 group 改动影响到文件显示时调用。

sysfs_remove_group() 在 kobj 目录下删除一个属性集合，并删除集合中的属性文件。

sysfs_add_file_to_group() 将一个属性 attr 加入 kobj 目录下已存在的的属性集合 group。

sysfs_remove_file_from_group() 将属性 attr 从 kobj 目录下的属性集合 group 中删除。

sysfs_merge_group() 将属性 grp 下的属性添加到 grp 目录下，要求 grp 的 name 属性赋值并且已经创建过这个目录。

sysfs_unmerge_group() 是 merge_group() 的反操作。

这些函数将是我们最常用的，特别是 sysfs_create_group.

来看一下实际使用方式 (摘自 drivers/i2c/chips/mma7660.c):

```
        err = sysfs_create_group(&client->dev.kobj, &mma7660_attr_group);
        if (err){
                printk(KERN_ERR "mma7660_probe: sysfs_create_group failed\n");
                goto err_create_sysfs;
        }
```

4.3 platform 驱动架构

在开始讲解前，需要说明总线的概念，我觉得 wikipedia 上的说明比较准确: 总线是指计算机组件间规范化的交换数据 (data) 的方式。明白了这个概念，我们就能理解总线与控制器的区别，一个总线系统一般由三部分组成，即控制器，总线，外设，不要将总线与控制器混淆，控制器其实是一个设备。以 mmc 为例，mmc 总线实现就是按照 MMC 规范发送 command 以及和文件系统层交互，并不和具体的 MMC 控制器相关。在 linux 系统里，往往把这种总线实现称之为一个子系统。

我使用了 “总线实现” 一词，而不是总线驱动，在 Linux 里，当我们称呼总线驱动时，一般是指特定平台的控制器的驱动，也就是说，是 driver，而不是 bus。

在 SoC 系统中集成的控制器、挂接在 SoC 内存空间的外设等都不依附于某条总线 (准确来说他们依附在 AHB/APB 等总线上)，但按照 Linux 的设备驱动模型, 设备和驱动都需要挂接在某条总线上，因此，

在 Linux 中使用一种虚拟的总线，称为 platform 总线，这种总线上的 driver 即控制器功能实现，device 即控制器的资源，有多少个控制器就有多少个 device，但驱动一般不随控制器的个数而变，再次强调 device 和 driver 的区别，前者代表着资源，后者代表着功能逻辑。

platform 驱动的实现位于 drivers/base/platform.c 中，感兴趣的话可以阅读其实现。网上关于 platform 驱动的资料较多，这里不再详述，和君正相关的代码将放在后续讲述君正 BSP 时再介绍。

4.4 电源管理

linux 的电源管理非常复杂，可以参见 Documentation/power 下的相关文档，在这里只做简单与必要的介绍。

linux 电源管理有两个入口点:

1. 用户态入口点是/sys/power/state，由 kernel/power/main.c 中的 state_store 实现，使用方式如下:

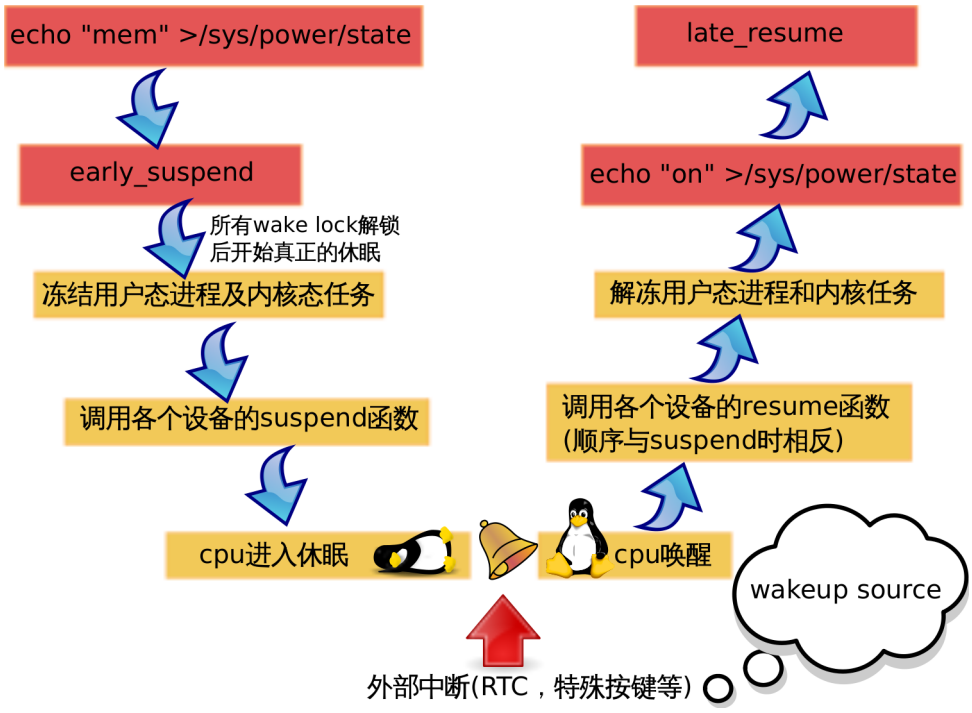
```
# echo "on" > /sys/power/state
# echo "mem" > /sys/power/state
```

on 启动 late resume, mem 启动 early suspend，在 Android 系统上，实际的休眠由 wake lock 控制。

尽管 state_store 里还有 disk 的实现，disk 和关机差不多，但会将内存中的内容写入外部存储，开机后会回到开机前的状态，目前我们不支持 disk 方式。

2. 内核态的入口点是 pm_suspend

这两个入口点最终都会调用 enter_state(state) 开始休眠唤醒流程，整个流程是非常复杂的，大致如下图所示:



其中的 early suspend、late resume 和 wake lock 是 Android 特有的，

- **Early suspend** - 这个机制定义了在 suspend 的早期，关闭显示屏的时候，一些和显示屏相关的设备，比如背光、重力感应器和触摸屏等设备都应该被关掉，但是此时系统可能还有持有 wake lock 的任务在运行，如音乐播放，电话，或者扫描 sd 卡上的文件等，这个时候整个系统还不能进入真正睡眠，直到所有的 wake lock 都没释放。在嵌入式设备中，背光是一个很大的电源消耗，所以 android 加入了这种机制。
- **Late resume** - 这个机制定义了在 resume 的后期，也就是唤醒源已经将处理器唤醒，标准 linux 的唤醒流程已经走完了，在 android 上层系统识别出这个物理上的唤醒源是上层定义的，那么上层将会发出 late resume 的命令给下层，这个时候将会调用相关设备注册的 late resume 回调函数。
- **Wake lock** - wakelock 在 android 的电源管理系统中扮演一个核心的角色，wakelock 是一种锁的机制, 只要有 task 拿着这个锁, 系统就无法进入休眠, 可以被用户态进程和内核线程获得。这个锁可以是有超时的或者是没有超时的, 超时的锁会在时间过去以后自动解锁。如果没有锁了或者超时了, 内核就会启动标准 linux 的那套休眠机制机制来进入休眠。

