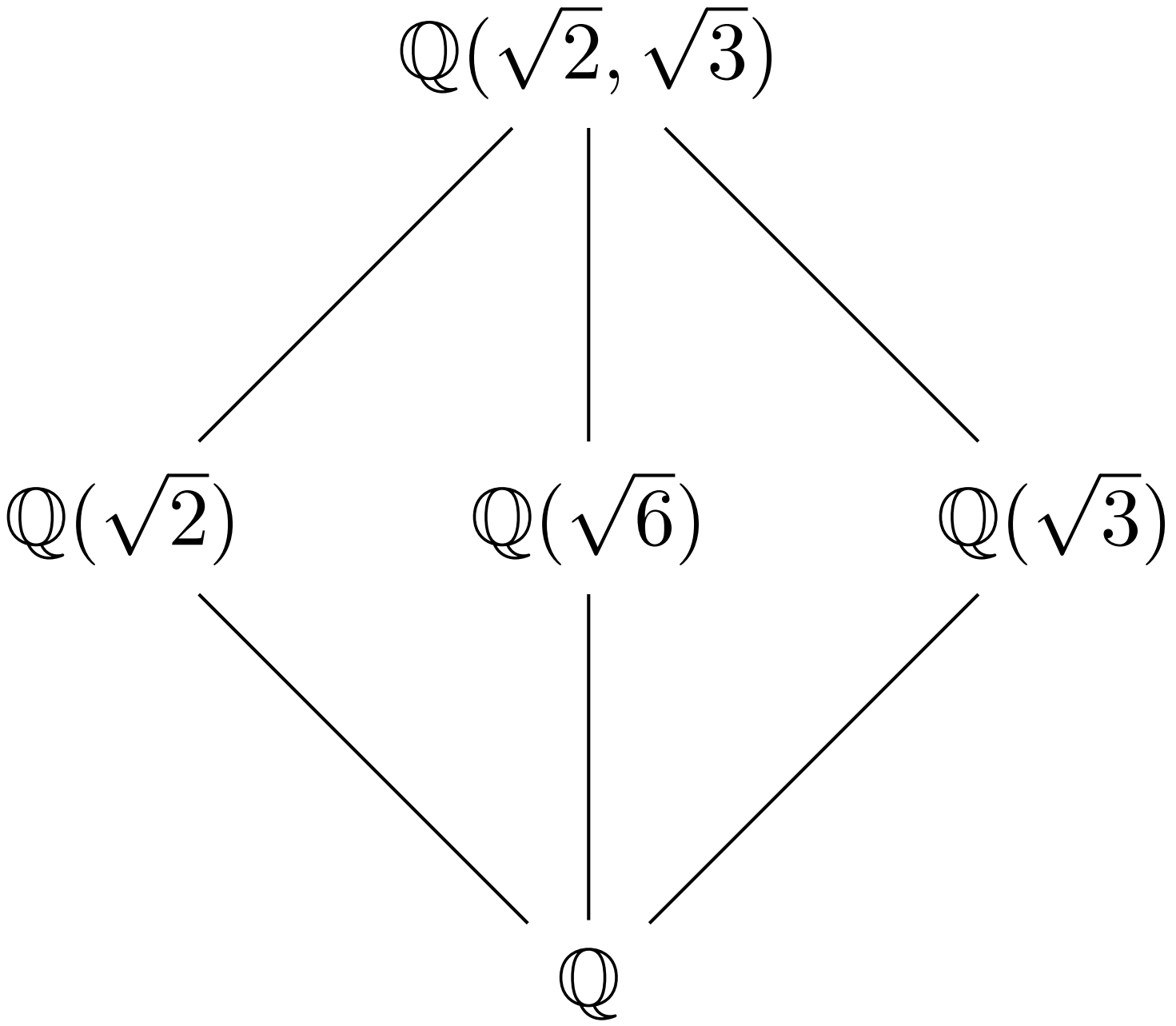

Linux **设备驱动开发教程**

作者：左元

目录

第一部分　Linux 驱动 1

第一章　简介 2
1.1 环境配置 . . . 2
1.1.1 获取源码 . . . 3
1.1.2 内核配置 . . . 4
1.1.3 构建自己的内核 . . . 4
1.2 内核代码编写风格 . . . 5
1.3 内核结构分配和初始化 . . . 6
1.4 类、对象、面向对象的编程 . . . 7

第二章　设备驱动程序基础 8
2.1 内核空间和用户空间 . . . 8
2.1.1 模块的概念 . . . 8
2.1.2 模块依赖 . . . 9
2.1.3 模块的加载和卸载 . . . 9
2.2 驱动程序框架 . . . 12
2.2.1 模块的入口点和出口点 . . . 12
2.2.2 模块信息 . . . 13
2.3 错误和消息打印 . . . 15
2.3.1 错误处理 . . . 16
2.3.2 处理空指针错误 . . . 19
2.3.3 消息打印——printk() . . . 20

第三章　一个完整的最简单的驱动 Demo 22
3.1 编写驱动程序 . . . 22
3.2 编写 Makefile . . . 24
3.3 编写应用程序 . . . 24

第四章　内核工具和辅助函数 27
4.1 理解宏 container_of . . . 27
4.2 链表 . . . 30
4.2.1 创建和初始化链表 . . . 31
4.2.2 创建链表节点 . . . 33
4.2.3 添加链表节点 . . . 33
4.2.4 删除链表节点 . . . 34
4.2.5 链表遍历 . . . 34

第五章　设备树的概念 36
5.1 设备树机制 . . . 36
5.1.1 命名约定 . . . 37
5.1.2 别名、标签和 phandle . . . 37

5.1.3 DT 编译器 . . . 38
5.2 表示和寻址设备 . . . 39
5.2.1 SPI 和 I^2C 寻址 . . . 40
5.2.2 平台设备寻址 . . . 41
5.3 处理资源 . . . 42
5.3.1 命名资源的概念 . . . 42
5.3.2 访问寄存器 . . . 43
5.3.3 处理中断 . . . 44
5.4 平台驱动程序和 DT . . . 45

第六章 字符设备驱动程序 47
6.1 scull 的设计 . . . 47
6.2 主设备号和次设备号 . . . 47
6.2.1 设备编号的内部表示 . . . 48
6.2.2 分配和释放设备编号 . . . 48
6.2.3 主编号的动态分配 . . . 49
6.3 一些重要的数据结构 . . . 52
6.3.1 文件操作 . . . 52
6.3.2 文件结构 . . . 54
6.3.3 inode 文件结构 . . . 55
6.4 注册字符设备 . . . 56
6.4.1 scull 中的设备注册 . . . 57
6.5 open 和 release . . . 57
6.5.1 open 方法 . . . 57
6.5.2 release 方法 . . . 59
6.6 scull 的内存使用 . . . 60
6.7 读和写 . . . 62
6.7.1 read 方法 . . . 63
6.7.2 write 方法 . . . 65
6.8 使用新设备 . . . 66

第七章 块设备驱动程序 68

第八章 I^2C 客户端驱动程序 69
8.1 驱动程序架构 . . . 69
8.1.1 i2c_driver 结构 . . . 69
8.2 总结 . . . 71

第二部分 Linux 移植 72

第九章 移植到 SD 卡 73
9.1 编译 U-Boot . . . 74
9.2 编译 Linux 内核 . . . 75
9.3 编译 Buildroot . . . 78
9.4 写入 SD 卡 . . . 78
9.5 将开发板作为 ssh 服务器来使用 . . . 79

9.6 安装无线网卡驱动 80

第三部分 驱动程序举例 81

第十章 V3S 按键驱动 82
10.1 查找设备树 82
10.2 查找驱动代码，准备测试程序 83
10.3 解决问题 85

第十一章 GPIO 驱动示例-1 87

第十二章 GPIO 驱动示例-2 89

第十三章 PWM 驱动示例-1 95

第十四章 PWM 驱动示例-2 97

第十五章 LCD 驱动示例 101

第十六章 I2C 驱动程序示例 108

第一部分

Linux 驱动

第一章 简介

Linux 内核是一种复杂、轻便、模块化并被广泛使用的软件。大约 80% 的服务器和全世界一半以上设备的嵌入式系统上运行着 Linux 内核。设备驱动程序在整个 Linux 系统中起着至关重要的作用。由于 Linux 已成为非常流行的操作系统。

设备驱动程序通过内核在用户空间和设备之间建立连接。

Linux 起源于芬兰的莱纳斯 · 托瓦尔兹（Linus Torvalds）在 1991 年凭个人爱好开创的一个项目。这个项目不断发展，至今全球有 1000 多名贡献者。现在，Linux 已经成为嵌入式系统和服务器的必选。内核作为操作系统的核心，其开发不是一件容易的事。

和其他操作系统相比，Linux 拥有更多的优点。

- 免费。
- 丰富的文档和社区支持。
- 跨平台移植。
- 源代码开放。
- 许多免费的开源软件。

本教程尽可能做到通用，但是仍然有些特殊的模块，比如设备树，目前在 x86 上没有完整实现。那么话题将专门针对 ARM 处理器，以及所有完全支持设备树的处理器。为什么选这两种架构？因为它们在桌面和服务器（x86）以及嵌入式系统（ARM）上得到广泛应用。

1.1 环境配置

在 Ubuntu 下，安装如下包。

安装一些包

```
$ sudo apt-get update
$ sudo apt-get install gawk wget git diffstat
unzip texinfo \
gcc-multilib build-essential chrpath socat
libsdl1.2-dev \
xterm ncurses-dev lzop
```

安装针对 ARM 体系结构的交叉编译器。

安装交叉编译器

```
$ sudo apt-get install gcc-arm-linux-gnueabihf
```

1.1.1 获取源码

在早期内核（2003 年前）中，使用奇偶数对版本进行编号：奇数是稳定版，偶数是不稳定版。随着 2.6 版本的发布，版本编号方案切换为 X.Y.Z 格式。

- X：代表实际的内核版本，也被称为主版本号，当有向后不兼容的 API 更改时，它会递增。
- Y：代表修订版本号，也被称作次版本号，在向后兼容的基础上增加新的功能后，它会递增。
- Z：代表补丁，表示与错误修订相关的版本。

这就是所谓的语义版本编号方案，这种方案一直持续到 2.6.39 版本；当 Linus Torvalds 决定将版本升级到 3.0 时，意味着语义版本编号在 2011 年正式结束，然后采用的是 X.Y 版本编号方案。

升级到 3.20 版时，Linus 认为不能再增加 Y，决定改用随意版本编号方案：当 Y 值增加到手脚并用也数不过来时就递增 X。这就是版本直接从 3.20 变化到 4.0 的原因。

现在内核使用的 X.Y 随意版本编号方案，这与语义版本编号无关。

> ! Linus：仁慈的独裁者。

源代码的组织

必须使用 Linus Torvald 的 Github 仓库。

下载源码

```
1 $ git clone https://github.com/torvalds/linux
2 $ git checkout 版本号 # 例如：git checkout v4.1
3 $ ls
```

内核中各文件夹的含义：

- arch/：Linux 内核是一个快速增长的工程，支持越来越多的体系结构。这意味着，内核尽可能通用。与体系结构相关的代码被分离出来，并放入此目录中。该目录包含与处理器相关的子目录，例如 alpha/、arm/、mips/、blackfin/等。
- block/：该目录包含块存储设备代码，实际上也就是 I/O 调度算法。
- crypto/：该目录包含密码 API 和加密算法代码。
- Documentation/：这应该是最受欢迎的目录。它包含不同内核框架和子系统所使用 API 的描述。在论坛发起提问之前，应该先看这里。
- drivers/：这是最重的目录，不断增加的设备驱动程序都被合并到这个目录，不同的子目录中包含不同的设备驱动程序。
- fs/：该目录包含内核支持的不同文件系统的实现，诸如 NTFS、FAT、ETX{2,3,4}、sysfs、procfs、NFS 等。
- include/：该目录包含内核头文件。
- init/：该目录包含初始化和启动代码。
- ipc/：该目录包含进程间通信（IPC）机制的实现，如消息队列、信号量和共享内存。
- kernel/：该目录包含基本内核中与体系结构无关的部分。
- lib/：该目录包含库函数和一些辅助函数，分别是通用内核对象（kobject）处理程序和循环冗余码（CRC）计算函数等。
- mm/：该目录包含内存管理相关代码。
- net/：该目录包含网络（无论什么类型的网络）协议相关代码。
- scripts/：该目录包含在内核开发过程中使用的脚本和工具，还有其他有用的工具。

- security/：该目录包含安全框架相关代码。
- sound/：该目录包含音频子系统代码。
- usr/：该目录目前包含了 initramfs 的实现。

内核必须保持它的可移植性。任何体系结构特定的代码都应该位于 arch 目录中。当然，与用户空间 API 相关的内核代码不会改变（系统调用、/proc、/sys），因为它会破坏现有的程序。

1.1.2 内核配置

Linux 内核是一个基于 makefile 的工程，有 1000 多个选项和驱动程序。配置内核可以使用基于 ncurse 的接口命令 `make menuconfig`，也可以使用基于 X 的接口命令 make xconfig。一旦选择，所有选项会被存储到源代码根目录下的 `.config` 文件中。

大多情况下不需要从头开始配置。每个 arch 目录下面都有默认的配置文件可用，可以把它们用作配置起点：

列出配置文件

```
1 $ ls arch/<you_arch>/configs/
```

对于基于 ARM 的 CPU，这些配置文件位于 `arch/arm/configs/`；

对于基于 V3S 处理器的 Atguigu 派，默认的配置文件位于 `arch/arm/configs/atguigupi_defconfig`；

类似地，对于 x86 处理器，可以在 `arch/x86/configs/` 找到配置文件，仅有两个默认配置文件：

- `i386_defconfig`：32 位版本
- `x86_64_defconfig`：64 位版本

对 x86 系统，内核配置非常简单：

内核配置

```
1 $ make x86_64_defconfig
2 $ make zImage -j16
3 $ make modules
4 $ makeINSTALL_MOD_PATH </where/to/install>
5 $ modules_install
```

对于基于 V3S 的开发板 AtguiguPI：

可以先执行 `ARCH=arm make atguigupi_defconfig`，然后执行 `ARCH=arm make menuconfig`。前一个命令把默认的内核选项存储到 `.config` 文件中；后一个命令则根据需求来更新、增加或者删除选项。

1.1.3 构建自己的内核

构建自己的内核需要指定相关的体系结构和编译器。

交叉编译

```
1 $ ARCH=arm make atguigupi_defconfig
2 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make zImage -j16
```

内核构建完成后，会在 `arch/arm/boot/` 下生成一个单独的二进制映像文件。使用下列命令构建模块：

构建模块

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make modules
```

可以通过下列命令安装编译好的模块：

安装模块

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make modules_install
```

`modules_install` 目标需要指定一个环境变量 `INSTALL_MOD_PATH`，指出模块安装的目录。如果没有设置，则所有的模块将会被安装到 `/lib/modules/$(KERNELRELEASE)/kernel/` 目录下，具体细节将会在第 2 章讨论。

V3S 处理器支持设备树，设备树是一些文件，可以用来描述硬件（相关细节会在第 6 章介绍）。无论如何，运行下列命令可以编译所有 ARCH 设备树：

编译所有设备树

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make dtbs
```

然而，dtbs 选项不一定适用于所有支持设备树的平台。要构建一个单独的 DTB，应该执行下列命令：

单独编译 *DTB* 文件

```
1 $ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make atguigupi.dtb
```

1.2 内核代码编写风格

深入学习本节之前应该先参考一下内核编码风格手册，它位于内核源代码树的 Documentation/CodingStyle 目录下。编码风格是应该遵循的一套规则，如果想要内核开发人员接受其补丁就应该遵守这一规则。其中一些

规则涉及缩进、程序流程、命名约定等。

常见的规则如下。

- 始终使用 8 个字符的制表符缩进，每一行不能超过 80 个字符。如果缩进妨碍函数书写，那只能说明嵌套层次太多了。
- 每一个不被导出的函数或变量都必须声明为静态的（static）。
- 在带括号表达式的内部两端不要添加空格。
  `s = sizeof(struct file);` 是可以接受的，
  而 `s = sizeof( struct file );` 是不被接受的。
- 禁止使用 `typedef` 。
- 请使用 `/* this */` 注释风格，不要使用 `// this` 。
- 宏定义应该大写，但函数宏可以小写。
- 不要试图用注释去解释一段难以阅读的代码。应该重写代码，而不是添加注释。

1.3 内核结构分配和初始化

内核总是为其数据结构和函数提供两种可能的分配机制。

下面是其中的一些数据结构。

- 工作队列。
- 列表。
- 等待队列。
- Tasklet。
- 定时器。
- 完成量。
- 互斥锁。
- 自旋锁。

动态初始化器是通过宏定义实现的，因此全用大写：

- `INIT_LIST_HEAD()`
- `DECLARE_WAIT_QUEUE_HEAD()`
- `DECLARE_TASKLET()`
- 等等

表示框架设备的数据结构总是动态分配的，每个都有其自己的分配和释放 API。框架设备类型如下。

- 网络设备。
- 输入设备。
- 字符设备。
- IIO 设备。
- 类设备。
- 帧缓冲。
- 调节器。
- PWM 设备。
- RTC。

静态对象在整个驱动程序范围内都是可见的，并且通过该驱动程序管理的每个设备也是可见的。而动态分配对象则只对实际使用该模块特定实例的设备可见。

1.4 类、对象、面向对象的编程

内核通过类和设备实现面向对象的编程。内核子系统被抽象成类，有多少子系统，`/sys/class/`下几乎就有多少个目录。`struct kobject`结构是整个实现的核心，它包含一个引用计数器，以便于内核统计有多少用户使用了这个对象。每个对象都有一个父对象，在`sysfs`（加载之后）中会有一项。

属于给定子系统的每个设备都有一个指向`operations(ops)`结构的指针，该结构提供一组可以在此设备上执行的操作。

第二章 设备驱动程序基础

驱动程序是专用于控制和管理特定硬件设备的软件，因此也被称作设备驱动程序。从操作系统的角度来看，它可以位于内核空间（以特权模式运行），也可以位于用户空间（具有较低的权限）。本教程仅涉及内核空间驱动程序，特别是 Linux 内核驱动程序。我们给出的定义是，设备驱动程序把硬件功能提供给用户程序。

本章涉及以下主题。

- 模块构建过程及其加载和卸载。
- 驱动程序框架以及调试消息管理。
- 驱动程序中的错误处理。

2.1 内核空间和用户空间

内核空间和用户空间的概念有点抽象，主要涉及内存和访问权限，如图2.1所示。可以这样认为：内核是有特权的，而用户应用程序则是受限制的。这是现代 CPU 的一项功能，它可以运行在特权模式或非特权模式。学习第 11 章之后，这个概念会更加清晰。

图2.1说明内核空间和用户空间的分离，并强调了系统调用代表它们之间的桥梁（将在本章后面讨论）。每个空间的描述如下。

- 内核空间：内核驻留和运行的地址空间。内核内存（或内核空间）是由内核拥有的内存范围，受访问标志保护，防止任何用户应用程序有意或无意间与内核搞混。另一方面，内核可以访问整个系统内存，因为它在系统上以更高的优先级运行。在内核模式下，CPU 可以访问整个内存（内核空间和用户空间）。
- 用户空间：正常程序（如 vim 等）被限制运行的地址（位置）空间。可以将其视为沙盒或监狱，以便用户程序不能混用其他程序拥有的内存或任何其他资源。在用户模式下，CPU 只能访问标有用户空间访问权限的内存。用户应用程序运行到内核空间的唯一方法是通过系统调用，其中一些调用是 read、write、open、close 和 mmap 等。用户空间代码以较低的优先级运行。当进程执行系统调用时，软件中断被发送到内核，这将打开特权模式，以便该进程可以在内核空间中运行。系统调用返回时，内核关闭特权模式，进程再次受限。

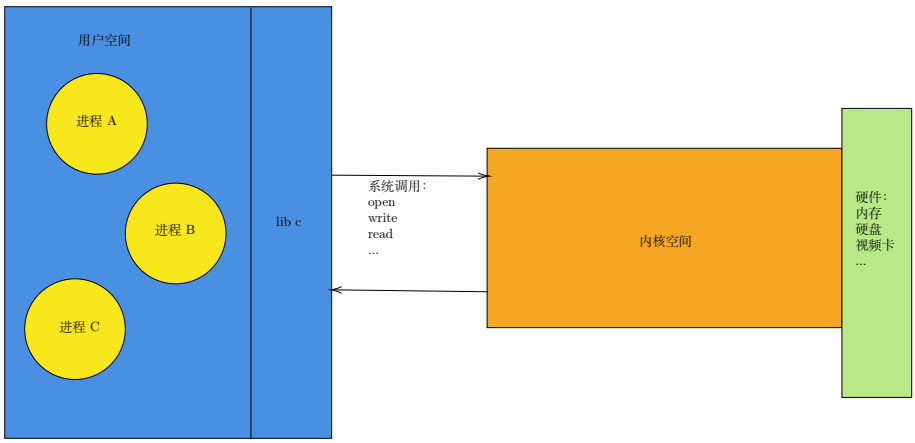

图 2.1: 内核空间和用户空间示意图

2.1.1 模块的概念

模块之于 Linux 内核就像插件（组件）之于用户软件（如 Firefox），模块动态扩展了内核功能，甚至不需要重新启动计算机就可以使用。大多数情况下，内核模块是即插即用的。一旦插入，就可以使用了。为了支持模块，构建内核时必须启用下面的选项：

支持模块

```
1 CONFIG_MODULES=y
```

2.1.2 模块依赖

Linux 内核中的模块可以提供函数或变量，用 EXPORT_SYMBOL 宏导出它们即可供其他模块使用，这些被称作符号。模块 B 对模块 A 的依赖是指模块 B 使用从模块 A 导出的符号。

在内核构建过程中运行 depmod 工具可以生成模块依赖文件。

它读取 `/lib/modules/<kernel_release>/` 中的每个模块来确定它应该导出哪些符号以及它需要什么符号。

该处理的结果写入文件 modules.dep 及其二进制版本 modules.dep.bin。它是一种模块索引。

2.1.3 模块的加载和卸载

模块要运行，应该先把它加载到内核，可以用 insmod 或 modprobe 来实现，前者需要指定模块路径作为参数，这是开发期间的首选；后者更智能化，是生产系统中的首选。

1. 手动加载。

   手动加载需要用户的干预，该用户应该拥有 root 访问权限。实现这一点的两种经典方法如下。

   在开发过程中，通常使用 insmod 来加载模块，并且应该给出所加载模块的路径：

   insmod 加载模块

   ```
   1       $ insmod /path/to/mydrv.ko
   ```

   这种模块加载形式低级，但它是其他模块加载方法的基础，也是本教程中将要使用的方法。相反，系统管理员或在生产系统中则常用 modprobe。modprobe 更智能，它在加载指定的模块之前解析文件 modules.dep，以便首先加载依赖关系。它会自动处理模块依赖关系，就像包管理器所做的那样：

   modprobe 加载模块

   ```
   1       $ modprobe mydrv
   ```

   能否使用 modprobe 取决于 depmod 是否知道模块的安装。

   能否使用 *modprobe*

   ```
   1       $ /etc/modules-load.d/<filename>.conf
   ```

如果要在启动的时候加载一些模块，则只需创建文件 `/etc/modules-load.d/<filename>.conf`，并添加应该加载的模块名称（每行一个）。
`<filename>` 应该是有意义的名称，人们通常使用模块：`/etc/modules-load.d/modules.conf`。
当然也可以根据需要创建多个 `.conf` 文件。
下面是一个 `/etc/modules-load.d/mymodules.conf` 文件中的内容：

配置文件示例

```
#this line is a comment
uio
iwlwifi
```

2. 自动加载
depmod 实用程序的作用不只是构建 modules.dep 和 modules.dep.bin 文件。内核开发人员实际编写驱动程序时已经确切知道该驱动程序将要支持的硬件。他们把驱动程序支持的所有设备的产品和厂商 ID 提供给该驱动程序。
depmod 还处理模块文件以提取和收集该信息，并在/lib/modules/<kernel_release>/modules.alias 中生成 modules.alias 文件，该文件将设备映射到其对应的驱动程序。
下面的内容摘自 modules.alias：

modules.alias

```
alias usb:v0403pFF1Cd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pFF18d*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFFd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFEd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFDd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0403pDAFCd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio
alias usb:v0D8Cp0103d*dc*dsc*dp*ic*isc*ip*in* snd_usb_audio
alias usb:v*p*d*dc*dsc*dp*ic01isc03ip*in* snd_usb_audio
alias usb:v200Cp100Bd*dc*dsc*dp*ic*isc*ip*in* snd_usb_au
```

在这一步，需要一个用户空间热插拔代理（或设备管理器），通常是 udev（或 mdev），它将在内核中注册，以便在出现新设备时得到通知。
通知由内核发布，它将设备描述（pid、vid、类、设备类、设备子类、接口以及可标识设备的所有其他信息）发送到热插拔守护进程，守护进程再调用 modprobe，并向其传递设备描述信息。接下来，modprobe 解析 modules.alias 文件，匹配与该设备相关的驱动程序。在加载模块之前，modprobe 会在 module.dep 中查找与其有依赖关系的模块。如果发现，则在相关模块加载之前先加载所有依赖模块；否则，直接加载该模块。

3. 模块卸载
常用的模块卸载命令是 rmmod，人们更喜欢用这个来卸载 insmod 命令加载的模块。使用该命令时，应该把要卸载的模块名作为参数向其传递。
模块卸载是内核的一项功能，该功能的启用或禁用由 CONFIG_MODULE_UNLOAD 配置选项的值决

定。没有这个选项，就不能卸载任何模块。以下设置将启用模块卸载功能：

模块卸载配置

```
1        CONFIG_MODULE_UNLOAD=y
```

在运行时，如果模块卸载会导致其他不良影响，则即使有人要求卸载，内核也将阻止这样做。这是因为内核通过引用计数记录模块的使用次数，这样它就知道模块是否在用。如果内核认为删除一个模块是不安全的，就不会删除它。然而，以下设置可以改变这种行为：

强制卸载模块配置

```
1        MODULE_FORCE_UNLOAD=y
```

上面的选项应该在内核配置中设置，以强制卸载模块：

强制卸载模块

```
1        rmmod -f mymodule
```

而另一个更高级的模块卸载命令是`modeprobe -r`，它会自动卸载未使用的相关依赖模块：

自动卸载相关依赖

```
1        modeprobe -r mymodule
```

这对于开发者来说是一个非常有用的选择。用下列命令可以检查模块是否已加载：

列出模块

```
1        lsmod
```

2.2 驱动程序框架

helloworld.c

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4
5  static int __init helloworld_init(void) {
6      pr_info("Hello world!\n");
7      return 0;
8  }
9
10 static void __exit helloworld_exit(void) {
11     pr_info("End of the world\n");
12 }
13
14 module_init(helloworld_init);
15 module_exit(helloworld_exit);
16 MODULE_AUTHOR("Yuan Zuo <zuoyuantc@gmail.com>");
17 MODULE_LICENSE("GPL");
```

2.2.1 模块的入口点和出口点

内核驱动程序都有入口点和出口点：前者对应于模块加载时调用的函数（modprobe 和 insmod），后者是模块卸载时执行的函数（在执行 rmmod 或 modprobe -r 时）。

main() 函数是用 C/C++ 编写的每个用户空间程序的入口点，当这个函数返回时，程序将退出。而对于内核模块，情况就不一样了：入口点可以随意命名，它也不像用户空间程序那样在 main() 返回时退出，其出点在另一个函数中定义。开发人员要做的就是通知内核把哪些函数作为入口点或出口点来执行。实际函数 hellowolrd_init 和 hellowolrd_exit 可以被命名成任何名字。实际上，唯一必须要做的是把它们作为参数提供给 module_init() 和 module_exit() 宏，将它们标识为相应的加载和删除函数。

综上所述，module_init() 用于声明模块加载（使用 insmod 或 modprobe）时应该调用的函数。初始化函数中要完成的操作是定义模块的行为。module_exit() 用于声明模块卸载（使用 rmmod）时应该调用的函数。

> ! 在模块加载或卸载后，init 函数或 exit 函数立即运行一次。

`__init`和`__exit`属性

`__init` 和 `__exit` 实际上是在 include/linux/init.h 中定义的内核宏，如下所示：

宏定义

```
1  #define __init __section(.init.text)
```

```
2 #define __exit __section(.exit.text)
```

`__init` 关键字告诉链接器将该代码放在内核对象文件的专用部分。这部分事先为内核所知，它在模块加载和 init 函数执行后被释放。这仅适用于内置驱动程序，而不适用于可加载模块。内核在启动过程中第一次运行驱动程序的初始化函数。

由于驱动程序不能卸载，因此在下次重启之前不会再调用其 init 函数，没有必要在 init 函数内记录引用次数。对于 `__exit` 关键字也是如此，在将模块静态编译到内核或未启用模块卸载功能时，其相应的代码会被忽略，因为在这两种情况下都不会调用 exit 函数。 `__exit` 对可加载模块没有影响。

我们花一点时间进一步了解这些属性的工作方式，这涉及被称作可执行和可链接格式（ELF）的目标文件。ELF 目标文件由不同的命名部分组成，其中一些部分是必需的，它们成为 ELF 标准的基础，但也可以根据自己的需要构建任一部分，并由特殊程序使用。内核就是这样做。执行 `objdump -h module.ko` 即可打印出指定内核模块 module.ko 的不同组成部分。

打印内容只有少部分属于 ELF 标准。

- .text：包含程序代码，也称为代码。
- .data：包含初始化数据，也称为数据段。
- .rodata：用于只读数据。
- .comment：注释。
- 未初始化的数据段，也称为由符号开始的块（block started by symbol，bss）。

其他部分是根据内核的需要添加的。本章较重要的部分是.modeinfo 和.init.text，前者存储有关模块的信息，后者存储以 __init 宏为前缀的代码。

链接器（Linux 系统上的 ld）是 binutils 的一部分，负责将符号（数据、代码等）放置到生成的二进制文件中的适当部分，以便在程序执行时可以被加载器处理。二进制文件中的这些部分可以自定义、更改它们的默认位置，甚至可以通过提供链接器脚本 [称为链接器定义文件（LDF）或链接器定义脚本（LDS）] 来添加其他部分。要实现这些操作只需通过编译器指令把符号的位置告知链接器即可，GNU C 编译器为此提供了一些属性。Linux 内核提供了一个自定义 LDS 文件，它位于 arch/<arch>/kernel/vmlinux.lds.S 中。对于要放置在内核 LDS 文件所映射的专用部分中的符号，使用 __init 和 __exit 进行标记。

总之，__init 和 __exit 是 Linux 指令（实际上是宏），它们使用 C 编译器属性指定符号的位置。这些指令指示编译器将以它们为前缀的代码分别放在.init.text 和.exit.text 部分，虽然内核可以访问不同的对象部分。

2.2.2 模块信息

即使不读代码，也应该能够收集到关于给定模块的一些信息（如作者、参数描述、许可）。内核模块使用其.modinfo 部分来存储关于模块的信息，所有 MODULE_* 宏都用参数传递的值更新这部分的内容。其中一些宏是 MODULE_DESCRIPTION()、MODULE_AUTHOR() 和 MODULE_LICENSE()。内核提供的在模块信息部分添加条目的真正底层宏是 MODULE_INFO(tag,info)，它添加的一般信息形式是 tag=info。这意味着驱动程序作者可以自由添加其想要的任何形式信息，例如：

作者信息

```
1 MODULE_INFO(my_field_name, "What eeasy value");
```

在给定模块上执行 `objdump -d -j .modinfo` 命令可以转储内核模块.modeinfo 部分的内容，如图 2-3 所示。

modinfo 部分可以看作模块的数据表。实际格式化打印信息的用户空间工具是 modinfo，如图 2-4 所示。

除自定义信息之外，还应该提供标准信息，内核为这些信息提供了宏，包括许可、模块作者、参数描述、模块版本和模块描述。

1. 许可

模块的许可由 MODULE_LICENSE() 宏定义：

许可

```
1 MODULE_LICENSE ("GPL");
```

应该如何与其他开发人员共享 (或不共享) 许可定义源代码。MODULE_LICENSE() 告诉内核模块采用何种许可。它对模块行为有影响，因为与 GPL 不兼容的许可将导致模块不能通过 EXPORT_SYMBOL_GPL() 宏看到/使用内核导出的服务/函数，这个宏只对 GPL 兼容模块显示符号，这与 EXPORT_SYMBOL() 相反，后者为具有任何许可的模块导出函数。加载 GPL 不兼容模块也会导致内核被污染，这意味着已经加载非开源或不可信代码，可能没有社区支持。请记住，没有 MODULE_LICENSE() 的模块被认为是非开源的，也会污染内核。以下摘自 include/linux/module.h，描述了内核支持的许可：