由于本文定位于入门介绍性的，这里不具体介绍 suspend/resume 及 wake lock 的使用，网上有很多资料。

TODO: 理解 wakeup source

4.4.1 电源管理的调试

串口信息打印

调试过程中我们往往依赖串口打印，默认情况下，串口在冻结进程后会关闭，我们可以在内核命令行中指定 no_console_suspend 保持串口打开，这样直到关闭 CPU 之前串口都可以使用。

事实上，只是指定 no_console_suspend 还是不够的，在休眠过程中，我们会为所有 GPIO 指定一个状态，如果串口相关的 GPIO 不再是功能 pin 了，串口是不会有打印的，因此如果碰到没有输出的情况，注意检查相关的 pin 是不是作为串口功能 pin 使用。

事实上，就算不指定 no_console_suspend，我们也能在 resume 后看到串口输出，在 console suspend 的时候，printk 的输出只放到了环形缓冲区里，并不真正输出到串口，系统唤醒后才会输出。

关于 P0

关不关 P0 除了功耗上的区别，在休眠唤醒时的路径也不同，如果需要 P0，需要先将休眠后的执行地址保存到 Scratch pad register 里，唤醒流程其实和正常开机差不多，都要经过 bootrom, 然后 bootloader，此时我们可以通过判断 CPM 的 Reset Status Register(RSR) 来确定起动原因，其中的 P0R 表示是从休眠唤醒的，直接跳转到 Scratch Pad Register 中记录的地址执行。

4.5 如何编写驱动

编写驱动看上去是件很复杂的工作，但其实不然，I2C, MMC 等驱动都有自己的一套框架，可以从现有的驱动直接把框架搬过来，然后专注于驱动功能的编写。

理解驱动在 linux 系统中的位置以及工作机制，对于编写功能代码有很大的帮助，也能更得心应手，没有什么事是困难的，没有什么事是你做不来的，只要你愿意，你可以做好任何事情。

在后面专门介绍各个驱动的章节中会详细介绍驱动原理及框架，这里暂不详述。

Chapter 5

君正 BSP 概述

在前面我们介绍过 BSP 的概念，它是介于硬件平台和系统核心之间的中间层软件，主要由三部分组成:

- 系统核心支持，包括 CPU 的初始化，中断管理，时钟源管理等，Linux 称之为 Platform 支持。
- 外设驱动
- 板级配置

下面我们来介绍君正在这三方面相关代码:

5.1 Ingenic Platform

Platform 实现主要是提供一些接口，供核心层在适当的时候调用，先来看一下代码结构。

5.1.1 代码结构

Linux-MIPS 相关的代码位于 arch/mips 目录下，来看一下 arch/mips/Kbuild 文件 (注: Linux 下每个 architecture 都有一个 kbuild 文件，如果不了解这个的话，考虑复习一下 Kbuild 系统):

```
1 ......
2 # platform specific definitions
3 include arch/mips/Kbuild.platforms
4 obj-y := $(platform-y)
5 ......
```

Platform 相关的放在 Kbuild.platforms 中，从 “obj-y := $(platform-y)” 可知，需要定义一个 platform-$(CONFIG_XXX) 的变量

Kbuild.platforms 由两部分组成:

```
1 ...
2 platforms += xxx
3 ...
4 include $(patsubst %, $(srctree)/arch/mips/%/Platform, $(platforms)
```

这意味着需要: 1) 在 arch/mips/有一个 xxx 目录; 2) 在 xxx 目录下有一个 Platform 文件，因此添加 Jz4770 平台的支持需要以下步骤:

1. 在 arch/mips 下新建一个目录，如 jz4770
2. 在 jz4770 目录下新建一个 Platform 文件, 其内容可以参照其他 Platform 的, 内容如下:

```
1 platform-$(CONFIG_SOC_JZ4770)   += jz4770/
2 cflags-$(CONFIG_SOC_JZ4770)     += -I$(srctree)/arch/mips/jz4770/include/
3 cflags-$(CONFIG_SOC_JZ4770)     += -I$(srctree)/arch/mips/jz4770/boards/
4 cflags-$(CONFIG_SOC_JZ4770)     += -I$(srctree)/arch/mips/include/asm/mach-xburst/
```

```
cflags-$(CONFIG_SOC_JZ4770)     += -I$(srctree)/drivers/video/include/
load-$(CONFIG_SOC_JZ4770)       += 0xffffffff80010000
```

3. 修改 Kbuild.platforms, 追加一个 platforms:

```
platforms += jz4770
```

jz4770/Platform 的内容决定了 Platform 相关的目录结构:

- 源码位于 jz4770 目录下
- 供整个系统查找头文件路径位于 arch/mips/jz4770/include, arch/mips/jz4770/boards, arch/mips/include/asm/mach-xburst, drivers/video/include 目录下
- 内核加载地址, 这里是 0xffffffff80010000

至于 jz4770 目录，里面的内容可以任由我们组织，目前我们的目录结构如下:

```
jz4770
├── boards: 板级文件，每个板级一个目录
│   ├── ......
│   ├── h700: H700 的目录，介绍板级配置时再描述
│   ├── ......
├── common: 所有基于 Jz4770 的板级公共的逻辑
│   ├── board_device_id.c: 垃圾文件，别去管他
│   ├── cpm.c: 对应 CPU 的 CPM 模块
│   ├── cpufreq.c: cpufreq 是 Linux 的一种变频机制，
│   │              根据系统的负载动态改变 cpu 的运行频率，目前我们暂未启用
│   ├── dma.c: 对应 CPU 的 DMA 模块，实现了一些常用功能供其他模块使用
│   │         除了 jz_request_dma，其他的代码几乎无人使用，急待整理
│   ├── fpu.c: FPU 相关
│   ├── gpiolib.c: gpiolib 相关，gpiolib 是内核对 GPIO 的一种抽象，
│   │              对系统其他部分屏蔽掉 GPIO 硬件相关细节，
│   │              目前情况下，大部分时候我们直接使用了 __gpio_xxx 函数
│   │              但将来有可能会全部替换成 gpiolib，使驱动更为‘‘标准’’
│   ├── irq.c: 中断管理 (INTC)
│   ├── Makefile
│   ├── platform.c: platform 设备集中‘‘帖’’，几乎所有外设都是 platform 设备
│   ├── pm.c: 电源管理相关
│   ├── proc.c: 表现为/proc/jz 目录，一些用于调试的接口
│   ├── prom.c: 实现 prom_init，提供底层板子上的一些重要信息 (取系统固件传递的参数，
│   │           io 的映射方式等)，内存初始化 (大小，分布)，
│   │          还包括提供早期的信息输出接口 (如串口) 以方便调试 (用于 early printk).
│   ├── reset.c: reboot 以及关机相关的代码
│   ├── setup.c: 实现 plat_mem_setup，本该是内存初始化相关，
│   │            但被初始化串口，打开一些关键时钟的代码占据
│   ├── sleep.S: CPU 休眠唤醒例程
```

```
|   └── time.c: 系统时钟源 (clock source: TCU, OST) 管理
├── include: Jz4770 相关的头文件
├── Kconfig
├── Makefile
```