内核支持的许可

```
1  /*
2  * The following license indents are currently accepted as indicating free
3  * software modules
4  *
5  * "GPL"                      [GNU Public License v2 or later]
6  * "GPL v2"                   [GNU Public License v2]
7  * "GPL and additional rights" [GNU Public License v2 rights and more]
8  * "Dual BSD/GPL"             [GNU Public License v2 or BSD license choice]
9  * "Dual MIT/GPL"             [GNU Public License v2 or MIT license choice]
10 * "Dual MPL/GPL"             [GNU Public License v2 or Mozilla license choice
       ]
11 *
12 * The following other idents are available
13 *
14 * "Proprietary"              [Non free products]
15 *
16 * There are dual licensed components, but when running with Linux it is the
17 * GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL
18 * is a GPL combined work.
19 *
20 * This exists for several reasons
```

```
21 * 1. So modinfo can show license info for users wanting to vet their setup is
       free
22 * 2. So the community can ignore bug reports including proprietary modules
23 * 3. So vendors can do likewise based on their own policies
24 */
```

模块至少必须与 GPL 兼容，才能享受完整的内核服务。

2. 模块作者

MODULE_AUTHOR() 声明模块的作者：

模块作者

```
1 MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

作者可能有多个，在这种情况下，每个作者必须用 MODULE_AUTHOR() 声明：

多个模块作者

```
1 MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
2 MODULE_AUTHOR("Lorem Ipsum <l.ipsum@foobar.com>");
```

3. 模块描述

MODULE_DESCRIPTION() 简要描述模块的功能：

模块描述

```
1 MODULE_DESCRIPTION("Hello, world! Module");
```

2.3 错误和消息打印

错误代码由内核或用户空间应用程序 (通过 errno 变量) 解释。错误处理在软件开发中非常重要，而不仅仅是在内核开发中。幸运的是，内核提供的几种错误，几乎涵盖了可能会遇到的所有错误，有时需要把它们打印出来以帮助进行调试。

2.3.1 错误处理

为给定的错误返回错误代码会导致内核或用户空间应用产生不必要的行为, 从而做出错误的决定。为了保持清楚, 内核树中预定义的错误几乎涵盖了我们可能遇到的所有情况。

一些错误 (及其含义) 在 include/uapi/asm-generic/errno-base.h 中定义, 列表的其余错误可以在 include/uapi/asm-generic/errno.h 中找到。

以下是从 include/uapi/asm-generic/errno-base.h 中摘录的错误列表:

错误编码

```
1  #define EPERM     1   /* 操作不允许 */
2  #define ENOENT    2   /* 没有这样的文件或者目录 */
3  #define ESRCH     3   /* 没有这样的进程 */
4  #define EINTR     4   /* 中断系统调用 */
5  #define EIO       5   /* I/O错误 */
6  #define ENXIO     6   /* 没有这样的设备或地址 */
7  #define E2BIG     7   /* 参数列表太长 */
8  #define ENOEXEC   8   /* Exec格式错误 */
9  #define EBADF     9   /* 错误的文件数量*/
10 #define ECHILD    10  /* 没有子进程 */
11 #define EAGAIN    11  /* 再试一次 */
12 #define ENOMEM    12  /* 内存不足 */
13 #define EACCES    13  /* 没有权限 */
14 #define EFAULT    14  /* 错误的地址 */
15 #define ENOTBLK   15  /* 块设备要求*/
16 #define EBUSY     16  /* 设备或资源忙 */
17 #define EEXIST    17  /* 文件已存在 */
18 #define EXDEV     18  /* 跨设备的链接 */
19 #define ENODEV    19  /* 没有这样的设备 */
20 #define ENOTDIR   20  /* 不是目录 */
21 #define EISDIR    21  /* 是目录 */
22 #define EINVAL    22  /* 无效参数 */
23 #define ENFILE    23  /* 文件表溢出*/
24 #define EMFILE    24  /* 打开的文件太多 */
25 #define ENOTTY    25  /* 不是打字机 */
26 #define ETXTBSY   26  /* 文本文件忙 */
27 #define EFBIG     27  /* 文件太大 */
28 #define ENOSPC    28  /* 设备上没有空间了 */
29 #define ESPIPE    29  /* 非法寻求 */
30 #define EROFS     30  /* 只读文件系统 */
31 #define EMLINK    31  /* 链接太多 */
32 #define EPIPE     32  /* 破坏的管道 */
33 #define EDOM      33  /* 函数域外的数学参数 */
34 #define ERANGE    34  /* 数学结果无法表示 */
```

大多情况下, 经典的返回错误方式是这种形式:return -ERROR, 特别是在响应系统调用时。例如, 对于 I/O 错误, 错误代码是 EIO, 应该执行的语句是 return -EIO:

错误返回

```
dev = init(&ptr);
if (!dev)
    return -EIO;
```

错误有时会跨越内核空间, 传播到用户空间。如果返回的错误是对系统调用 (open、read、ioctl、mmap) 的响应, 则该值将自动赋给用户空间 errno 全局变量, 在该变量上调用 strerror(errno) 可以将错误转换为可读字符串:

code

```
#include <errno.h> /* to access errno global variable */
#include <string.h>
[...]
if(write(fd, buf, 1) < 0) {
    printf("something gone wrong! %s\n", strerror(errno));
}
[...]
```

当遇到错误时, 必须撤销在这个错误发生之前的所有设置。通常的做法是使用 goto 语句:

使用 *goto* 进行错误处理

```
ptr = kmalloc(sizeof (device_t));
if(!ptr) {
    ret = -ENOMEM;
    goto err_alloc;
}

dev = init(&ptr);

if(dev) {
    ret = -EIO
    goto err_init;
}

return 0;
```

```
15
16 err_init:
17     free(ptr);
18 err_alloc:
19     return ret;
```

使用 goto 语句的原因很简单。当处理错误时, 假设错误出现在第 5 步, 则必须清除以前的操作 (步骤 1 步骤 4)。而不是像下面这样执行大量的嵌套检查操作:

嵌套处理

```
1 if (ops1() != ERR) {
2     if (ops2() != ERR) {
3         if ( ops3() != ERR) {
4             if (ops4() != ERR) {
```

这可能会令人困惑, 并可能导致缩进问题。像下面这样用 goto 语句会使控制流程显得更直观, 这种方法更受欢迎:

goto 处理错误

```
1  if (ops1() == ERR)
2      goto error1;
3  if (ops2() == ERR)
4      goto error2;
5  if (ops3() == ERR)
6      goto error3;
7  if (ops4() == ERR)
8      goto error4;
9  error5:
10 [...]
11 error4:
12 [...]
13 error3:
14 [...]
15 error2:
16 [...]
17 error1:
18 [...]
```

这就是说应该只在函数中使用 goto 跳转。

2.3.2 处理空指针错误

当返回指针的函数返回错误时, 通常返回的是 NULL 指针。而去检查为什么会返回空指针是没有任何意义的, 因为无法准确了解为什么会返回空指针。为此, 内核提供了 3 个函数 ERR_PTR、IS_ERR 和 PTR_ERR:

处理空指针错误

```
1 void *ERR_PTR(long error);
2 long IS_ERR(const void *ptr);
3 long PTR_ERR(const void *ptr);
```

第一个函数实际上把错误值作为指针返回。假若函数在内存申请失败后要执行语句 return -ENOMEM, 则必须改为这样的语句:return ERR_PTR (-ENOMEM);。

第二个函数用于检查返回值是否是指针错误: `if (IS_ERR(foo))` 。

最后一个函数返回实际错误代码: `return PTR_ERR(foo);` 。

以下是一个例子, 说明如何使用 `ERR_PTR` 、 `IS_ERR` 和 `PTR_ERR` :

example

```
1  static struct iio_dev *indiodev_setup(){
2      [...]
3      struct iio_dev *indio_dev;
4      indio_dev = devm_iio_device_alloc(&data->client->dev, sizeof(data));
5      if (!indio_dev)
6          return ERR_PTR(-ENOMEM);
7      [...]
8      return indio_dev;
9  }
10
11 static int foo_probe([...]){
12     [...]
13     struct iio_dev *my_indio_dev = indiodev_setup();
14     if (IS_ERR(my_indio_dev))
15         return PTR_ERR(data->acc_indio_dev);
16     [...]
17 }
```

关于错误处理补充一点，摘录自内核编码风格部分：如果函数名称是动作或命令式命令，则函数返回的错误代码应该是整数；如果函数名称是一个谓词，则该函数应返回布尔值 succeeded(成功的)。例如,add work 是一个命令,add_work() 函数返回 0 表示成功，返回-EBUSY 表示失败。同样,PCI device present 是谓词,pci_dev_present() 函数如果成功找到匹配设备，则返回 1；否则返回 0。

2.3.3 消息打印——printk()

printk() 是在内核空间使用的，其作用和在用户空间使用 printf() 一样，执行 dmesg 命令可以显示 printk() 写入的行。根据所打印消息的重要性不同，可以选用 include/linux/kern_levels.h 中定义的八个级别的日志消息，下面将介绍它们的含义。

下面列出的是内核日志级别，每个级别对应一个字符串格式的数字，其优先级与该数字的值成反比。例如,0 具有较高的优先级：

日志级别

```
1  #define KERN_SOH          "\001"       /* ASCII头开始 */
2  #define KERN_SOH_ASCII    '\001'
3  #define KERN_EMERG        KERN_SOH "0" /* 系统不可用 */
4  #define KERN_ALERT        KERN_SOH "1" /* 必须立即采取行动 */
5  #define KERN_CRIT         KERN_SOH "2" /* 重要条件 */
6  #define KERN_ERR          KERN_SOH "3" /* 错误条件 */
7  #define KERN_WARNING      KERN_SOH "4" /* 警报条件 */
8  #define KERN_NOTICE       KERN_SOH "5" /* 正常但重要的情况 */
9  #define KERN_INFO         KERN_SOH "6" /* 信息 */
10 #define KERN_DEBUG        KERN_SOH "7" /* 调试级别的信息 */
```

以下代码显示如何打印内核消息和日志级别：

example

```
1    printk(KERN_ERR "This is an error\n");
```

如果省略调试级别 (`printk("This is an error\n")`)，则内核将根据`CONFIG_DEFAULT_MESSAGE_LOGLEVEL`配置选项 (这是默认的内核日志级别) 向该函数提供一个调试级别。实际上可以使用以下宏，其名称更有意义，它们是对前面所定义内容的包装——pr_emerg、pr_alert、pr_crit、pr_err、pr_warning、pr_notice、pr_info 和 pr_debug：

example

```
1    pr_err("This is the same error\n");
```

对于新开发的驱动程序，建议使用这些包装。printk() 的实现是这样的：调用它时，内核会将消息日志级别与

当前控制台的日志级别进行比较; 如果前者比后者更高 (值更低), 则消息会立即打印到控制台。可以这样检查日志级别参数:

检查日志级别

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

上面的输出中, 第一个值是当前日志级别 (4), 第二个值是按照 CONFIG_DEFAULT_MESSAGE_LOGLEVEL 选项设置的默认值。其他值与本章内容无关, 可以忽略。

内核日志级别列表如下:

example

```
/* integer equivalents of KERN_<LEVEL> */
#define LOGLEVEL_SCHED -2 /* 计划代码中的延迟消息设置为此特殊级别 */
#define LOGLEVEL_DEFAULT -1 /* 默认(或最新)日志级别 */
#define LOGLEVEL_EMERG 0 /* 系统不可用 */
#define LOGLEVEL_ALERT 1 /* 必须立即采取行动 */
#define LOGLEVEL_CRIT 2 /* 重要条件 */
#define LOGLEVEL_ERR 3 /* 错误条件 */
#define LOGLEVEL_WARNING 4 /* 警报条件 */
#define LOGLEVEL_NOTICE 5 /* 正常但重要的情况 */
#define LOGLEVEL_INFO 6 /* 信息 */
#define LOGLEVEL_DEBUG 7 /* 调试级别信息 */
```

当前日志级别可以这样更改:

example

```
# echo <level> > /proc/sys/kernel/printk
```

! printk() 永远不会阻塞, 即使在原子上下文中调用也足够安全。它会尝试锁定控制台打印消息。如果锁定失败, 输出则被写入缓冲区, 函数返回, 永不阻塞。然后通知当前控制台占有者有新的消息, 在它释放控制台之前打印它们。

内核也支持其他调试方法: 动态调试或者在文件的顶部使用 `#define DEBUG` 。对这种调试方式感兴趣的人可以参考以下内核文档:Documentation/dynamic-debug-howto.txt。

第三章 一个完整的最简单的驱动 Demo

本章描述了基于全志 V3S 开发板的简单驱动程序和测试应用程序的设计流程。我们设计的驱动程序和测试程序极其简单，适合初学者上手学习。

软件运行的硬件环境是基于 V3S 开发板中的全志 V3S 处理器，该处理器集成了一个 1.2GHz 工作主频的单 ARM CortexTM-A7 核，芯片内部集成了 64MB DRAM 存储器。

设计流程概述

1. 步骤一：编写一个 `demo_driver.c` 的驱动程序。
2. 步骤二：编写 makefile 文件。
3. 步骤三：编写一个 `demo_app.c` 的应用程序。
4. 步骤四：在 V3S 开发板中安装 `demo_driver` 驱动程序，并测试 `demo_app` 应用程序。

linux 系统是一个分层结构，我们设计的 `demo_driver` 位于内核中的驱动部分， `demo_app` 位于用户级。

3.1 编写驱动程序

编写一个 `demo_driver.c` 的驱动程序，驱动程序源码如下：

demo_driver.c

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/fs.h>
4  #include <linux/init.h>
5
6  #define  DRIVER_MAJOR  188
7  #define  DEVICE_NAME  "demo_driver"
8
9  static int demo_open(struct inode *inode, struct file *file)
10 {
11    printk(KERN_EMERG "open\n");
12    return 0;
13 }
14
15 static ssize_t demo_write(struct file *file, const char __user * buf, size_t
       count, loff_t *ppos)
16 {
17    printk(KERN_EMERG "write\n");
18    return 0;
19 }
20 static ssize_t demo_read(struct file *file, char __user * buf, size_t count,
       loff_t *ppos)
21 {
```

```
22      printk(KERN_EMERG "read\n");
23      return 0;
24  }
25
26  static int demo_close(struct inode *inode, struct file *file)
27  {
28      printk(KERN_EMERG "close\n");
29      return 0;
30  }
31
32  static struct file_operations demo_flops =
33  {
34      .owner =  THIS_MODULE,
35      .open  =  demo_open,
36      .write =  demo_write,
37      .read   =     demo_read,
38      .release = demo_close,
39  };
40
41  static int __init demo_init(void)
42  {
43      int ret;
44
45      //注册设备
46      ret = register_chrdev(DRIVER_MAJOR,DEVICE_NAME, &demo_flops);
47
48      if (ret < 0)
49      {
50        printk(KERN_EMERG DEVICE_NAME " can't register major number.\n");
51        return ret;
52      }
53      else
54      {
55         printk(KERN_EMERG DEVICE_NAME "demo init\n");
56      }
57
58      return 0;
59  }
60
61  static void __exit demo_exit(void)
62  {
```

```
63      unregister_chrdev(DRIVER_MAJOR, DEVICE_NAME);
64      printk(KERN_EMERG DEVICE_NAME "demo exit\n");
65  }
66
67  module_init(demo_init);
68  module_exit(demo_exit);
69  MODULE_LICENSE("GPL");
```

编译驱动程序依赖 linux 内核源码环境，需要把驱动 C 文件放在 linux 源码目录中，通过编译 linux 内核的方式得到驱动 ko 文件。

为了得到驱动 ko，我们在 linux/drivers/char 目录下面创建一个 demo_driver 的文件夹。

然后把 demo_driver.c 程序拷贝到 demo_driver 的文件夹中。

3.2 编写 Makefile

我们通过编译 linux 内核的方式得到驱动 ko 文件，为了在编译内核时得到相应的驱动，我们需要在 linux 内核代码中增加和修改 Makefile 文件（Makefile 的作用指定了工程编译的方法和步骤）。

首先我们修改 linux/drivers/char 目录下 Makefile 文件，在 Makefile 文件内容最末行加入一行代码：

添加一行代码

```
1  obj += demo_driver/
```

这句代码的意思是：编译内核时，子目录 demo_driver/下的文件也将进行编译。

我们在 linux/drivers/char/demo_driver 中创建一个 Makefile 文件，同时在 Makefile 文件中加入一行代码：

Makefile

```
1  obj-m += demo_driver.o
```

这句代码的意思是：编译当前目录中的 demo_driver.c 文件，输出一个 demo_driver.ko 的驱动文件。

准备好 demo_driver.c 文件和 Makefile 文件后，我们执行编译内核指令。

经过上述一波操作，我们得到了 demo_driver.ko 的驱动文件。

3.3 编写应用程序

编写一个 demo_app.c 的应用程序，程序源码如下：

demo_app.c

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

int main(int arvc, char *argv[])
{
    int fd;
    int value = 0;
    printf("demo test\n");

    /// 打开驱动
    fd = open("/dev/demo_driver", O_RDWR);

    while(1)
    {
        /// 执行驱动读操作
        read(fd, &value, 4);
        sleep(1);
        /// 执行驱动写操作
        write(fd, &value, 4);
        sleep(1);

        printf("demo run\n");
    }
    return 0;
}
```

然后编译应用程序。

shell

```shell
arm-linux-gnueabihf-gcc -static -o demo_app demo_app.c
```

于是我们得到一个 demo_app 可执行文件。
然后将得到的两个二进制 `demo_driver.ko` 和 `demo_app` 拷贝到我们的嵌入式系统里面。
安装驱动：

安装驱动

```
$ insmod demo_driver.ko
# 执行创建文件节点指令
$ mknod /dev/demo_driver c 188 0
# 查看驱动设备
$ ls /dev
```

188 的是设备号，我们在设计的驱动 C 文件中有如下定义：

设备号定义

```
#define  DRIVER_MAJOR  188
#define  DEVICE_NAME   "demo_driver"
```

执行 `demo_app` 程序：

执行程序

```
$ chmod 777 demo_app
$ ./demo_app
```

如果重启设备，则需要重新加载驱动。

第四章 内核工具和辅助函数

内核是独立的软件, 正如将在本章中看到的那样, 它没有使用任何 C 语言库。它实现了现代库中可能具有的所有机制 (甚至更多), 如压缩、字符串函数等。

本章涉及以下主题。

- 介绍内核容器数据结构。
- 探讨内核睡眠机制。
- 使用定时器。
- 深入研究内核锁定机制 (互斥锁、自旋锁)。
- 使用内核专用 API 实现延迟工作。
- 使用 IRQ。

4.1 理解宏 container_of

在代码中管理多个数据结构时, 几乎总是需要将一个结构嵌入另一个结构中, 并随时检索它们, 而不关心有关内存偏移或边界的问题。假设有一个 struct person, 其定义如下:

struct person

```
1 struct person {
2     int age;
3     char *name;
4 } p;
```

只用 age 或 name 上的指针就可以检索包装 (包含) 该指针的整个结构。顾名思义,container_of 宏用于查找指定结构字段的容器。该宏在 include/linux/kernel.h 中定义, 如下所示:

container_of

```
1 #define container_of(ptr, type, member) ({        \
2     const typeof(((type *)0)->member) * __mptr = (ptr); \
3     (type *)((char *)__mptr - offsetof(type, member)); })
```

不要害怕指针, 就当它是:

函数表达

```
1     container_of(pointer, container_type, container_field);
```

前面代码片段中包含的元素如下。

- pointer: 指向结构字段的指针。
- container_type: 包装 (包含) 指针的结构类型。
- container_field:pointer 指向的结构内字段的名称。

来看下面的容器:

容器

```
1 struct person {
2     int age;
3     char *name;
4 };
```

现在考虑其一个实例, 以及一个指向其 name 成员的指针:

example

```
1 struct person somebody;
2 [...]
3 char *the_name_ptr = somebody.name;
```

除指向 name 成员的指针 (the_name_ptr) 外, 还可以使用 container_of 宏来获取指向包装此成员的整个结构 (容器) 的指针, 方法如下:

example

```
1 struct person *the_person;
2 the_person = container_of(the_name_ptr, struct person, name);
```

container_of 考虑 name 从该结构开始处的偏移量, 进而获得正确的指针位置。从指针 the_name_ptr 中减去字段 name 的偏移量, 即可得到正确的位置。这就是该宏最后一行代码的功能:

example

```
1 (type *)( (char *)__mptr - offsetof(type,member) );
```

下面的例子实际应用该宏:

example

```
struct family {
    struct person *father;
    struct person *mother;
    int number_of_suns;
    int salary;
} f;

/*
* 指向结构字段的指针
* (可以是任何家庭的任何成员吗)
*/
struct *person = family.father;
struct family *fam_ptr;

/* 找回他的家族 */
fam_ptr = container_of(person, struct family, father);
```

关于 container_of 宏, 只需了解这些就够了。本书将进一步开发的真正驱动程序像下面这样:

example

```
struct mcp23016 {
    struct i2c_client *client;
    struct gpio_chip chip;
}

/* 检索给定指针chip字段的mcp23016结构体 */
static inline struct mcp23016 *to_mcp23016(struct gpio_chip *gc)
{
    return container_of(gc, struct mcp23016, chip);
}

static int mcp23016_probe(struct i2c_client *client, const struct i2c_device_id
     *id)
{
    struct mcp23016 *mcp;
    [...]
    mcp = devm_kzalloc(&client->dev, sizeof(*mcp), GFP_KERNEL);
    if (!mcp)
        return -ENOMEM;
```

```
19      [...]
20  }
```

宏 container_of 主要用在内核的通用容器中。在本书的一些例子 (从第 5 章开始) 中, 会用到 container_of 宏。

4.2 链表

想象一下, 有一个驱动程序管理多个设备, 假设有 5 个设备, 可能需要在驱动程序中跟踪每个设备, 这就需要链表。链表实际上有两种类型。

- 单链表
- 双链表

内核开发者只实现了循环双链表, 因为这个结构能够实现 FIFO 和 LIFO, 并且内核开发者要保持最少代码。

为了支持链表, 代码中要添加的头文件是 <linux/list.h>。内核中链表实现核心部分的数据结构是 struct list_head, 其定义如下:

链表定义

```
1  struct list_head {
2      struct list_head *next, *prev;
3  };
```

struct list_head 用在链表头和每个节点中。在内核中, 将数据结构表示为链表之前, 该结构必须嵌入 struct list_head 字段。例如, 我们来创建汽车链表:

汽车链表

```
1  struct car {
2      int door_number;
3      char *color;
4      char *model;
5  };
```

在创建汽车链表之前, 必须修改其结构, 嵌入 struct list_head 字段。结构变成如下形式:

example

```
1 struct car {
2     int door_number;
3     char *color;
4     char *model;
5     struct list_head list; /* 内核的链表结构 */
6 };
```

创建 struct list_head 变量, 该变量总是指向链表的头部 (第一个元素)。list_head 的这个实例与任何汽车都无关, 而是一个特殊实例:

特殊实例

```
1     static LIST_HEAD(carlist);
```

现在可以创建汽车并将其添加到链表 carlist:

example

```
1  #include <linux/list.h>
2  struct car *redcar = kmalloc(sizeof(*car), GFP_KERNEL);
3  struct car *bluecar = kmalloc(sizeof(*car), GFP_KERNEL);
4
5  /* 初始化每个节点的列表条目*/
6  INIT_LIST_HEAD(&bluecar->list);
7  INIT_LIST_HEAD(&redcar->list);
8
9  /* 为颜色和模型字段分配内存, 并填充每个字段 */
10 [...]
11 list_add(&redcar->list, &carlist) ;
12 list_add(&bluecar->list, &carlist) ;
```

现在,carlist 包含两个元素。接下来深入介绍链表 API。

4.2.1 创建和初始化链表

有两种方法创建和初始化链表。

1. 动态方法

动态方法由 struct list_head 组成, 用 INIT_LIST_HEAD 宏初始化:

动态初始化链表

```
1 struct list_head mylist;
2 INIT_LIST_HEAD(&mylist);
```

以下是 INIT_LIST_HEAD 的展开形式:

example

```
1 static inline void INIT_LIST_HEAD(struct list_head *list)
2 {
3     list->next = list;
4     list->prev = list;
5 }
```

2. 静态方法
静态分配通过 LIST_HEAD 宏完成:

example

```
1 LIST_HEAD(mylist)
```

LIST_HEAD 的定义如下:

example

```
1 #define LIST_HEAD(name) \
2     struct list_head name = LIST_HEAD_INIT(name)
```

其展开如下:

example

```
1 #define LIST_HEAD_INIT(name) { &(name), &(name) }
```

这为 name 字段内的每个指针 (prev 和 next) 赋值, 使其指向 name 自身 (就像 INIT_LIST_HEAD 做的那样)。

4.2.2 创建链表节点

要创建新节点，只需创建数据结构实例，初始化嵌入在其中的 list_head 字段。以汽车为例，其代码如下：

example

```
struct car *blackcar = kzalloc(sizeof(struct car), GFP_KERNEL);

/* 非静态初始化，因为它是嵌入的列表字段 */
INIT_LIST_HEAD(&blackcar->list);
```

如前所述，使用的 INIT_LIST_HEAD 是动态分配的链表，它通常是另一个结构的一部分。

4.2.3 添加链表节点

内核提供的 list_add 用于向链表添加新项，它是内部函数 __list_add 的包装：

example

```
void list_add(struct list_head *new, struct list_head *head);
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

__list_add 将两个已知项作为参数，在它们之间插入元素。它在内核中的实现非常简单：

example

```
static inline void __list_add(struct list_head *new, struct list_head *prev,
    struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

下面的例子在链表中添加两辆车：

添加两辆车

```
1 list_add(&redcar->list, &carlist);
2 list_add(&blue->list, &carlist);
```

这种模式可以用来实现堆栈。将节点添加到链表的另一个函数, 代码如下:

example

```
1 void list_add_tail(struct list_head *new, struct list_head *head);
```

这将把指定新项插入链表的末尾。对于之前的例子, 可以使用以下代码:

example

```
1 list_add_tail(&redcar->list, &carlist);
2 list_add_tail(&blue->list, &carlist);
```

这种模式可以用来实现队列。

4.2.4 删除链表节点

内核代码中的链表处理是一项简单的任务。删除节点很简单:

删除节点

```
1 void list_del(struct list_head *entry);
```

删除红色车

删除红色车

```
1 list_del(&redcar->list);
```

list_del 断开指定节点的 prev 和 next 指针, 移除该节点。分配给该节点的内存需要使用 kfree 手动释放。

4.2.5 链表遍历

使用宏 `list_for_each_entry(pos, head, member)` 进行链表遍历。

- head: 链表的头节点。
- member: 数据结构 (在我们的例子中, 它是 list) 中链表 struct list_head 的名称。
- pos: 用于迭代。它是一个循环游标, 就像 for(i=0; i<foo; i++) 中的 i。head 可以是链表的头节点或任一项, 这没关系, 因为所处理的是双向链表:

example

```
struct car *acar; /* loop counter */
int blue_car_num = 0;
/* 'list' is the name of the list_head struct in our data structure */
list_for_each_entry(acar, carlist, list) {
    if(acar->color == "blue")
        blue_car_num++;
}
```

为什么需要数据结构中 list_head 类型字段的名称? 请看 list_for_each_entry:

example

```
#define list_for_each_entry(pos, head, member)       \
for (pos = list_entry((head)->next, typeof(*pos), member); \
    &pos->member != (head);                          \
    pos = list_entry(pos->member.next, typeof(*pos), member))

#define list_entry(ptr, type, member)                \
    container_of(ptr, type, member)
```

鉴于此, 我们可以理解这都是 container_of 的功能。另外, 还请记住 list_for_each_entry_safe(pos, n, head, member)。

第五章 设备树的概念

设备树（DT，Derive Tree）是易于阅读的硬件描述文件，它采用 JSON 式的格式化风格，在这种简单的树形结构中，设备表示为带有属性的节点。属性可以是空（只有键，用来描述布尔值），也可以是键值对，其中的值可以包含任意的字节流。本章简单地介绍 DT，每个内核子系统或框架都有自己的 DT 绑定。讲解有关话题时将包括具体的绑定。DT 源于 OF，这是计算机公司公认的标准，其主要目的是定义计算机固件系统的接口。

本章涉及以下主题。

- 命名约定，以及别名和标签。
- 描述数据类型及其 API。
- 管理寻址方案和访问设备资源。
- 实现 OF 匹配风格，提供应用程序的相关数据。

5.1 设备树机制

将选项 `CONFIG_OF` 设置为 Y 即可在内核中启用 DT。要在驱动程序中调用 DT API，必须添加以下头文件：

头文件

```
#include <linux/of.h>
#include <linux/of_device.h>
```

DT 支持一些数据类型。下面通过对实例节点的描述来介绍这些数据类型：

数据类型

```
/* 注释 */
/* 另一个注释 */
node_label: nodename@reg{
    string-property = "a string";
    string-list = "red fish", "blue fish";
    one-int-property = <197>; /* 该属性中的一个单元格*/
    /* 每个数字（单元格）是一个 32 位的整型（uint32）
     * 属性中有 3 个单元格
     */
    int-list-property = <0xbeef 123 0xabcd4>;
    mixed-list-property = "a string", <0xadbcd45>, <35>, [0x01 0x23 0x45]
    byte-array-property = [0x01 0x23 0x45 0x67];
    boolean-property;
};
```

以下是设备树中使用的一些数据类型的定义。

- 文本字符串用双引号表示。可以使用逗号来创建字符串列表。
- 单元格是由尖括号分隔的 32 位无符号整数。
- 布尔数据不过是空属性。其取值是 true 或 false 取决于属性存在与否。

5.1.1 命名约定

每个节点都必须有`<name> [@ <address>]`形式的名称，其中`<name>`是一个字符串，其长度最多为 31 个字符，`[@ <address>]`是可选的，具体取决于节点代表是否为可寻址的设备。`<address>`是用来访问设备的主要地址。设备命名的一个例子如下：

例子

```
expander@20 {
    compatible = "microchip,mcp23017";
    reg = <20>;
    [...]
};
```

或者

例子

```
i2c@021a0000 {
    compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
    reg = <0x021a0000 0x4000>;
    [...]
};
```

另外，仅当打算从另一节点的属性引用节点时，标记节点才有用。在 6.1.2 节将看到标签是指向节点的指针。

5.1.2 别名、标签和 phandle

了解这 3 个要素的工作机制非常重要，它们经常在 DT 中使用。下面的 DT 例子解释它们是如何工作的:

三要素

```
aliases {
    ethernet0 = &fec;
    gpio0 = &gpio1;
    gpio1 = &gpio2;
    mmc0 = &usdhc1;
    [...]
```

```
};
gpio1: gpio@0209c000 {
    compatible = "fsl,imx6q-gpio", "fsl,imx35-gpio";
    [...]
};
node_label: nodename@reg {
    [...];
    gpios = <&gpio1 7 GPIO_ACTIVE_HIGH>;
};
```

标签不过是标记节点的方法, 可以用唯一的名称来标识节点。在现实世界中,DT 编译器将该名称转换为唯一的 32 位值。在前面的例子中,gpio1 和 node_label 都是标签。之后可以用标签来引用节点, 因为标签对于节点是唯一的。

指针句柄 (pointer handle, 简写为 phandle) 是与节点相关联的 32 位值, 用于唯一标识该节点, 以便可以从另一个节点的属性引用该节点。标签用于一个指向节点的指针。使用 `<&mylabel>` 可以指向标签为 mylabel 的节点。

> ! & 的用途就像在 C 编程语言中一样, 用于获取元素的地址。

在前面的例子中, `&gpio1` 被转换为 phandle, 以便引用 gpio1 节点。下面的例子也是这样:

t

```
hename@address {
    property = <&mylabel>;
};

mylabel: thename@adresss {
    [...]
}
```

为了在查找节点时不遍历整棵树, 引入了别名的概念。在 DT 中, 别名节点可以看作是快速查找表, 即另一个节点的索引。可以使用函数 `find_node_by_alias()` 来查找指定别名的节点。别名不是直接在 DT 源中使用, 而是由 Linux 内核来引用。

5.1.3 DT 编译器

DT 有两种形式: 文本形式 (代表源, 也称作 DTS) 和二进制块形式 (代表编译后的 DT), 也称作 DTB。源文件的扩展名是.dts。实际上, 还有.dtsi 文本文件, 代表 SoC 级定义, 而.dts 文件代表开发板定义。就像在源文件 (.c) 中包含头文件 (.h) 一样, 应该把.dtsi 作为头文件包含在.dts 文件中。而二进制文件则使用.dtb 扩展名。

实际上还有第三种形式, 即 DT 在 `/proc/device-tree` 中的运行时表示。

正如其名称所述, 用于编译设备树的工具称为设备树编译器 (dtc)。从根内核源代码, 可以为特定的体系结

构编译独立的特定 DT 或所有 DT。

下面为 ARM SoC 编译所有 DT(.dts) 文件:

编译设备树