CONFIG_SOC_JZ4770 的定义位于 arch/mips/Kconfig 中:

```
config SOC_JZ4770
        bool "Ingenic JZ4770 based machines"
        select JZSOC
        select JZRISC
        select CPU_SUPPORTS_HIGHMEM
        select SYS_SUPPORTS_HIGHMEM
        select SYS_HAS_CPU_MIPS32_R1
        select SYS_SUPPORTS_32BIT_KERNEL
        select SYS_SUPPORTS_LITTLE_ENDIAN
        select DMA_NONCOHERENT
        select GENERIC_GPIO
        select ARCH_REQUIRE_GPIOLIB
        select SYS_HAS_EARLY_PRINTK
```

相关选项的意义请看 Kconfig 中的帮助信息。

在 arch/mips/jz4770 下也有 Kconfig, 因此需要告诉 Kbuild 系统, 同样在 arch/mips/Kconfig, 添加了以下内容:

```
source "arch/mips/jz4770/Kconfig"
```

至此，通过 Kconfig 和 Makefile，我们的代码被串联进了 Linux 内核，成为一体。

但是我们还差一步，我们需要让内核识别我们的 CPU，探测 cpu 的任务由 cpu_probe 函数完成，位于 arch/mips/kernel/cpu-probe.c 中，

从其实现来看，我们需要定义一个 PRID_COMP_XXX, 即 Company ID，还要实现一个 cpu_prob_xxx

君正的 Company ID 是 0xd1，在 arch/mips/include/asm/cpu.h 中增加了以下定义:

```
#define PRID_COMP_INGENIC       0xd10000
......
#define PRID_IMP_JZRISC         0x0200
```

PRID_IMPL_JZRISC 是 Processor ID。

5.1.2 内核启动流程简述

从代码可知，我们主要提供以下接口供核心层调用:

- plat_mem_setup
- plat_time_init
- reset.c 中的开关机接口，在 setup.c 中赋值给了_machine_restart，_machine_halt, pm_power_off, 这些都是函数指针，由 arch/mips/kernel/reset.c 回调
- prom_init
- prom_putchar: 供 early printk 使用
- prom_getcmdline: 供 early printk 使用

- prom_free_prom_memory: 在 free_initmem 时调用，用于回收只用于系统启动的一些内存，目前实现为空，我们没什么可回收的。
- get_system_type: 用于 cat /proc/cpuinfo 时显示 system type,
- arch_init_irq: 用于初始化中断处理相关的数据结构
- plat_irq_dispatch: 来自 INTC 的中断分发接口，由 handle_int 调用

代码中还使用 arch_initcall 之类的代码，要切实理解这些代码何时被调用，我们需要大致理解 Linux-MIPS 的启动流程,。

Linux-MIPS 启动流程

首先是找到内核的入口点，Linux 编译时使用的 Linker Script 位于 arch/mips/kernel/vmlinux.lds, 是由 vmlinux.lds.S 生成的，从这个文件中我们可以获取两点信息:

1. ENTRY(kernel_entry): 说明内核的入口点是一个叫 kernel_entry 的函数
2. . = VMLINUX_LOAD_ADDRESS: 加载地址是 VMLINUX_LOAD_ADDRESS

在讲述 kernel_entry 之前，我们先简要讲述一下 VMLINUX_LOAD_ADDRESS 的由来:
查看 Makefile 可知 vmlinux.lds.S 的编译指定了一个 CPPFLAGS:

```
CPPFLAGS_vmlinux.lds            := $(KBUILD_CFLAGS)
```

其中 KBUILD_CFLAGS 在 arch/mips/Makefile 里生成:

```
KBUILD_CPPFLAGS += -D"VMLINUX_LOAD_ADDRESS=$(load-y)"
```

其中的 load-y 在 arch/mips/jz4770/Platform 里指定，当前为 0xffffffff80010000。
来看 kernel_entry 的实现在 arch/mips/kernel/head.S 中，去掉了不相关的代码):

```
NESTED(kernel_entry, 16, sp)                    # kernel entry point

        kernel_entry_setup                      # cpu specific setup

        setup_c0_status_pri

        /* We might not get launched at the address the kernel is linked to,
           so we jump there.  */
        PTR_LA  t0, 0f
        jr      t0
0:

        PTR_LA          t0, __bss_start         # clear .bss
        LONG_S          zero, (t0)
        PTR_LA          t1, __bss_stop - LONGSIZE
1:
        PTR_ADDIU       t0, LONGSIZE
        LONG_S          zero, (t0)
        bne             t0, t1, 1b

        LONG_S          a0, fw_arg0             # firmware arguments
        LONG_S          a1, fw_arg1
        LONG_S          a2, fw_arg2
```

```
24      LONG_S          a3, fw_arg3
25
26      MTC0            zero, CP0_CONTEXT       # clear context register
27      PTR_LA          $28, init_thread_union
28      /* Set the SP after an empty pt_regs.  */
29      PTR_LI          sp, _THREAD_SIZE - 32 - PT_SIZE
30      PTR_ADDU        sp, $28
31      back_to_back_c0_hazard
32      set_saved_sp    sp, t0, t1
33      PTR_SUBU        sp, 4 * SZREG           # init stack pointer
34
35      j               start_kernel
36      END(kernel_entry)
```

初始化.bss 段之后，程序将 a0, a1, a2, a3 存入了 fw_arg0, fw_arg1, fw_arg2, fw_arg3 四个变量，这意味着 kernel_entry 可以传四个参数, 这些参数的格式其实没有特别的规定，因为是由我们的 prom_init 解析的，但内核的代码里有一些约定俗成的规定:

- fw_arg0: argc
- fw_arg1: argv
- fw_arg2: envp, 环境变量
- fw_arg3: 很少用到

其中的 init_thread_union 就是著名的第 0 号进程，往后的代码可以认为是运行在这个进程中的，kernel_entry 最后调用了 start_kernel, 定义在 kernel/initmain.c 中, 感兴趣的可以接着往下分析。