```
1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make dtbs
```

编译单独的 DT:

编译单独的设备树文件

```
1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make imx6dl-sabrelite.dtb
```

在上一个例子中, 源文件的名称是 imx6dl-sabrelite.dts。

对于编译过的设备树 (.dtb) 文件, 可以做相反的操作, 从中提取源 (.dts) 文件:

反编译设备树文件

```
1 dtc -I dtb -O dtsarch/arm/boot/dts imx6dl-sabrelite.dtb > path/to/my_devicetree
      .dts
```

! 为了调试, 将 DT 公开给用户空间可能是有用的。使用 CONFIG_PROC_DEVICETREE 配置变量可以实现该目的。然后, 可以浏览/proc/device-tree 中的 DT。

5.2 表示和寻址设备

每个设备在 DT 中至少有一个节点。某些属性对于许多设备类型是通用的, 特别是位于内核已知总线 (SPI、I2C、平台、MDIO 等) 上的设备。这些属性是 reg、#address-cells 和 #size-cells, 它们的用途是在其所在总线上进行设备寻址。也就是说, 主要的寻址属性是 reg, 这是一个通用属性, 其含义取决于设备所在的总线。size-cell 和 address-cell 的前缀 #(sharp) 可以翻译为 length。

每个可寻址设备都具有 reg 属性, 该属性是 reg = <address0 size0 [address1 size1] [address2 size2] ...> 形式的元组列表, 其中每个元组代表设备使用的地址范围。#size-cells(长度单元) 指示使用多少个 32 位单元来表示大小, 如果与大小无关, 则可以是 0。而 #address-cells(地址单元) 指示用多少个 32 位单元来表示地址。换句话说, 每个元组的地址元素根据 #address-cell 来解释; 长度元素也是如此, 它根据 #size-cell 进行解释。

实际上, 可寻址设备继承自它们父节点的 #size-cell 和 #address-cell, 父节点代表总线控制器。指定设备中存在 #size-cell 和 #address-cell 不会影响设备本身, 但影响其子设备。换句话说, 在解释给定节点的 reg 属性之前, 必须知道父节点 #address-cells 和 #size-cells 的值。父节点可以自由定义适用于设备子节点 (孩子) 的寻址方案。

5.2.1 SPI 和 I^2C 寻址

SPI 和 I^2C 设备都属于非内存映射设备, 因为它们的地址对 CPU 不可访问。而父设备的驱动程序 (总线控制器驱动程序) 将代表 CPU 执行间接访问。每个 I^2C/SPI 设备都表示为设备所在 I^2C/SPI 总线节点的子节点。对于非存储器映射的设备,#size-cells 属性为 0, 寻址元组中的 size 元素为空。这意味着这种设备的 reg 属性总是只有一个单元:

例子

```
&i2c3 {
    [...]
    status = "okay";

    temperature-sensor@49 {
        compatible = "national,lm73";
        reg = <0x49>;
    };

    pcf8523: rtc@68 {
        compatible = "nxp,pcf8523";
        reg = <0x68>;
    };
};

&ecspi1 {
    fsl,spi-num-chipselects = <3>;
    cs-gpios = <&gpio5 17 0>, <&gpio5 17 0>, <&gpio5 17 0>;
    status = "okay";
    [...]

    ad7606r8_0: ad7606r8@1 {
        compatible = "ad7606-8";
        reg = <1>;
        spi-max-frequency = <1000000>;
        interrupt-parent = <&gpio4>;
        interrupts = <30 0x0>;
        convst-gpio = <&gpio6 18 0>;
    };
};
```

查看 arch/arm/boot/dts/imx6qdl.dtsi 中的 SoC 级文件就会发现: 在 i2c 和 spi 节点中,#size-cells 和 #address-cells 分别设置为 0(前者) 和 1(后者), 它们分别是前面所列 I2C 和 SPI 设备的父节点。这有助于理解它们的 reg 属性,reg 属性只是一个保存地址值的单元格。

I^2C 设备的 reg 属性用于指定总线上设备的地址。对于 SPI 设备,reg 表示从控制器节点所具有的芯片选择

列表中分配给设备的芯片选择线的索引。例如, 对于 ad7606r8 ADC, 芯片选择线索引是 1, 对应于 cs-gpios 中的 <&gpio5 17 0>,cs-gpios 是控制器节点的芯片选择列表。

为什么使用 I^2C/SPI 节点的 phandle: 因为 I^2C/SPI 设备应在开发板文件 (.dts) 中声明, 而 I2C/SPI 总线控制器在 SoC 级文件 (.dtsi) 中声明。

5.2.2 平台设备寻址

本节介绍简单的内存映射设备, 其内存可由 CPU 访问。在这里,reg 属性仍然定义设备的地址, 这是可以访问设备的内存区域列表。每个区域用单元格元组表示, 其中第一个单元格是内存区域的基地址, 第二个元组是该区域的大小。它具有的形式是 reg = <base0 length0 [base1 length1] [address2 length2] ...>。每个元组代表设备使用的地址范围。

在现实世界中, 人们应该在知道其他两个属性 (#size-cells 和 #address-cells) 值的情况下解释 reg 属性。#size-cells 指出在每个子 reg 元组中长度字段有多大。#address-cell 也一样, 它说明指定一个地址必须使用多少个单元。

这种设备应该在具有特殊值 compatible = "simple-bus" 的节点内声明, 这意味着简单的内存映射总线, 没有特定的处理和驱动程序:

例子

```
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    aips-bus@02000000 { /* AIPS1 */
        compatible = "fsl,aips-bus", "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x02000000 0x100000>;
        [...];

        spba-bus@02000000 {
            compatible = "fsl,spba-bus", "simple-bus";
            #address-cells = <1>;
            #size-cells = <1>;
            reg = <0x02000000 0x40000>;
            [...]
            ecspi1: ecspi@02008000 {
                #address-cells = <1>;
                #size-cells = <0>;
                compatible = "fsl,imx6q-ecspi", "fsl,imx51-ecspi";
                reg = <0x02008000 0x4000>;
                [...]
            };
            i2c1: i2c@021a0000 {
```

```
26              #address-cells = <1>;
27              #size-cells = <0>;
28              compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
29              reg = <0x021a0000 0x4000>;
30              [...]
31          };
32      };
33  };
```

在前面的示例中, 父节点 compatible 的属性值为 simple-bus, 其子节点将被注册为平台设备。设置 #size-cells = <0> 也能够看到 I2C 和 SPI 总线控制器怎样改变其子节点的寻址方式, 因为这与它们无关。从内核设备树文档可以查找所有绑定信息:Documentation/devicetree/bindings/。

5.3 处理资源

驱动程序的主要目的是处理和管理设备, 并且大部分时间将其功能展现给用户空间。这里的目标是收集设备的配置参数, 特别是资源 (存储区、中断线、DMA 通道、时钟等)。

以下是本节中将要使用的设备节点。它是在 arch/arm/boot/dts/imx6qdl.dtsi 中定义的 i.MX6 UART 设备节点:

uart

```
1  uart1: serial@02020000 {
2      compatible = "fsl,imx6q-uart","fsl,imx21-uart";
3      reg = <0x02020000 0x4000>;
4      interrupts = <0 26 IRQ_TYPE_LEVEL_HIGH>;
5      clocks = <&clks IMX6QDL_CLK_UART_IPG>,<&clks IMX6QDL_CLK_UART_SERIAL>;
6      clock-names = "ipg", "per";
7      dmas = <&sdma 25 4 0>, <&sdma 26 4 0>;
8      dma-names = "rx", "tx";
9      status = "disabled";
10 };
```

5.3.1 命名资源的概念

当驱动程序期望某种类型的资源列表时, 由于编写开发板设备树的人通常不是写驱动程序的人, 因此不能保证该列表是以驱动程序期望的方式排序。例如, 驱动程序可能期望其设备节点具有 2 条 IRQ 线路, 一条用于索引 0 处的 Tx 事件, 另一条用于索引 1 处的 Rx。如果这种顺序得不到满足会发生什么情况? 驱动就会发生异常行为。为了避免这种不匹配, 引入了命名资源 (clock、irq、dma、reg) 的概念。它由定义资源列表和命名组成, 因此无论索引是什么, 给定的名称总将与资源相匹配。

命名资源的相应属性如下。

- reg-names:reg 属性中的内存区域列表。
- clock-names:clocks 属性中命名 clocks。
- interrupt-names: 为 interrupts 属性中的每个中断指定一个名称。
- dma-names: 用于 dma 属性。

现在, 创建一个假的设备节点以解释上述概念:

例子

```
fake_device {
    compatible = "packt,fake-device";
    reg = <0x02020000 0x4000>, <0x4a064800 0x200>, <0x4a064c00 0x200>;
    reg-names = "config", "ohci", "ehci";
    interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>, <0 67 IRQ_TYPE_LEVEL_HIGH>;
    interrupt-names = "ohci", "ehci";
    clocks = <&clks IMX6QDL_CLK_UART_IPG>, <&clks IMX6QDL_CLK_UART_SERIAL>;
    clock-names = "ipg", "per";
    dmas = <&sdma 25 4 0>, <&sdma 26 4 0>;
    dma-names = "rx", "tx";
};
```

驱动程序中提取每个命名资源的代码如下所示:

提取命名资源

```
struct resource *res1, *res2;
res1 = platform_get_resource_byname(pdev, IORESOURCE_MEM, "ohci");
res2 = platform_get_resource_byname(pdev, IORESOURCE_MEM, "config");
struct dma_chan *dma_chan_rx, *dma_chan_tx;
dma_chan_rx = dma_request_slave_channel(&pdev->dev, "rx");
dma_chan_tx = dma_request_slave_channel(&pdev->dev, "tx");
int txirq, rxirq;
txirq = platform_get_irq_byname(pdev, "ohci");
rxirq = platform_get_irq_byname(pdev, "ehci");
structclk *clck_per, *clk_ipg;
clk_ipg = devm_clk_get(&pdev->dev, "ipg");
clk_ipg = devm_clk_get(&pdev->dev, "pre");
```

这样, 就可以确保把正确的名字映射到正确的资源上, 而不用再使用索引了。

5.3.2 访问寄存器

在这里, 驱动程序将占用内存区域, 并将其映射到虚拟地址空间。第 11 章将进一步讨论这个问题。

例子

```
1  struct resource *res;
2  void __iomem *base;
3  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
4  /*
5  * 这里使用request_mem_region(res->start,
6  resource_size(res), pdev->name)
7  * 和ioremap(iores->start, resource_size(iores)请
8  求和映射内存区域
9  *
10 * 这些功能在第11章中讨论
11 */
12 base = devm_ioremap_resource(&pdev->dev, res);
13 if (IS_ERR(base))
14    return PTR_ERR(base);
```

`platform_get_resource()` 将根据第一个 (索引 0)reg 赋值中提供的内存区域设置 struct res 的开始和结束字段。请记住 `platform_get_resource()` 的最后一个参数代表资源索引。在前面的示例中,0 指定了该资源类型的第一个值, 以防在 DT 节点中为设备分配多个存储区域。在这个例子中,reg = <0x02020000 0x4000> 表示分配的区域从物理地址 0x02020000 开始, 大小为 0x4000 字节。`platform_get_resource()` 将设置 res.start = 0x02020000 和 res.end = 0x02023fff。

5.3.3 处理中断

中断接口实际上分为两部分, 消费者端和控制器端。DT 中用 4 个属性描述中断连接。

控制器是为消费者提供中断线的设备。在控制器端有以下属性。

- interrupt-controller: 为了将设备标记为中断控制器而应该定义的空 (布尔) 属性。
- #interrupt-cells: 这是中断控制器的属性。它指出为该中断控制器指定一个中断要使用多少个单元。

消费者是生成中断的设备。消费者绑定需要以下属性。

- interrupt-parent: 对于产生中断的设备节点, 这个属性包含指向设备所连接的中断控制器节点的指针 phandle。如果省略, 则设备从其父节点继承该属性。
- interrupts: 它是中断说明符。

中断绑定和中断说明符与中断控制器设备相关联。通过 #interrupt-cells 属性定义的中断输入单元数量取决于中断控制器, 这是唯一的决定因素。对于 i.MX6, 中断控制器是全局中断控制器 (Global Interrupt Controller,GIC)。在 Documentation/devicetree/bindings/arm/gic.txt 中很好地解释了其绑定。

1. 中断处理程序

这包括从 DT 获取 IRQ 号码, 映射到 Linux IRQ, 再为其注册函数回调。执行此操作的驱动程序代码非常简单:

example

```
1 int irq = platform_get_irq(pdev, 0);
2 ret = request_irq(irq, imx_rxint, 0,
3 dev_name(&pdev->dev), sport);
```

platform_get_irq() 调用将返回中断号,devm_request_irq() 可以用这个中断号 (之后 IRQ 在/proc/interrupts 中是可见的)。

第二个参数 0 表示需要在设备节点中指定的第一个中断。如果有多个中断, 则可以根据需要的中断更改这个索引, 或者只使用命名的资源。在前面的例子中, 设备节点包含中断说明符, 像下面这样:

example

```
1     interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>;
```

- 根据 ARM GIC, 第一个单元说明中断类型。
- 0: 共享外设中断 (SPI), 用于核间共享的中断信号, 可由 GIC 路由至任意核。
- 1: 专用外设中断 (PPI), 专用于单核的中断信号。
- 第二个单元格保存中断号。该中断号取决于中断线是 PPI 还是 SPI。
- 第三个单元, 这里的 `IRQ_TYPE_LEVEL_HIGH` 代表感知级别。所有可用的感知级别在 include/linux/irq.h 中定义。

2. 中断控制器代码

interrupt-controller 属性用于将设备声明为中断控制器。#interruptcells 属性定义必须使用多少个单元格来定义单个中断线。

5.4 平台驱动程序和 DT

平台驱动程序也使用 DT。也就是说, 这是现在推荐的处理平台设备的方法, 不再需要使用开发板文件, 甚至不需要在设备的属性更改时重新编译内核。可否记得, 在第 5 章讨论了 OF 匹配样式, 这是一种基于 DT 的匹配机制。下面介绍它的工作原理。

“OF 匹配风格” 是平台核心执行的第一种匹配机制, 以匹配设备及其驱动程序。它使用设备树的 compatible 属性来匹配 of_match_table 中的设备项, 设备项是 struct driver 子结构的一个字段。每个设备节点都有 compatible 属性, 它是字符串或字符串列表。任何驱动程序只要声明 compatible 属性中列出的字符串之一, 就将触发匹配, 并看到其 probe 函数执行。

DT 匹配项在内核中被描述为 struct of_device_id 结构的实例, 该结构定义在 linux/mod_devicetable.h 中, 如下所示:

example

```
1     // 我们只对结构的最后两个元素感兴趣
2     struct of_device_id {
```

```
3         [...]
4         char compatible[128];
5         const void *data;
6     }
```

该结构中每个元素的含义如下。

- char compatible [128]: 这是用来匹配 DT 中设备节点兼容属性的字符串。它们必须完全相同才算匹配。
- const void * data: 这可以指向任何结构, 这个结构可以用作每个设备类型的配置数据。

由于 of_match_table 是指针, 因此可以传递 struct of_device_id 数组, 使驱动程序兼容多个设备:

example

```
1 static const struct of_device_id imx_uart_dt_ids[] = {
2     { .compatible = "fsl,imx6q-uart", },
3     { .compatible = "fsl,imx1-uart", },
4     { .compatible = "fsl,imx21-uart", },
5     { /* sentinel */ }
6 };
```

一旦在驱动程序的子结构中填充了 ID 数组, 就必须把它传递到平台驱动程序的 of_match_table 字段:

第六章 字符设备驱动程序

本章的目的是编写一个完整的字符设备驱动。我们开发一个字符驱动是因为这一类适合大部分简单硬件设备。字符驱动也比块驱动易于理解（我们在后续章节接触）。我们的最终目的是编写一个模块化的字符驱动，但是我们不会在本章讨论模块化的事情。

贯串本章，我们展示从一个真实设备驱动提取的代码片段：scull（Simple Character Utility for Loading Localities）。scull 是一个字符驱动，操作一块内存区域好像它是一个设备。在本章，因为 scull 的这个怪特性，我们可互换地使用设备这个词和 “scull 使用的内存区”。

scull 的优势在于它不依赖硬件。scull 只是操作一些从内核分配的内存。任何人都可以编译和运行 scull，并且 scull 在 Linux 运行的体系结构中可移植。另一方面，这个设备除了演示内核和字符驱动的接口和允许用户运行一些测试之外，不做任何有用的事情。

6.1 scull 的设计

编写驱动的第一步是定义驱动将要提供给用户程序的能力（机制）。因为我们的 “设备” 是计算机内存的一部分，我们可自由做我们想做的事情。它可以是一个顺序的或者随机读写的设备，一个或多个设备，等等。

为使 scull 作为一个模板来编写真实设备的真实驱动，我们将展示给你如何在计算机内存上实现几个设备抽象，每个有不同的特点。

scull 源码实现下面的设备。模块实现的每种设备都被引用做一种类型。

- *scull*0 ~ *scull*3：4 个设备，每个由一个全局永久的内存区组成。全局意味着如果设备被多次打开，设备中含有的数据由所有打开它的文件描述符共享。永久意味着如果设备关闭又重新打开，数据不会丢失。这个设备用起来有意思，因为它可以用惯常的命令来读写和测试，例如 cp，cat，以及 I/O 重定向。
- *scullpipe*0 ~ *scullpipe*3：4 个 FIFO（先入先出）设备，行为类似管道。一个进程读的内容来自另一个进程所写的。如果多个进程读同一个设备，它们竞争数据。scullpipe 的内部将展示阻塞读写和非阻塞读写如何实现，而不必采取中断。尽管真实的驱动使用硬件中断来同步它们的设备，阻塞和非阻塞操作的主题是重要的并且与中断处理是分开的。
- scullsingle
- scullpriv
- sculluid
- scullwuid

  这些设备与 scull0 相似，但是在什么时候允许打开上有一些限制。第一个（snullsingle）只允许一次一个进程使用驱动，而 scullpriv 对每个虚拟终端（或者 X 终端会话）是私有的，因为每个控制台/终端上的进程有不同的内存区。sculluid 和 scullwuid 可以多次打开，但是一次只能是一个用户；前者返回一个 “设备忙” 错误，如果另一个用户锁着设备，而后者实现阻塞打开。这些 scull 的变体可能看来混淆了策略和机制，但是它们值得看看，因为一些实际设备需要这类管理。

每个 scull 设备演示了驱动的不同特色，并且呈现了不同的难度。本章涉及 scull0 到 scull3 的内部结构。

6.2 主设备号和次设备号

字符设备通过文件系统中的名字来读写。那些名字称为文件系统的特殊文件，或者设备文件，或者文件系统的简单节点；惯例上它们位于 `/dev` 目录。字符驱动的特殊文件由使用 `ls -l` 的输出的第一列的 “c” 标识。块设备也出现在 `/dev` 中，但是它们由 “b” 标识。本章集中在字符设备，但是下面的很多信息也适用于块设备。

如果执行 `ls -l` 命令，你会看到在设备文件项中有 2 个数（由一个逗号分隔）在最后修改日期前面，这里通常是文件长度出现的地方。这些数字是给特殊设备的主次设备编号。下面的列表显示了一个典型系统上出

现的几个设备。它们的主编号是 1, 4, 7, 和 10, 而次编号是 1, 3, 5, 64, 65, 和 129。

列表

```
crw-rw-rw- 1 root root 1,3 Apr 11 2002 null
crw------- 1 root root 10,1 Apr 11 2002 psaux
crw------- 1 root root 4,1 Oct 28 03:04 tty1
crw-rw-rw- 1 root tty 4,64 Apr 11 2002 ttys0
crw-rw---- 1 root uucp 4,65 Apr 11 2002 ttyS1
crw--w---- 1 vcsa tty 7,1 Apr 11 2002 vcs1
crw--w---- 1 vcsa tty 7,129 Apr 11 2002 vcsa1
crw-rw-rw- 1 root root 1,5 Apr 11 2002 zero
```

传统上, 主编号标识设备相连的驱动. 例如, /dev/null 和 /dev/zero 都由驱动 1 来管理，而虚拟控制台和串口终端都由驱动 4 管理；同样，vcs1 和 vcsa1 设备都由驱动 7 管理。现代 Linux 内核允许多个驱动共享主编号，但是你看到的大部分设备仍然按照一个主编号一个驱动的原则来组织。

次编号被内核用来决定引用哪个设备。依据你的驱动是如何编写的（如同我们下面见到的），你可以从内核得到一个你的设备的直接指针，或者可以自己使用次编号作为本地设备数组的索引。不论哪个方法，内核自己几乎不知道次编号的任何事情，除了它们指向你的驱动实现的设备。

6.2.1 设备编号的内部表示

在内核中，dev_t 类型（在 <linux/types.h> 中定义）用来保存设备编号——包括主设备号和次设备号。对于 2.6.0 内核，dev_t 是 32 位的数值，其中 12 位用作主编号，其余的 20 位用作次编号。你的代码应当对于设备编号的组织从不做任何假设；相反，应当利用在 <linux/kdev_t.h> 中的一套宏定义。为获得一个 dev_t 的主或者次编号，使用：

设备编号

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

相反，如果你有主次编号，需要将其转换为一个 dev_t ，使用：

转换成 *dev_t*

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

6.2.2 分配和释放设备编号

在建立一个字符驱动时你的驱动需要做的第一件事是获取一个或多个设备编号来使用。为此目的的必要的函数是 `register_chrdev_region`，在 <linux/fs.h> 中声明：

获取设备号

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

这里，first 是你要分配的起始设备编号。first 的次编号部分常常是 0，但是没有要求必须这样。count 是你请求的连续设备编号的总数。注意，如果 count 太大，你要求的范围可能溢出到下一个次编号；但是只要你要求的编号范围可用，一切都仍然会正确工作。最后，name 是应当连接到这个编号范围的设备的名字；它会出现在 /proc/devices 和 sysfs 中。

如同大部分内核函数，如果分配成功进行，`register_chrdev_region` 的返回值是 0 。出错的情况下，返回一个负的错误码，你不能读写请求的编号区域。

如果你确实事先知道你需要哪个设备编号，`register_chrdev_region` 工作会工作的很好。然而，你常常不会知道你的设备使用哪个主编号；在 Linux 内核开发社团中一直努力使用动态分配设备编号。内核会乐于动态为你分配一个主编号，但是你必须使用一个不同的函数来请求这个分配。

动态分配设备号

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

使用这个函数，dev 是一个只输出的参数，它在函数成功完成时持有你的分配范围的第一个数。firstminor 应当是请求的第一个要用的次编号；它常常是 0 。count 和 name 参数如同给 `request_chrdev_region` 的一样。

不管你任何分配你的设备编号，你应当在不再使用它们时释放它。设备编号的释放使用：

释放编号

```
1 void unregister_chrdev_region(dev_t first, unsigned int count);
```

调用 `unregister_chrdev_region` 的地方常常是你的模块的 cleanup 函数。上面的函数分配设备编号给你的驱动使用，但是它们不告诉内核你实际上会对这些编号做什么。在用户空间程序能够读写这些设备号中一个之前，你的驱动需要连接它们到它的实现设备操作的内部函数上。我们将描述如何简短完成这个连接，但首先顾及一些必要的枝节问题。

6.2.3 主编号的动态分配

一些主设备编号是静态分派给最普通的设备的。一个这些设备的列表在内核源码树的 Documentation/devices.txt 中。分配给你的新驱动使用一个已经分配的静态编号的机会很小，但是，并且新编号没在分配。因此，作为一个驱动编写者，你有一个选择：你可以简单地捡一个看来没有用的编号，或者你以动态方式分配主编号。只要你是你的驱动的唯一用户就可以捡一个编号用；一旦你的驱动更广泛的被使用了，一个随机捡来的主编号将导致冲突和麻烦。

因此，对于新驱动，我们强烈建议你使用动态分配来获取你的主设备编号，而不是随机选取一个当前空闲的编号。换句话说，你的驱动应当几乎肯定地使用 `alloc_chrdev_region`，不是 `register_chrdev_region` 。

动态分配的缺点是你无法提前创建设备节点，因为分配给你的模块的主编号会变化。对于驱动的正常使用，这不是问题，因为一旦编号分配了，你可从 /proc/devices 中读取它。

为使用动态主编号来加载一个驱动，因此，可使用一个简单的脚本来代替调用 insmod ，在调用 insmod 后，读取 /proc/devices 来创建特殊文件。

一个典型的 /proc/devices 文件看来如下：

/proc/devices

```
Character devices:
1  mem
2  pty
3  ttyp
4  ttyS
6  lp
7  vcs
10 misc
13 input
14 sound

Block devices:
2  fd
8  sd
11 sr
65 sd
66 sd
```

因此加载一个已经安排了一个动态编号的模块的脚本，可以使用一个工具来编写，如 awk ，来从 /proc/devices 获取信息以创建 /dev 中的文件。

下面的脚本，snull_load，是 scull 发布的一部分。以模块发布的驱动的用户可以从系统的 rc.local 文件中调用这样一个脚本，或者在需要模块时手工调用它。

加载脚本

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# 使用传入到该脚本的所有参数调用 insmod，同时使用路径名来指定模块位置，
# 这是因为新的modutils默认不会在当前目录中查找模块。
```

```
/sbin/insmod ./$module.ko $* || exit 1

# 删除原有节点
rm -f /dev/${device}[0-3]

major=$(awk "\\$2==\"$module\" {print \\$1}" /proc/devices)
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# 给定适当的组属性以及许可，并修改属组。
# 并非所有的发行版都具有staff组，有些有wheel组。
group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

这个脚本可以通过重定义变量和调整 mknod 行来适用于另外的驱动。这个脚本仅仅展示了创建 4 个设备，因为 4 是 scull 源码中缺省的。

脚本的最后几行可能有些模糊：为什么改变设备的组和模式？理由是这个脚本必须由超级用户运行，因此新建的特殊文件由 root 拥有。许可位缺省的是只有 root 有写权限，而任何人都可以读。通常，一个设备节点需要一个不同的读写策略，因此在某些方面别人的读写权限必须改变。我们的脚本缺省是给一个用户组读写，但是你的需求可能不同。

还有一个 scull_unload 脚本来清理 /dev 目录并去除模块。作为对使用一对脚本来加载和卸载的另外选择，你可以编写一个 init 脚本，准备好放在你的发布使用这些脚本的目录中。作为 scull 源码的一部分，我们提供了一个相当完整和可配置的 init 脚本例子，称为 scull.init；它接受传统的参数——start，stop，和 restart ——并且完成 scull_load 和 scull_unload 的角色。

如果反复创建和销毁 /dev 节点，听来过分了，有一个有用的办法。如果你在加载和卸载单个驱动，你可以在你第一次使用你的脚本创建特殊文件之后，只使用 rmmod 和 insmod：这样动态编号不是随机的。并且你每次都可以使用所选的同一个编号，如果你不加载任何别的动态模块。在开发中避免长脚本是有用的。但是这个技巧，显然不能扩展到一次多于一个驱动。安排主编号最好的方式，我们认为，是缺省使用动态分配，而留给自己在加载时指定主编号的选项权，或者甚至在编译时。scull 实现以这种方式工作；它使用一个全局变量，scull_major，来持有选定的编号（还有一个 scull_minor 给次编号）。这个变量初始化为 SCULL_MAJOR，定义在 scull.h 。发布的源码中的 SCULL_MAJOR 的缺省值是 0，意思是“使用动态分配”。用户可以接受缺省值或者选择一个特殊主编号，或者在编译前修改宏定义或者在 insmod 命令行指定一个值给 scull_major 。最后，通过使用 scull_load 脚本，用户可以在 scull_load 的命令行传递参数给 insmod 。

下面是 scull.c 中用来获取主设备号的代码：

获取主设备号

```
1  if (sucll_major) {
2      dev = MKDEV(scull_major, scull_minor);
3      result = register_chrdev_region(dev, scull_nr_devs, "scull");
4  } else {
5      result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
6      scull_major = MAJOR(dev);
7  }
8  if (result < 0) {
9      printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
10     return result;
11 }
```

本书使用的几乎所有例子驱动使用类似的代码来分配它们的主编号。

6.3 一些重要的数据结构

如同你想象的，注册设备编号仅仅是驱动代码必须进行的诸多任务中的第一个。我们将很快看到其他重要的驱动组件，但首先需要涉及一个别的。大部分的基础性的驱动操作包括 3 个重要的内核数据结构，称为 file_operations，file，和 inode 。需要对这些结构的基本了解才能够做大量感兴趣的事情，因此我们现在在进入如何实现基础性驱动操作的细节之前，会快速查看每一个。

6.3.1 文件操作

到现在，我们已经保留了一些设备编号给我们使用，但是我们还没有连接任何我们设备操作到这些编号上。file_operation 结构是一个字符驱动如何建立这个连接。这个结构，定义在 <linux/fs.h> ，是一个函数指针的集合。每个打开文件（内部用一个 file 结构来代表，稍后我们会查看）与它自身的函数集合相关连（通过包含一个称为 f_op 的成员，它指向一个 file_operations 结构）。这些操作大部分负责实现系统调用，因此，命名为 open，read，等等。我们可以认为文件是一个 “对象” 并且其上的函数操作称为它的 “方法”，使用面向对象编程的术语来表示一个对象声明的用来操作对象的动作。这是我们在 Linux 内核中看到的第一个面向对象编程的现象，后续章中我们会看到更多。

传统上，一个 file_operation 结构或者其一个指针称为 fops（或者它的一些变体）。结构中的每个成员必须指向驱动中的函数，这些函数实现一个特别的操作，或者对于不支持的操作留置为 NULL 。当指定为 NULL 指针时内核的确切的行为是每个函数不同的，如同本节后面的列表所示。

下面的列表介绍了一个应用程序能够在设备上调用的所有操作。我们已经试图保持列表简短，这样它可作为一个参考，只是总结每个操作和在 NULL 指针使用时的缺省内核行为。

在你通读 file_operations 方法的列表时，你会注意到不少参数包含字串 __user 。这种注解是一种文档形式，注意，一个指针是一个不能被直接解引用的用户空间地址。对于正常的编译，__user 没有效果，但是它可被外部检查软件使用来找出对用户空间地址的错误使用。

本章剩下的部分，在描述一些其他重要数据结构后，解释了最重要操作的角色并且给了提示，告诫和真实代码例子。我们推迟讨论更复杂的操作到后面章节，因为我们还不准备深入如内存管理，阻塞操作，和异步通知。

- `scull module *owner`：第一个 file_operations 字段并不是一个操作；相反，它是指向 “拥有” 该结构

的模块的指针。内核使用这个字段以避免在模块的操作正在被使用时卸载该模块。几乎在所有的情况下，该成员都会被初始化为 `THIS_MODULE` ，它是定义在 <linux/module.h> 中的一个宏。

- `loff_t (*llseek)(struct file *, loff_t, int);` : llseek 方法用作改变文件中的当前读/写位置，并且新位置作为（正的）返回值。loff_t 参数是一个 “long offset’’，并且就算在 32 位平台上也至少 64 位宽。错误由一个负返回值指示。如果这个函数指针是 NULL，seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器。
- `ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);` :用来从设备中获取数据。在这个位置的一个空指针导致 read 系统调用以 -EINVAL（"Invalid argument"）失败。一个非负返回值代表了成功读取的字节数（返回值是一个 “signed size” 类型，常常是目标平台本地的整数类型）。
- `ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);` : 初始化一个异步读——可能在函数返回前不结束的读操作。如果这个方法是 NULL，所有的操作会由 read 代替进行（同步地）。
- `ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);` :发送数据给设备。如果 NULL，-EINVAL 返回给调用 write 系统调用的程序。如果非负，返回值代表成功写的字节数。
- `ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);` : 初始化设备上的一个异步写。
- `int (*readdir)(struct file *, void *, filldir_t);` : 对于设备文件这个成员应当为 NULL；它用来读取目录，并且仅对文件系统有用。
- `unsigned int (*poll)(struct file *, struct poll_table_struct *);` : poll 方法是 3 个系统调用的后端：poll，epoll，和 select ，都用作查询对一个或多个文件描述符的读或写是否会阻塞。poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的，并且，可能地，提供给内核信息用来使调用进程睡眠直到 I/O 变为可能。如果一个驱动的 poll 方法为 NULL，设备假定为不阻塞地可读可写。
- `int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);` : ioctl 系统调用提供了发出设备特定命令的方法（例如格式化软盘的一个磁道，这不是读也不是写）。另外，几个 ioctl 命令被内核识别而不必引用 fops 表。如果设备不提供 ioctl 方法，对于任何未事先定义的请求（-ENOTTY，“设备无这样的 ioctl’’），系统调用返回一个错误。
- `int (*mmap)(struct file *, struct vm_area_struct *);` : mmap 用来请求将设备内存映射到进程的地址空间。如果这个方法是 NULL，mmap 系统调用返回 -ENODEV 。
- `int (*open)(struct inode *, struct file *);` : 尽管这常常是对设备文件进行的第一个操作，不要求驱动声明一个对应的方法。如果这个项是 NULL，设备打开一直成功，但是你的驱动不会得到通知。
- `int (*flush)(struct file *);` : flush 操作在进程关闭它的设备文件描述符的拷贝时调用；它应当执行（并且等待）设备的任何未完成的操作。这个必须不要和用户查询请求的 fsync 操作混淆了。当前，flush 在很少驱动中使用；SCSI 磁带驱动使用它，例如，为确保所有写的数据在设备关闭前写到磁带上。如果 flush 为 NULL，内核简单地忽略用户应用程序的请求。
- `int (*release)(struct inode *, struct file *);` :在文件结构被释放时引用这个操作。如同 open，release 可以为 NULL 。
- `int (*fsync)(struct file *, struct dentry *, int);` : 这个方法是 fsync 系统调用的后端，用户调用来刷新任何挂着的数据。如果这个指针是 NULL，系统调用返回 -EINVAL 。
- `int (*aio_fsync)(struct kiocb *, int);` : 这是 fsync 方法的异步版本。
- `int (*fasync)(int, struct file *, int);` : 这个操作用来通知设备它的 FASYNC 标志的改变。异步通知是一个高级的主题，在第 6 章中描述。这个成员可以是 NULL 如果驱动不支持异步通知。
- `int (*lock)(struct file *, int, struct file_lock *);` : lock 方法用来实现文件加锁；加锁对常规文件是必不可少的特性，但是设备驱动几乎从不实现它。
- `ssize_t (*readv)(struct file *, const struct iovec *, unsigned long, loff_t *);`
- `ssize_t (*writev)(struct file *, const struct iovec *, unsigned long, loff_t *);`

这些方法实现发散/汇聚读和写操作。应用程序偶尔需要做一个包含多个内存区的单个读或写操作；这些系统调用允许它们这样做而不必对数据进行额外拷贝。如果这些函数指针为 NULL ，read 和 write 方法被调用（可能多于一次）。

- `ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);`：这个方法实现 sendfile 系统调用的读，使用最少的拷贝从一个文件描述符搬移数据到另一个。例如，它被一个需要发送文件内容到一个网络连接的 web 服务器使用。设备驱动常常使 sendfile 为 NULL。
- `ssize_t (*sendpage)(struct file *, struct page *, int, size_t, loff_t *, int);`：sendpage 是 sendfile 的另一半；它由内核调用来发送数据，一次一页，到对应的文件。设备驱动实际上不实现 sendpage 。
- `unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);`
  这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中。这个任务通常由内存管理代码进行；这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求。大部分驱动可以置这个方法为 NULL 。
- `int (*check_flags)(int)`：这个方法允许模块检查传递给 `fnctl(F_SETFL...)` 调用的标志。
- `int (*dir_notify)(struct file *, unsigned long);`：这个方法在应用程序使用 fcntl 来请求目录改变通知时调用。只对文件系统有用；驱动不需要实现 dir_notify。

scull 设备驱动只实现了最重要的设备方法。它的 `file_operations` 结构是如下初始化的：

初始化