5.2 外设驱动概况

下面是当前和君正相关的外设驱动的概况:

- 串口驱动为 drivers/tty/serial/jz_uart.c
- MSC 驱动位于 drivers/mmc/host/jz_mmc 目录下
- i2c 总线驱动位于 drivers/i2c/busses 目录下
- LCD 驱动位于 drivers/video 目录下
- 音频驱动位于 sound/oss 目录下
- 触摸屏驱动位于 drivers/input/touchscreen
- usb 驱动位于 drivers/usb 下, 主要代码在 drives/usb/musb
- 按键驱动 (power-on, 音量 +, 音量 -等) 位于 drivers/input/keyboard 下, 主要文件为 jz_gpio_keypad.c
- RTC 驱动位于 drivers/rtc

后面的章节中我们将对这些驱动进行详细的分析。

5.3 板级配置

增加一个板级配置需要在 arch/mips/jz4770/boards 目录下新建一个目录，如 h700，然后在 arch/mips/jz4770/boards/Makefile 中增加如下一行:

```
obj-$(CONFIG_JZ4770_H700)               += h700/
```

还需要在 arch/mips/jz4770/Kconfig 的 “JZ4770 board selection” choice 中增加以下项:

```
config JZ4770_H700
        bool "BBK JZ4770 H700 board"
        select DMA_NONCOHERENT
        select SYS_HAS_CPU_MIPS32_R1
        select SYS_SUPPORTS_32BIT_KERNEL
        select SYS_SUPPORTS_LITTLE_ENDIAN
        select SYS_HAS_EARLY_PRINTK
```

这些项有些和 CONFIG_SOC_JZ4770 重复。

h700 的目录结构可以是任意的，当前 h700 的目录结构如下:

```
arch/mips/jz4770/boards/h700
├── h700-audio.c: 配合 sound/oss 下的音频驱动，提供板级回调函数及
│                 一些板级配置 (如从 Mic1 还是 Mic2 录音，最大音量限制等)
├── h700.h: 主要是板级相关的 GPIO 的宏，因为背光配置是宏实现，也位于这个文件
├── h700-lcd.c: LCD 相关的板级配置
├── h700-misc.c: 主要是 TF 卡及 iNand 的配置，还包括其他一些配置
│                (显然，显示和音频是大头，TF 和 iNand 成 misc 类别了)
├── h700-pm.c: 电源管理相关，主要是休眠时各个 gpio 状态的配置
│              (休眠时所有 gpio 必须手工指定，不允许未知状态的 gpio 存在)
└── Makefile
```

Chapter 6

Git Quick Reference

git 很难学，命令好多好多，堪比 emacs，因此学习方法上也大致一样：先大致学习一遍，然后从实际使用中慢慢积累经验。

首先要知道如何看帮助，git 的帮助还不错的，相关命令如下：

git help <command>: or

git <command> –help: Show help for a *command*.

通常的工作流程是:

1. 创建本地 (local) 库
2. 更新 (update) 共享 (shared/remote) 库里的内容到本地
3. Make changes(new, modify, remove, rename, etc.), 并将一个 topic 的阶段性工作放到 staging area;
4. commit changes，一个 commit 一个主题，切忌一次 commit 涉及到多个主题
5. review(amend) commits，看一下都 commit 了什么，是否可以删了某些 commit(因为没用了), 是否可以合并某些 commit，各个 commit 的顺序是否需要调整一下以便更合逻辑。
6. push 本地修改到共享库里

下面我们就这些内容依次进行讲解。

在讲解前，首先介绍一下 staging area,staging area 是一个暂存区 (cache)，比如说为了改个 bug, 某个文件你要修改三处 (想象成分三个阶段 [stage] 修改)，每修改一处，测试一下，ok 了就可以放到 staging area，再修改第二处，发现改得不对，checkout 一下就可以回到第一个 stage 了。

6.1 Create

From existing data:

```
cd ~/projects/myprojects
git init
git add .
```

以 push 方式向 gitolite 新增 repository 还需要执行以下命令:

```
git remote add origin <url>
```

然后就可以用 git push 到 gitolite 了。这样新建的 repo 在 pull 的时候会报错，因为 branch master 的设置没有 (查看.git/config 可以看到只有 merge “ours” 这样的 section), 使用以下命令：

```
$ git config --remove-section merge.ours
$ git config branch.master.remote origin
$ git config branch.master.merge refs/heads/master
```

From existing repo:

```
git clone ~/existing/repo ~/new/repo
git clone git://host.org/project.git ~/new/repo
git clone ssh://you@host.org/project.git ~/new/repo
git clone http://git.software.org/project.git ~/new/repo
```

6.2 Update

git fetch: (不带任何参数)fetch latest changes from origin(but does not merge them).

fetch 下来的 commit 存放在 remotes/origin/master 里, 可以 git checkout remotes/origin/master 切换到 remote 的 master 进行查看，如果没有问题，用 git checkout -f master 切换到 local master, 然后 git merge remotes/origin/master 进行 merge

git pull: fetch + merge

git am -3 patch.mbox:

Apply a patch that someone send you(by email)

incase of a conflict, resolve and use git am –resolved.

git apply –reject patch.diff:

打 patch，如果失败，生成 *.rej 文件

6.3 git diff

```
git diff [<common diff options>] <commit>{0,2} [--] [<path>...]
```

git diff: 比较当前 work tree 和 staging area

git diff --cached: 比较 staging area 与未修改前 repository 的区别

git diff HEAD: 比较当前 work tree 与未修改前 repository 的区别 (git diff + git diff –cached)

git diff HEAD~1: 与 HEAD 的上一个 commit 比较

git diff other-branch:
比较当前 work tree 与 other-branch 的区别

git diff master: 比较当前 work tree 和 master 的区别

git diff tag1 tag2: tag1 和 tag2 的 diff

git diff tag1:file1 tag2:file2:
tag1 中的 file1 和 tag2 中的 file2 的区别

git diff --stat: 以汇总的形式显示，例如:

```
file1.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

可以将产生的 patch 重定向到文件中并发送给其他人:

```
git diff -a tag1 tag2 >some.patch
```

使用 -a 选项能确保二进制文件也会包含在 some.patch 中

接收方必须使用 p1 来打 patch:

```
patch -Np1 -i path/to/some.patch
```

6.4 staging

git add .: 将当前目录下所有 (修改过或新增的) 文件先暂存到 staging area

git add filename: 将文件 filename(修改过的或新增的) 暂存到 staging area

git add -u: 将修改过的文件暂到到 staging area，新增的文件不暂存

git add -i: 进入互动模式

6.5 删除文件

git rm filename: 删除文件 (注意: 文件会立即从所在的文件夹消失)

注: 删起来简单，如果 stage 了，要恢复比较麻烦，以 file1.txt 为例，需分两步

1. git reset HEAD – file1.txt
2. git checkout – file1.txt

6.6 还原已删除的文件

git ls-files -d: 查看已删除的文件列表

git ls-files -d | xargs git checkout --:
将已删除的文件全部还原 (注：如果 stage 了需要先 reset)

6.7 文件重命名

git mv filename new-filename:
需要注意: 本地的文件立即被重命名，无须 commit.

而且要恢复比 rm 更麻烦，以将 file1.txt 重命名 file1-1.txt 为例，需三步:

1. git reset HEAD – file1.txt file1-1.txt
2. git checkout – file1.txt
3. rm file1-1.txt

请注意 file1-1.txt 先 reset 后 rm 了

6.8 查看状态

git status: (nothing to explain)

6.9 commit

git commit: 这是推荐的用法。会打开 editor 让你编辑 commit message. some of Git's viewing tools need commit messages in the following format:

```
A brief one-line summary(short message, less than 50 characters)
<blank line>
Details about the commit
```

git commit -m 'commit message':
commit message 由 -m 指定

git commit -a -m 'commit message':
将所有修改过的文件都 commit.

注: 新增的档案还是需要先 add

git commit -a -v: 会在打开的 editor 里显示 git diff HEAD 的结果

若发现提交有问题，比如说 commit message 写得不是很对，可以使用 –amend 进行修改后重新 commit.

```
git commit --amend
```

此命令将使用当前的暂存区域快照提交。如果刚才提交完没有作任何改动，直接运行此命令的话，相当于有机会重新编辑提交说明，但将要提交的文件快照和之前的一样。

启动文本编辑器后，会看到上次提交时的说明，编辑它确认没问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交。

如果刚才提交时忘了暂存某些修改，可以先补上暂存操作，然后再运行 –amend 提交:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

上面的三条命令最终只是产生一个提交，第二个提交命令修正了第一个的提交内容。

6.10 tag&branch

在 git 中，tag 和 branch 都只是一个标志 (相当于 C 指针)，一般对每个 release 版本做一个 tag，或者在你认为 (开发过程中) 重要的点做一个 tag, 要开发一个新功能或是新产口，就生成一个 branch，新建一个 branch 就好比在一个分叉的路口立一个路标，写着: 左边去往 master, 右边去往 new-branch.

git branch: 列出所有本地 branch(当前 branch 前有 * 号)

git brahch -a: 列出所有 branch(local + remote)

git branch new-branch:
由当前所在的 branch 新建 new-branch

git branch new-branch master:
由 master 分支生成 new-branch

git branch new-branch v1:
由 tag(v1) 生成 new-branch

git branch -d branch:
删除 branch

git branch -D branch:
强制删除 branch

git checkout branch:
切换当前 work tree 到 branch

git checkout -b new-branch other-branch:
相当于:

```
$ git branch new-branch other-branch
$ git checkout new-branch
```

other-branch 可以不给出，表示从当前 working tree 所在 branch 创建。

git branch -r: 列出所有 remote branch

git tag v1 ebff810: 从 commit ebff810c461ad1924fc422fd1d01db23d858773b 创建 tag

git tag -d v1: 删除 tag v1

git tag -l [match-pattern]:
列出所有 tag(可根据 pattern 过滤)

6.11 git checkout

注意:

- 当前 work tree 下有没有 commit 的修改时，不允许 checkout 到另一个 branch, 解决方法:
  1. 撤销所有修改: 手动一个一个删，或者使用 checkout -f
  2. stash(暂存)
- 当 checkout 不是用来切换 branch 时，是从 *staging area*里 checkout 的，因此，你无法执行 git checkout HEAD^ 这样的命令 (尽管执行时什么错也不报)

git checkout [-f] branch-name:
切换到 branch-name

git checkout maskter:
切换到 master

git checkout -b new-branch other-branch:
请看tag&branch中介绍。

git checkout [-f: filename] 还原文件 filename 到未修改前的状态 (相当于 svn revert)

git checkout HEAD .:
还原当前目录下所有文件

git checkout xxxxxxx .:
还原当前目录下所有文件到 commit xxxxxxx

git checkout –ours filename:
就像名字所预示的那样：checkout 我们的。这在有 merge conflict 的时候有用，

git checkout –theirs filename:
同上，这回是 checkout 他们的，呵

注: 请谨慎使用，一旦用了，本地的 merge conflict 现场就消失了。merge conflict 时，默认情况下得到的文件里有<<<<<<< 和>>>>>>> 等东东，你可以手动去选择你要留下的，如果你想你当前的，就直接用 –ours，如果想你另一个 branch 的，就 –theirs

6.12 git log

git log: 列出所有 log(使用 less 输出, 不会像 svn 那样狂刷屏)

注: 要显示特定 commit 的 log，请使用 git show

git log -all: 列出所有 log(含 branch)

git log filename: 列出文件 filename 相关的 log

git log dir: 列出目录 dir(及其下所有文件) 相关的 log

git log -S 'foo': 列出有 foo 这个字符串的 log

git log –no-merges: 不显示 merge 的 log

git log –since="2 weeks agon":
列出两星期前的 log

git log –pretty=oneline:
单行显示 (格式是: commit_hash< 空格 >short commit message)

git log –pretty=short:
只显示 short message.

所谓的 short message，请查看commit一节的说明。

git log –pretty=format:'%h was %an, %ar, message: %s':
以特定格式显示 (具体怎么显示自己试:-))

git log –pretty=format:'%h : %s' –graph:
会有简单的图形化文字，分支等

git log –pretty=format:'%h : %s' –topo-order –graph:
依照主分支排序

git log –pretty=format:'%h : %s' –date-order –graph:
依照时间排序

6.13 git show

通用格式是:

```
git show <commit>[:file]
```

git show ebff810: 查看 commit ebff810c461ad1924fc422fd1d01db23d858773b 的内容，显示时先显示 commit 的 log, 然后是 diff 的输出。

git show v1: 查看 tag(v1) 的修改内容。其实这相当于 git show < 打 tag 时所在的 commit>. 在 git 中，tag 只是一个指针而已。

git show v1:test.txt: 查看 tag(v1) 中的 test.txt 文件的内容

git show HEAD: show 最近的 commit

git show HEADˆ: show 前一个 commit

git show HEADˆˆ: show 前前一个 commit

git show HEAD~4: show 前前前前一个 commit(*o*)

6.14 reset(unstaging)&revert

git reset HEAD filename:
unstage file for commit(磁盘上文件的内容并不会改变，只是不 stage 了)

git reset –hard HEAD:
还原到 HEAD(所有本地修改都会消失)

git reset –hard HEAD˜3:

git reset –soft HEAD˜3:

git revert <commit ref>:

git commit -a –amend:
<TODO: 更多的 amend 相关用法 >

6.15 git grep

git grep "te" v1: 查看 tag(v1) 中是否有 "te" 字符串

git grep "te": 查看当前 work tree 中是否有 "te" 字符串

6.16 git stash

git stash: save to stash area
git stash list: 列出 stash area 都有什么
git stash pop: 从 stash area 取出最新的一笔，并移除，如果当前 worktree 有 unstage 的修改，此命令将会执行失败。
git stash pop: 取出最新的一笔 stash 暂存资料，但是不移除
git stash drop <stash name>:
删除一笔 stash <stash name>
git stash clear: 把 stash 都清掉

6.17 merge

有几种合并的模式:

- **straight merge**, 默认的合并模式，会有全部的被合并的 branch commit 记录加上一个 merge commit，看线图会有两条 parents 线，并保留所有 commit log.
- **squashed commit**, 压缩成一个 merge-commit, 不会有被合并的 log.
- **cherry-pick**, 只合并指定的 commit
- **rebase, 变更 branch 的分支点**, 找到要合并的两个 branch 的共同的祖先，然后只用要被 merge 的 branch 来 commit 一遍，然后再用目前 branch 再 commit 上支。这方式仅适合还没分享给别人的 local branch，因为等于砍掉重新 commit log.

git merge other-branch:
将 other-branch 合并到当前 work area. 若没有冲突会直接 commit。若需要解决冲突则会多一个 commit.
git merge –squash <branch-name>:
将另一个 branch 的 commit 合并为一笔，特别适合需要做实验的 fixes bug 或 new feature，最后只留结果。合并完不会帮你 commit，需手动再 commit 一次。
git cherry-pick 321d76f:
只合并特定其中一个 commit。如果不加 -n，会自动 commit, 如果要合并多个，可以加上 -n 指令，这样不会自动 commit，cherry-pick 所有需要的 commit 后，再 git commit 即可。

merge 过程中会产生 conflict，相关的有帮助的命令有：view the merge conflicts:

git diff:
git diff –base <filename>:
against base file
git diff –ours <filename>:
against your changes
git diff –theirs <filename>:
against other changes

discard conflict patch:

git reset –hard:
git rebbase –skip:

after resolving conflicts, merge with:

git add <conflict file>:

git rebase –continue:

6.18 remote

git remote:
git remote add new-branch http://git.example.org/project.git:
增加远端 repository 的 branch
git remote show: 列出现在有多少 repository
git remote rm new-branch:
删掉一个 remote branch
git remote update: 更新所有的 remote branch
git push origin :heads/<branch-name>:
删除 branch-name

6.19 Finding regressions

git bisect start: start bisect
git bisect good <commit>:
<commit> is the last working version
git bisect bad <commit>:
<commit> is a broken version
git bisect good/bad: mark as good or bad
git bisect visualize: to lunch gitk and mark it
git bisect reset: once you're done

6.20 check errors and cleanup repository

git fsck:

git gc –prune:

6.21 git config

常用的一些 config(~/.gitconfig)

```
[color]
        ui = auto
[user]
        name = Lutts Cao
        email = lutts.cao@gmail.com
[core]
        editor = vim
        excludesfile = /home/lutts/.gitignore
[alias]
        st = status
        ci = commit
        co = checkout
[color "diff"]
        whitespace = red reverse
```

你可以直接编辑 ~/.gitconfig，也可以使用 git config 命令，双引号内是 subsection，配置示例如下:

```
git config --global user.name "Lutts Cao"
git config --global color.diff.whitespace "red reverse"
```

还有一个重要的文件就是 gitignore, 其语法请 man gitignore 查看.

Part 3

Android 基础

Chapter 7

Android 基础

Part 4

Android 眼里的设备驱动

Chapter 8

MMC/SD/SDIO 驱动

Chapter 9

USB 驱动

Chapter 10

vold: android 上的 udev

Chapter 11

I2C 驱动

Chapter 12

触模屏驱动

Chapter 13

G-Sensor 驱动

Chapter 14

Android input 子系统

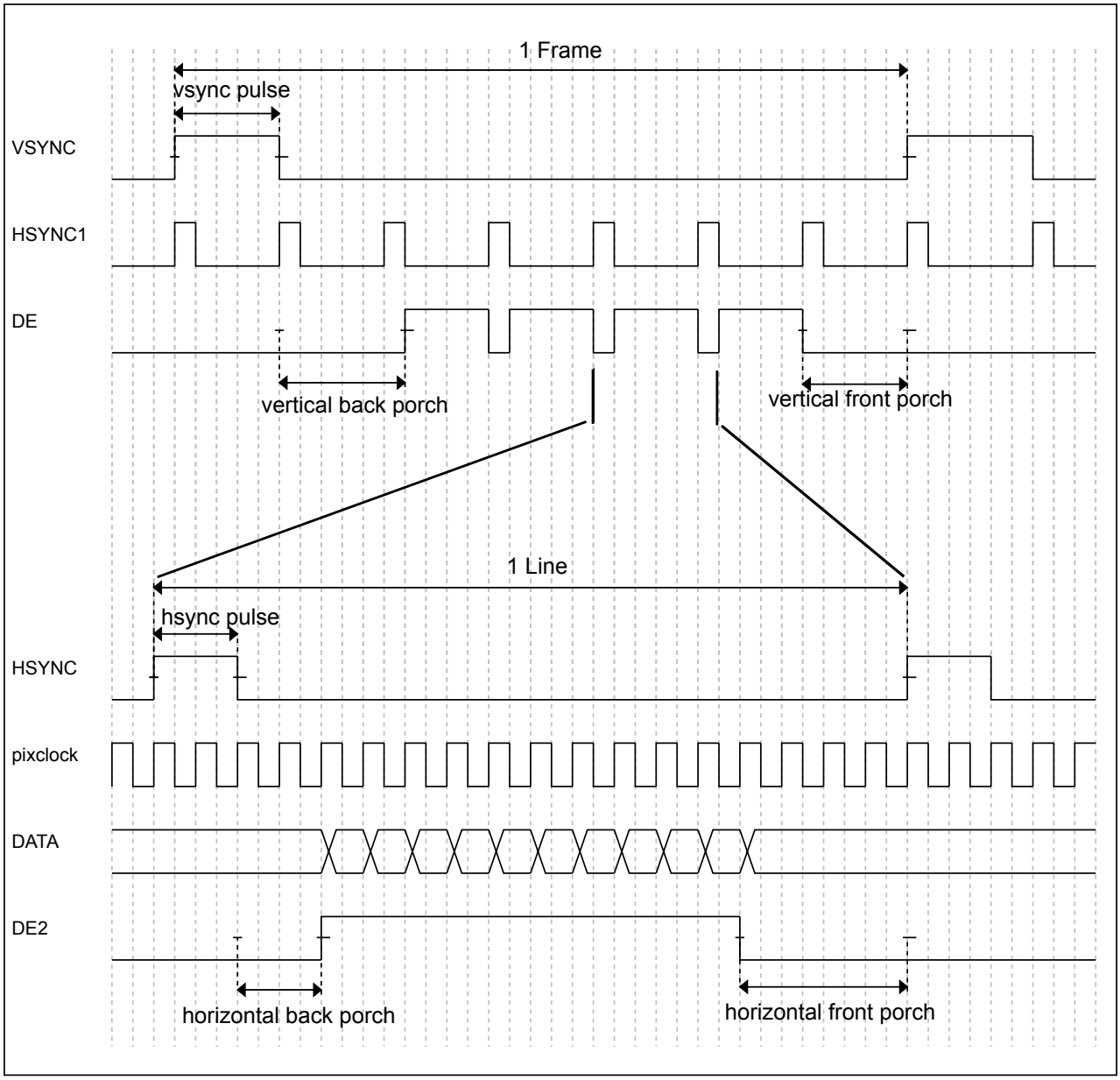
Chapter 15

LCD 驱动详解

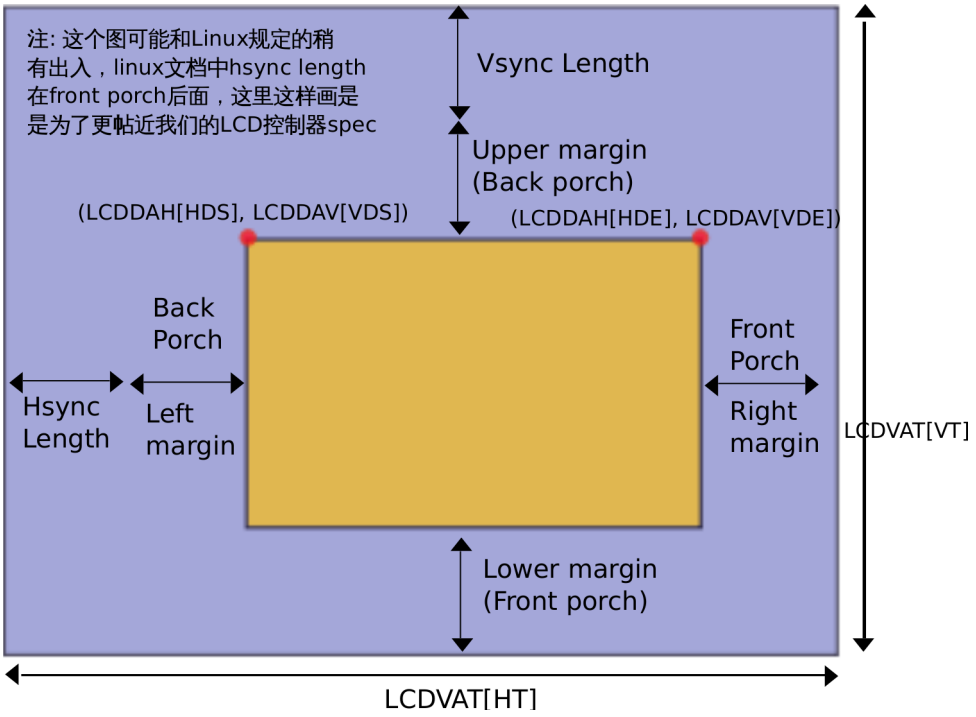
15.1 硬件常识

15.1.1 Generic 16-/18-/24-bit Parallel TFT Panel Timing Grame

下图是常见的并行 TTL TFT 屏的时序:



在 CEA 或 LCD 屏的规范中，一般都是以 back porch, front portch 表示 blanking 的，但 Linux 使用了另外的“术语”, 下图表示了 Linux 是如何理解这些时序的:



小知识

这个图对应了 fb_videomode 结构体。在 fb_var_screeninfo 中也有相应的字段。

15.1.2 君正 JZ4780 LCD 控制器 spec 导读

这里对 JZ4780 LCD 控制器的寄存器进行分类解析，希望能对理解 LCD 控制器的工作原理有帮助。

1. 特殊功能

(a) TVE 相关

- LCDCFG[TVEPEH]
- LCDCFG[TVEN]

(b) palette 相关

- LCDCFG[PALBP]

(c) 双 LCD 相关:

- LCDCDUALCTRL: 好家伙，全是内部信号名呀

(d) ENH 功能

- dither (4bit, 5bit, 6bit, 8bit dither)
- YCbCr to RGB (TODO: 怎么用)
- RGB to YCbCr, LCDENH_CSCCFG 可以配置具体的转换公式
- Saturation
- Visibility enhance enable(什么效果?)
- Hue
- Brightness
- contrast
- Gamma control

(e) 不得不说的秘密:

- 对 TFT panel 来说，LCD AHB clock 必须至少是 pixclock 的 1.5 倍
- 对 STN panel 来说，LCD AHB clock 必须至少是 pixclock 的 3 倍
- HDMI 只能用 LCDC0
- LVDS 只能用 LCDC1
- 普通 (TTL) 的 TFT 屏可用 LCDC0 或 1
- 但当两个控制器都使用的时候，不能都接 TTL TFT 屏
- 这样一来就有以下三种搭配:
  - 目前来说常用搭配: 0 — HDMI, 1 — LVDS
  - 0 — HDMI, 1 — TTL TFT
  - 0 — TTL TFT, 1 —- LVDS
- LCDOSDC 的第 16 位是个隐藏位，表示 one transfer two pixel，不知道是什么东东
- IPU writeback 只适用于 LCDC1, 参见 LCDCDUALCTRL 寄存器的设置

2. 和屏相关的设置:

(a) pin 脚设置

- LCDCFG[LCDPIN]: 决定 pin 是用于普通 LCD 还是 SLCD

- 16bit 并行模式时，可以选择使用 LCD_D[15:0] 还是 LCD_D[17:10] 和 LCD_D[8:1], 由 LCDCTL[PINMD] 设置
- 其他设置可参照第 7 节: LCD Controller Pin Mapping
- 四根特殊 pin 的配置也位于 LCDCFG 中，还有 LCDPS, LCDCLS, LCDSPL, LCDREV 四个寄存器
- hsync, vsync, pixclock,data enable 的极性 (LCDCFG[HSP], LCDCFG[PCP], LCDCFG[DEP], LCDCFG[VSP])

(b) bit 位数设置 (LCDCFG[MODE]):

- 16-/18-/24-bit Parallel TFT panel
  - LCDCFG[18/16]: 0—> 16bit, 1—>18bit
  - LCDCFG[24]: 24bit (目前最常见的类型)
- special TFT panel Mode1
- special TFT panel Mode2
- special TFT panel Mode3
- Non-interlaced TV out
- interlaced TV out
- 8bit serial TFT
- LCM (注: smart LCD)

(c) timing 设置:

- LCDVAT[VT], LCDVAT[HT]: 包含 blanking 在内的宽高
- LCDDAH: 水平有效区域的起始点 (back porch + hsync pulse width) 及终点 (back porch + hsync pulse width + width)
- LCDDAV：垂直有效区域的起始点 (back porch + vsync pulse width) 及终点
- LCDVSYNC: vsync 的起始点 (固定为 0) 与终点
- LCDHSYNC: hsync 的起始点与终点

3. 数据传输相关

(a) LCDCTRL[BST]: burst 大小 (4word, 8word, 16word, 32word, 64word)

(b) LCDCFG[RECOVER]: Auto recover when output FIFO under run

(c) LCDCTRL[OFUP]: output FIFO under protection(和上面那个配置有什么区别？)

(d) 中断使能及状态寄存器:

- LCDCTRL[SOFM] —> LCDSTATE[SOF]
- LCDCTRL[EOFM] —> LCDSTATE[EOF]
- LCDCTRL[OFUM] —> LCDSTATE[OUF]
- LCDCTRL[IFUM0](fifo0) —-> LCDSTATE[IFU0]
- LCDCTRL[IFUM1](fifo1) —-> LCDSTATE[IFU1]
- LCDCTRL[LDDM](disable done) —-> LCDSTATE[LDD]
- LCDCTRL[QDM](quick disable done) —-> LCDSTATE[QD]
- 中断时可获取产生中断的帧 buffer id: LCDIID

(e) 字节序配置相关: LCDCTRL[BEDN], LCDCTRL[PEDN]

(f) forground0/1 配置

- bpp:
  - LCDCTRL[BPP0]: 1bpp, 2bpp, 4bpp, 8bpp, 15/16bpp, 18/24bpp, 24bpp compressed, 30bpp(30??? 是不是写错了，是 32 吧?)
  - LCDOSDCTRL[BPP1]: 15/16bpp, 18/24bpp, 24bpp compressed, 30bpp
  - 16bpp 时，可选择 RGB565 或 RGB555 —> LCDOSDCTRL[RGB0], LCDOSDCTRL[RGB1]
- fg 在显示区的左上角位置: LCDXYP0, LCDXYP1
- fb 的大小: LCDSIZE0, LCDSIZE1
- 是否预乘 (premulti): LCDOSDC[PREMULTI1], LCDOSDC[PREMULTI0]
  至于什么是预乘，请上网查询 alpha blending(port-duffer) 相关的资料。
- 是否启用 SOF 中断: LCDOSDC[SOFM0/1] —> LCDOSDSTATE[SOF0/1]
- 是否启用 EOF 中断: LCDOSDC[EOFM0/1] —> LCDOSDSTATE[EOF0/1]
- 是否 enable 相应的 fg: LCDOSDC[F1EN], LCDOSDC[F0EN]
- 是否启用 alpha blending: LCDOSDC[ALPHAEN]
  - 是否使用一个全局 alpha 值: LCDOSDC[ALPHAMD0], LCDOSDC[ALPHAMD1]
  - 如果启用全局 alpha, 需配置 LCDALPHA 中的 alph 值
  - 如果不启用全局 alpha，alpha 值来看每个像素的 alpha 分量
  - alpha blending 时的 alpha 系数 (0, 1, alphaX, 1 - alphaX)
- color key: LCDKEY0, LCDKEY1
  - color key enable: LCDKEY0[KEYEN], LCDKEY1[KEYEN]
  - color key mode: LCDKEY0[KEYMD], LCDKEY1[KEYMD]

        * color key
        * mask color key
      – color (RGB)
    - LCDOSDC[OSDEN]

(g) background color 设置

- bg0: LCDBGC0 (RGB)
- bg1: LCDBGC1 (RGB)

(h) IPU 控制

- LCDOSDCTRL[IPU_CLKE]
- LCDIPUR: 配置 IPU 前一帧与后一帧之间的时间间隔

(i) 输出格式设置

- LCDCTRL[OUTRGB]: bpp16 时 RGB565, RGB555 选择
- TV 输出时，可以将 YUV444 转为 YUV422, LCDRGBC[422]
- RGB format enable: LCDRGBC[RGBFMT]
- 串行输出时:
  – RGB dummy:
    * 使能位: LCDRGBC[RGBDM]
    * 格式: LCDRGBC[DMM]: R-G-B-dummy or dummy-R-G-B
  – odd line RGB 顺序: LCDRGBC[OddRGB] —> RGB, RBG.....
  – Even line RGB 顺序: LCDRGBC[EvenRGB]

(j) 总线优先级配置: LCDCPCFG

(k) DMA 配置

- LCDDAx: 两个 LCD 控制器各有两个 LCDDA, 分别对应 fg0, fg1
  Descriptor 的格式是 (规格书上的过时了):