```
1  struct file_operations scull_fops = {
2    .owner = THIS_MODULE,
3    .llseek = scull_llseek,
4    .read = scull_read,
5    .write = scull_write,
6    .ioctl = scull_ioctl,
7    .open = scull_open,
8    .release = scull_release,
9  };
```

这个声明使用标准的 C 标记式结构初始化语法。这个语法是首选的，因为它使驱动在结构定义的改变之间更加可移植，并且，有争议地，使代码更加紧凑和可读。标记式初始化允许结构成员重新排序；在某种情况下，真实的性能提高已经实现，通过安放经常使用的成员的指针在相同硬件高速存储行中。

6.3.2 文件结构

struct file，定义于 <linux/fs.h>，是设备驱动中第二个最重要的数据结构。注意 file 与用户空间程序的 FILE 指针没有任何关系。一个 FILE 定义在 C 库中，从不出现在内核代码中。一个 struct file，另一方面，是一个内核结构，从不出现在用户程序中。

文件结构代表一个打开的文件。（它不特定给设备驱动；系统中每个打开的文件有一个关联的 struct file 在内核空间）。它由内核在 open 时创建，并传递给在文件上操作的任何函数，直到最后的关闭。在文件的所有实例都关闭后，内核释放这个数据结构。

在内核源码中，struct file 的指针常常称为 file 或者 filp（“file pointer”）。我们将一直称这个指针为 filp 以

避免和结构自身混淆。因此，file 指的是结构，而 filp 是结构指针。

struct file 的最重要成员在这展示。如同在前一节，第一次阅读可以跳过这个列表。但是，在本章后面，当我们面对一些真实 C 代码时，我们将更详细讨论这些成员。

- `mode_t f_mode;`：文件模式确定文件是可读的或者是可写的（或者都是），通过位 FMODE_READ 和 FMODE_WRITE。你可能想在你的 open 或者 ioctl 函数中检查这个成员的读写许可，但是你不需要检查读写许可，因为内核在调用你的方法之前检查。当文件还没有为那种读写而打开时读或写的企图被拒绝，驱动甚至不知道这个情况。
- `loff_t f_pos;`：当前读写位置。loff_t 在所有平台都是 64 位（在 gcc 术语里是 long long）。驱动可以读这个值，如果它需要知道文件中的当前位置，但是正常地不应该改变它；读和写应当使用它们作为最后参数而收到的指针来更新一个位置，代替直接作用于 filp->f_pos 。这个规则的一个例外是在 llseek 方法中，它的目的就是改变文件位置。
- `unsigned int f_flags;`：这些是文件标志，例如 O_RDONLY，O_NONBLOCK，和 O_SYNC 。驱动应当检查 O_NONBLOCK 标志来看是否是请求非阻塞操作，其他标志很少使用。特别地，应当检查读/写许可，使用 f_mode 而不是 f_flags 。所有的标志在头文件 <linux/fcntl.h> 中定义。
- `struct file_operations *f_op;`：和文件关联的操作。内核安排指针作为它的 open 实现的一部分，接着读取它当它需要分派任何的操作时。filp->f_op 中的值从不由内核保存为后面的引用；这意味着你可改变你的文件关联的文件操作，在你返回调用者之后新方法会起作用。例如，关联到主编号 1（/dev/null，/dev/zero，等等）的 open 代码根据打开的次编号来替代 filp->f_op 中的操作。这个做法允许实现几种行为，在同一个主编号下而不必在每个系统调用中引入开销。替换文件操作的能力是面向对象编程的“方法重载”的内核对等体。
- `void *private_data;`：open 系统调用设置这个指针为 NULL，在为驱动调用 open 方法之前。你可自由使用这个成员或者忽略它；你可以使用这个成员来指向分配的数据，但是接着你必须记住在内核销毁文件结构之前，在 release 方法中释放那个内存。private_data 是一个有用的资源，在系统调用间保留状态信息，我们大部分例子模块都使用它。
- `struct dentry *f_dentry;`：关联到文件的目录入口（dentry）结构。设备驱动编写者正常地不需要关心 dentry 结构，除了作为 filp->f_dentry->d_inode 读写 inode 结构。

真实结构有多几个成员，但是它们对设备驱动没有用处。我们可以安全地忽略这些成员，因为驱动从不创建文件结构；它们真实读写别处创建的结构。

6.3.3 inode 文件结构

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

inode 结构包含大量关于文件的信息。作为一个通用的规则，这个结构只有 2 个成员对于编写驱动代码有用：

- `dev_t i_rdev;`：对于代表设备文件的节点，这个成员包含实际的设备编号。
- `struct cdev *i_cdev;`：struct cdev 是内核的内部结构，代表字符设备；这个成员包含一个指针，指向这个结构。当节点指的是一个字符设备文件时。

内核作者添加了 2 个宏，可以用来从一个 inode 中获取主次编号：

宏

```
1 unsigned int iminor(struct inode *inode);
2 unsigned int imajor(struct inode *inode);
```

6.4 注册字符设备

如我们提过的，内核在内部使用类型 struct cdev 的结构来代表字符设备。在内核调用你的设备操作前，你编写分配并注册一个或几个这些结构。为此，你的代码应当包含 <linux/cdev.h>，这个结构和它的关联帮助函数定义在这里。

有 2 种方法来分配和初始化一个这些结构。如果你想在运行时获得一个独立的 cdev 结构，你可以为此使用这样的代码：

获取 *cdev* 结构

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

但是，偶尔你会想将 cdev 结构嵌入一个你自己的设备特定的结构；scull 这样做了。在这种情况下，你应当初始化你已经分配的结构，使用：

example

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

任一方法，有一个其他的 struct cdev 成员你需要初始化。像 file_operations 结构，struct cdev 有一个拥有者成员，应当设置为 THIS_MODULE 。一旦 cdev 结构建立，最后的步骤是把它告诉内核，调用：

example

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

这里，dev 是 cdev 结构，num 是这个设备响应的第一个设备号，count 是应当关联到设备的设备号的数目。常常 count 是 1，但是有多个设备号对应于一个特定的设备的情形。例如，设想 SCSI 磁带驱动，它允许用户空间来选择操作模式（例如密度），通过安排多个次编号给每一个物理设备。

在使用 cdev_add 是有几个重要事情要记住。第一个是这个调用可能失败。如果它返回一个负的错误码，你的设备没有增加到系统中。它几乎会一直成功，但是，并且带起了其他的点：cdev_add 一返回，你的设备就是“活的”并且内核可以调用它的操作。除非你的驱动完全准备好处理设备上的操作，你不应当调用 cdev_add 。

为从系统去除一个字符设备，调用：

example

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

显然，你不应当在传递给 cdev_del 后存取 cdev 结构。

6.4.1 scull 中的设备注册

在内部，scull 使用一个 struct scull_dev 类型的结构表示每个设备。这个结构定义为：

example

```
struct scull_dev {
    struct scull_qset *data; /* Pointer to first quantum set */
    int quantum; /* the current quantum size */
    int qset; /* the current array size */
    unsigned long size; /* amount of data stored here */
    unsigned int access_key; /* used by sculluid and scullpriv */
    struct semaphore sem; /* mutual exclusion semaphore */
    struct cdev cdev; /* Char device structure */
};
```

我们在遇到它们时讨论结构中的各个成员，但是现在，我们关注于 cdev，我们的设备与内核接口的 struct cdev。这个结构必须初始化并且如上所述添加到系统中；处理这个任务的 scull 代码是：

example

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

因为 cdev 结构嵌在 struct scull_dev 里面，cdev_init 必须调用来进行那个结构的初始化。

6.5 open 和 release

6.5.1 open 方法

open 方法提供给驱动来做任何的初始化来准备后续的操作。在大部分驱动中，open 应当进行下面的工作：

- 检查设备特定的错误（例如设备没准备好，或者类似的硬件错误）
- 如果它第一次打开，初始化设备
- 如果需要，更新 f_op 指针。

- 分配并填充要放进 filp->private_data 的任何数据结构

但是, 事情的第一步常常是确定打开哪个设备. 记住 open 方法的原型是:

example

```
1    int (*open)(struct inode *inode, struct file *filp);
```

inode 参数有我们需要的信息, 以它的 i_cdev 成员的形式, 里面包含我们之前建立的 cdev 结构. 唯一的问题是通常我们不想要 cdev 结构本身, 我们需要的是包含 cdev 结构的 scull_dev 结构. C 语言使程序员玩弄各种技巧来做这种转换; 但是, 这种技巧编程是易出错的, 并且导致别人难于阅读和理解代码. 幸运的是, 在这种情况下, 内核 hacker 已经为我们实现了这个技巧, 以 container_of 宏的形式, 在 <linux/kernel.h> 中定义:

example

```
1    container_of(pointer, container_type, container_field);
```

这个宏使用一个指向 container_field 类型的成员的指针, 它在一个 container_type 类型的结构中, 并且返回一个指针指向包含结构. 在 scull_open, 这个宏用来找到适当的设备结构:

example

```
1    struct scull_dev *dev; /* device information */
2    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
3    filp->private_data = dev; /* for other methods */
```

一旦它找到 scull_dev 结构, scull 在文件结构的 private_data 成员中存储一个它的指针, 为以后更易存取. 识别打开的设备的另外的方法是查看存储在 inode 结构的次编号. 如果你使用 register_chrdev 注册你的设备, 你必须使用这个技术. 确认使用 iminor 从 inode 结构中获取次编号, 并且确定它对应一个你的驱动真正准备好处理的设备.

scull_open 的代码 (稍微简化过) 是:

example

```
1  int scull_open(struct inode *inode, struct file *filp)
2  {
3      struct scull_dev *dev; /* device information */
4      dev = container_of(inode->i_cdev, struct scull_dev, cdev);
5      filp->private_data = dev; /* for other methods */
6      /* now trim to 0 the length of the device if open was write-only */
7      if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
```

```
8       {
9           scull_trim(dev); /* ignore errors */
10      }
11      return 0; /* success */
12  }
```

代码看来相当稀疏, 因为在调用 open 时它没有做任何特别的设备处理. 它不需要, 因为 scull 设备设计为全局的和永久的. 特别地, 没有如 “在第一次打开时初始化设备” 等动作, 因为我们不为 scull 保持打开计数.

唯一在设备上的真实操作是当设备为写而打开时将它截取为长度为 0. 这样做是因为, 在设计上, 用一个短的文件覆盖一个 scull 设备导致一个短的设备数据区. 这类似于为写而打开一个常规文件, 将其截短为 0. 如果设备为读而打开, 这个操作什么都不做.

在我们查看其他 scull 特性的代码时将看到一个真实的初始化如何起作用的.

6.5.2 release 方法

release 方法的角色是 open 的反面. 有时你会发现方法的实现称为 device_close, 而不是 device_release. 任一方式, 设备方法应当进行下面的任务:

- 释放 open 分配在 filp->private_data 中的任何东西
- 在最后的 close 关闭设备

scull 的基本形式没有硬件去关闭, 因此需要的代码是最少的:

example

```
1   int scull_release(struct inode *inode, struct file *filp)
2   {
3       return 0;
4   }
```

你可能想知道当一个设备文件关闭次数超过它被打开的次数会发生什么. 毕竟, dup 和 fork 系统调用不调用 open 来创建打开文件的拷贝; 每个拷贝接着在程序终止时被关闭. 例如, 大部分程序不打开它们的 stdin 文件 (或设备), 但是它们都以关闭它结束. 当一个打开的设备文件已经真正被关闭时驱动如何知道?

答案简单: 不是每个 close 系统调用引起调用 release 方法. 只有真正释放设备数据结构的调用会调用这个方法——因此得名. 内核维持一个文件结构被使用多少次的计数. fork 和 dup 都不创建新文件 (只有 open 这样); 它们只递增正存在的结构中的计数. close 系统调用仅在文件结构计数掉到 0 时执行 release 方法, 这在结构被销毁时发生. release 方法和 close 系统调用之间的这种关系保证了你的驱动一次 open 只看到一次 release.

注意, flush 方法在每次应用程序调用 close 时都被调用. 但是, 很少驱动实现 flush, 因为常常在 close 时没有什么要做, 除非调用 release.

如你会想到的, 前面的讨论即便是应用程序没有明显地关闭它打开的文件也适用: 内核在进程 exit 时自动关闭了任何文件, 通过在内部使用 close 系统调用.

6.6 scull 的内存使用

在介绍读写操作前, 我们最好看看如何以及为什么 scull 进行内存分配. " 如何" 是需要全面理解代码, " 为什么" 演示了驱动编写者需要做的选择, 尽管 scull 明确地不是典型设备.

本节只处理 scull 中的内存分配策略, 不展示给你编写真正驱动需要的硬件管理技能.

scull 使用的内存区, 也称为一个设备, 长度可变. 你写的越多, 它增长越多; 通过使用一个短文件覆盖设备来进行修整. scull 驱动引入 2 个核心函数来管理 Linux 内核中的内存. 这些函数, 定义在 <linux/slab.h>, 是:

example