```
/**
 * @next: physical address of next frame descriptor
 * @databuf: physical address of buffer
 * @id: frame ID
 * @cmd: DMA command and buffer length(in word)
 * @offsize: DMA off size, in word
 * @page_width: DMA page width, in word
 * @cpos: smart LCD mode is commands' number, other is bpp,
 * premulti and position of foreground 0, 1
 * @desc_size: alpha and size of foreground 0, 1
 */
```

```
12                  struct jzfb_framedesc {
13                          uint32_t next;
14                          uint32_t databuf;
15                          uint32_t id;
16                          uint32_t cmd;
17                          uint32_t offsize;
18                          uint32_t page_width;
19                          uint32_t cpos;
20                          uint32_t desc_size;
21                  } __packed;
```

- LCDSAx: 两个 LCD 控制器各有两个 LCDSA, 分别对应 fg0, fg1，供调试用，知道当前 dma 的状态

Chapter 16

GPU 驱动

Chapter 17

Android surfaceflinger

Chapter 18

OSS 音频驱动

Chapter 19

Android audioflinger

Chapter 20

网络驱动

Chapter 21

Android wifi 及以太网管理

Chapter 22

电源管理

Chapter 23

背光管理

Chapter 24

Android 电源管理

Part 5

Android 进阶

Chapter 25

Android build system

Chapter 26

content provider

Chapter 27

skia

Chapter 28

调试方法进阶

Bibliography