```
1  void *kmalloc(size_t size, int flags);
2  void kfree(void *ptr);
```

对 kmalloc 的调用试图分配 size 字节的内存; 返回值是指向那个内存的指针或者如果分配失败为 NULL. flags 参数用来描述内存应当如何分配; 分配的内存应当用 kfree 来释放. 你应当从不传递任何不是从 kmalloc 获得的东西给 kfree. 但是, 传递一个 NULL 指针给 kfree 是合法的.

kmalloc 不是最有效的分配大内存区的方法, 所以挑选给 scull 的实现不是一个特别巧妙的. 一个巧妙的源码实现可能更难阅读, 而本节的目标是展示读和写, 不是内存管理. 这是为什么代码只是使用 kmalloc 和 kfree 而不依靠整页的分配, 尽管这个方法会更有效.

在 flip 一边, 我们不想限制" 设备" 区的大小, 由于理论上的和实践上的理由. 理论上, 给在被管理的数据项施加武断的限制总是个坏想法. 实践上, scull 可用来暂时地吃光你系统中的内存, 以便运行在低内存条件下的测试. 运行这样的测试可能会帮助你理解系统的内部. 你可以使用命令 cp /dev/zero /dev/scull0 来用 scull 吃掉所有的真实 RAM, 并且你可以使用 dd 工具来选择贝多少数据给 scull 设备.

在 scull, 每个设备是一个指针链表, 每个都指向一个 scull_dev 结构. 每个这样的结构, 缺省地, 指向最多 4 兆字节, 通过一个中间指针数组. 发行代码使用一个 1000 个指针的数组指向每个 4000 字节的区域. 我们称每个内存区域为一个量子, 数组 (或者它的长度) 为一个量子集. 一个 scull 设备和它的内存区如图所示.

选定的数字是这样, 在 scull 中写单个一个字节消耗 8000 或 12,000 KB 内存: 4000 是量子, 4000 或者 8000 是量子集 (根据指针在目标平台上是用 32 位还是 64 位表示). 相反, 如果你写入大量数据, 链表的开销不是太坏. 每 4 MB 数据只有一个链表元素, 设备的最大尺寸受限于计算机的内存大小.

为量子和量子集选择合适的值是一个策略问题, 而不是机制, 并且优化的值依赖于设备如何使用. 因此, scull 驱动不应当强制给量子和量子集使用任何特别的值. 在 scull 中, 用户可以掌管改变这些值, 有几个途径: 编译时间通过改变 scull.h 中的宏 SCULL_QUANTUM 和 SCULL_QSET, 在模块加载时设定整数值 scull_quantum 和 scull_qset, 或者使用 ioctl 在运行时改变当前值和缺省值.

使用宏定义和一个整数值来进行编译时和加载时配置, 是对于如何选择主编号的回忆. 我们在驱动中任何与策略相关或专断的值上运用这个技术.

余下的唯一问题是如果选择缺省值. 在这个特殊情况下, 问题是找到最好的平衡, 由填充了一半的量子和量子集导致内存浪费, 如果量子和量子集小的情况下分配释放和指针连接引起开销. 另外, kmalloc 的内部设计应当考虑进去. 缺省值的选择来自假设测试时可能有大量数据写进 scull, 尽管设备的正常使用最可能只传送几 KB 数据.

我们已经见过内部代表我们设备的 scull_dev 结构. 结构的 quantum 和 qset 分别代表设备的量子和量子集大小. 实际数据, 但是, 是由一个不同的结构跟踪, 我们称为 struct scull_qset:

example

```
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
```

下一个代码片段展示了实际中 struct scull_dev 和 struct scull_qset 是如何被用来持有数据的. sucll_trim 函数负责释放整个数据区, 由 scull_open 在文件为写而打开时调用. 它简单地遍历列表并且释放它发现的任何量子和量子集.

example

```
int scull_trim(struct scull_dev *dev) {
    struct scull_qset *next, *dptr;
    int qset = dev->qset;
    int i;
    for (dptr = dev->data; dptr; dptr = next) {
        if (dptr->dta) {
            for (i = 0; i < qset; i++) {
                kfree(dptr->data[i]);
            }
            kfree(dptr->data);
            dptr->data = NULL;
        }

        next = dptr->next;
        kfree(dptr);
    }

    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    dev->data = NULL;
    return 0;
}
```

scull_trim 也用在模块清理函数中, 来归还 scull 使用的内存给系统.

6.7 读和写

读和写方法都进行类似的任务, 就是, 从和到应用程序代码拷贝数据. 因此, 它们的原型相当相似, 可以同时介绍它们:

example

```
1 ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);
2 ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t
      *offp);
```

对于 2 个方法, filp 是文件指针, count 是请求的传输数据大小. buff 参数指向持有被写入数据的缓存, 或者放入新数据的空缓存. 最后, offp 是一个指针指向一个 “long offset type” 对象, 它指出用户正在存取的文件位置. 返回值是一个 “signed size type”; 它的使用在后面讨论.

让我们重复一下, read 和 write 方法的 buff 参数是用户空间指针. 因此, 它不能被内核代码直接解引用. 这个限制有几个理由:

- 依赖于你的驱动运行的体系, 以及内核被如何配置的, 用户空间指针当运行于内核模式可能根本是无效的. 可能没有那个地址的映射, 或者它可能指向一些其他的随机数据.
- 就算这个指针在内核空间是同样的东西, 用户空间内存是分页的, 在做系统调用时这个内存可能没有在 RAM 中. 试图直接引用用户空间内存可能产生一个页面错, 这是内核代码不允许做的事情. 结果可能是一个"oops", 导致进行系统调用的进程死亡.
- 置疑中的指针由一个用户程序提供, 它可能是错误的或者恶意的. 如果你的驱动盲目地解引用一个用户提供的指针, 它提供了一个打开的门路使用户空间程序存取或覆盖系统任何地方的内存. 如果你不想负责你的用户的系统的安全危险, 你就不能直接解引用用户空间指针.

显然, 你的驱动必须能够存取用户空间缓存以完成它的工作.

scull 中的读写代码需要拷贝一整段数据到或者从用户地址空间. 这个能力由下列内核函数提供, 它们拷贝一个任意的字节数组, 并且位于大部分读写实现的核心中.

example

```
1 unsigned long copy_to_user(void __user *to,const void *from,unsigned long count
      );
2 unsigned long copy_from_user(void *to,const void __user *from,unsigned long
      count);
```

尽管这些函数表现象正常的 memcpy 函数, 必须加一点小心在从内核代码中存取用户空间. 寻址的用户也当前可能不在内存, 虚拟内存子系统会使进程睡眠在这个页被传送到位时. 例如, 这发生在必须从交换空间获取页的时候. 对于驱动编写者来说, 最终结果是任何存取用户空间的函数必须是可重入的, 必须能够和其他驱动函数并行执行, 并且, 特别的, 必须在一个它能够合法地睡眠的位置.

这 2 个函数的角色不限于拷贝数据到和从用户空间: 它们还检查用户空间指针是否有效. 如果指针无效, 不进行拷贝; 如果在拷贝中遇到一个无效地址, 另一方面, 只拷贝部分数据. 在 2 种情况下, 返回值是还要拷贝的数据量. scull 代码查看这个错误返回, 并且如果它不是 0 就返回 -EFAULT 给用户.

用户空间存取和无效用户空间指针的主题有些高级. 然而, 值得注意的是如果你不需要检查用户空间指针,

你可以调用 `__copy_to_user` 和 `__copy_from_user` 来代替. 这是有用处的, 例如, 如果你知道你已经检查了这些参数. 但是, 要小心; 事实上, 如果你不检查你传递给这些函数的用户空间指针, 那么你可能造成内核崩溃和/或安全漏洞.

至于实际的设备方法, read 方法的任务是从设备拷贝数据到用户空间 (使用`copy_to_user`), 而 write 方法必须从用户空间拷贝数据到设备 (使用 `copy_from_user`). 每个 read 或 write 系统调用请求一个特定数目字节的传送, 但是驱动可自由传送较少数据 – 对读和写这确切的规则稍微不同, 在本章后面描述.

不管这些方法传送多少数据, 它们通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置. 内核接着在适当时候传播文件位置的改变到文件结构. pread 和 pwrite 系统调用有不同的语义; 它们从一个给定的文件偏移操作, 并且不改变其他的系统调用看到的文件位置. 这些调用传递一个指向用户提供的位置的指针, 并且放弃你的驱动所做的改变.

read 和 write 方法都在发生错误时返回一个负值. 相反, 大于或等于 0 的返回值告知调用程序有多少字节已经成功传送. 如果一些数据成功传送接着发生错误, 返回值必须是成功传送的字节数, 错误不报告直到函数下一次调用. 实现这个传统, 当然, 要求你的驱动记住错误已经发生, 以便它们可以在以后返回错误状态.

尽管内核函数返回一个负数指示一个错误, 这个数的值指出所发生的错误类型, 用户空间运行的程序常常看到 -1 作为错误返回值. 它们需要存取 errno 变量来找出发生了什么. 用户空间的行为由 POSIX 标准来规定, 但是这个标准没有规定内核内部如何操作.

6.7.1 read 方法

read 的返回值由调用的应用程序解释:

- 如果这个值等于传递给 read 系统调用的 count 参数, 请求的字节数已经被传送. 这是最好的情况.
- 如果是正数, 但是小于 count, 只有部分数据被传送. 这可能由于几个原因, 依赖于设备. 常常, 应用程序重新试着读取. 例如, 如果你使用 fread 函数来读取, 库函数重新发出系统调用直到请求的数据传送完成.
- 如果值为 0, 到达了文件末尾 (没有读取数据).
- 一个负值表示有一个错误. 这个值指出了什么错误, 根据 <linux/errno.h>. 出错的典型返回值包括 -EINTR(被打断的系统调用) 或者 -EFAULT(坏地址).

前面列表中漏掉的是这种情况" 没有数据, 但是可能后来到达". 在这种情况下, read 系统调用应当阻塞.

scull 代码利用了这些规则. 特别地, 它利用了部分读规则. 每个 scull_read 调用只处理单个数据量子, 不实现一个循环来收集所有的数据; 这使得代码更短更易读. 如果读程序确实需要更多数据, 它重新调用. 如果标准 I/O 库 (例如, fread) 用来读取设备, 应用程序甚至不会注意到数据传送的量子化.

如果当前读取位置大于设备大小, scull 的 read 方法返回 0 来表示没有可用的数据 (换句话说, 我们在文件尾). 这个情况发生在如果进程 A 在读设备, 同时进程 B 打开它写, 这样将设备截短为 0. 进程 A 突然发现自己过了文件尾, 下一个读调用返回 0.

这是 read 的代码 (忽略对 down_interruptible 的调用并且现在为 up;):

example

```
1 ssize_t scull_read(struct file *filp, char __user *buff, size_t count,
2                    loff_t *f_pos)
3 {
4     struct scull_dev *dev = filp->private_data;
5     struct scull_block *pblock = NULL;
6     loff_t retval = -ENOMEM;
7     loff_t tblock = 0, toffset = 0;
```

```
8     struct list_head *plist = NULL;
9
10    pr_debug("%s() is invoked\n", __FUNCTION__);
11
12    tblock = *f_pos / SCULL_BLOCK_SIZE;
13    toffset = *f_pos % SCULL_BLOCK_SIZE;
14
15    if (mutex_lock_interruptible(&dev->mutex))
16        return -ERESTARTSYS;
17
18    if (tblock + 1 > dev->block_counter)
19    {
20        retval = 0;
21        goto end_of_file;
22    }
23
24    plist = &dev->block_list;
25    for (int i = 0; i < tblock + 1; ++i)
26    {
27        plist = plist->next;
28    }
29
30    pblock = list_entry(plist, struct scull_block, block_list);
31    if (toffset >= pblock->offset)
32    {
33        retval = 0;
34        goto end_of_file;
35    }
36
37    if (count > pblock->offset)
38        count = pblock->offset;
39
40    if (copy_to_user(buff, pblock->data, count))
41    {
42        retval = -EFAULT;
43        goto cpy_user_error;
44    }
45
46    retval = count;
47    *f_pos += count;
48
```

```
49 end_of_file:
50 cpy_user_error:
51     pr_debug("RD pos = %lld, block = %lld, offset = %lld, read %lu bytes\n",
52             *f_pos, tblock, toffset, count);
53
54     mutex_unlock(&dev->mutex);
55     return retval;
56 }
```

6.7.2 write 方法

write, 像 read, 可以传送少于要求的数据, 根据返回值的下列规则:

- 如果值等于 count, 要求的字节数已被传送.
- 如果正值, 但是小于 count, 只有部分数据被传送. 程序最可能重试写入剩下的数据.
- 如果值为 0, 什么没有写. 这个结果不是一个错误, 没有理由返回一个错误码. 再一次, 标准库重试写调用.
- 一个负值表示发生一个错误; 如同对于读, 有效的错误值是定义于 <linux/errno.h> 中.

不幸的是, 仍然可能有发出错误消息的不当行为程序, 它在进行了部分传送时终止. 这是因为一些程序员习惯看写调用要么完全失败要么完全成功, 这实际上是大部分时间的情况, 应当也被设备支持. scull 实现的这个限制可以修改, 但是我们不想使代码不必要地复杂.

write 的 scull 代码一次处理单个量子, 如 read 方法做的:

example

```
1  ssize_t scull_write(struct file *filp, const char __user *buff, size_t count,
2                      loff_t *f_pos)
3  {
4      struct scull_dev *dev = filp->private_data;
5      struct scull_block *pblock = NULL;
6      loff_t retval = -ENOMEM;
7      loff_t tblock = 0, toffset = 0;
8
9      pr_debug("%s() is invoked\n", __FUNCTION__);
10
11     tblock = *f_pos / SCULL_BLOCK_SIZE;
12     toffset = *f_pos % SCULL_BLOCK_SIZE;
13
14     if (mutex_lock_interruptible(&dev->mutex))
15         return -ERESTARTSYS;
16
17     /*
```

```
18      * For simplicity, we write one block each write request.
19      */
20     while (tblock + 1 > dev->block_counter)
21     {
22         if (!(pblock = kmalloc(sizeof(struct scull_block), GFP_KERNEL)))
23             goto malloc_error;
24         memset(pblock, 0, sizeof(struct scull_block));
25         INIT_LIST_HEAD(&pblock->block_list);
26         list_add_tail(&pblock->block_list, &dev->block_list);
27         dev->block_counter++;
28     }
29     pblock = list_last_entry(&dev->block_list, struct scull_block, block_list);
30
31     if (count > SCULL_BLOCK_SIZE - toffset)
32         count = SCULL_BLOCK_SIZE - toffset;
33
34     if (copy_from_user(pblock->data + toffset, buff, count))
35     {
36         retval = -EFAULT;
37         goto cpy_user_error;
38     }
39
40     retval = count;
41     pblock->offset += count;
42     *f_pos += count;
43
44 malloc_error:
45 cpy_user_error:
46     pr_debug("WR pos = %lld, block = %lld, offset = %lld, write %lu bytes\n",
47             *f_pos, tblock, toffset, count);
48
49     mutex_unlock(&dev->mutex);
50     return retval;
51 }
```

6.8 使用新设备

一旦你装备好刚刚描述的 4 个方法, 驱动可以编译并测试了; 它保留了你写给它的任何数据, 直到你用新数据覆盖它. 这个设备表现如一个数据缓存器, 它的长度仅仅受限于可用的真实 RAM 的数量. 你可试着使用 cp, dd, 以及输入/输出重定向来测试这个驱动.

free 命令可用来看空闲内存的数量如何缩短和扩张的, 依据有多少数据写入 scull.

为对一次读写一个量子有更多信心, 你可增加一个 printk 在驱动的适当位置, 并且观察当应用程序读写大块数据中发生了什么. 可选地, 使用 strace 工具来监视程序发出的系统调用以及它们的返回值. 跟踪一个 cp 或者一个 ls -l > /dev/scull0 展示了量子化的读和写.

第七章 块设备驱动程序

第八章 I^2C 客户端驱动程序

由飞利浦（现为恩智浦）发明的 I^2C 总线是双线制：由串行数据（SDA）、串行时钟（SCL）构成的异步串行总线。它是多主总线，但多主模式未广泛使用。SDA 和 SCL 都是漏极开路/集电极开路，这意味着它们都可以使输出驱动为低电平，但是如果没有上拉电阻，则都不能使输出驱动为高电平。SCL 由主设备生成，以便在总线上同步数据（由 SDA 传送）传送。从机和主机都可以发送数据（当然不是同时），因此 SDA 是双向线路。这就是说 SCL 信号也是双向的，因为从机可以通过保持 SCL 线低电平来延长时钟。总线由主机控制，在这里的例子中它是 SoC 的一部分。该总线经常在嵌入式系统中，用于连接串行 EEPROM、RTC 芯片、GPIO 扩展器、温度传感器等，如图8.1所示。

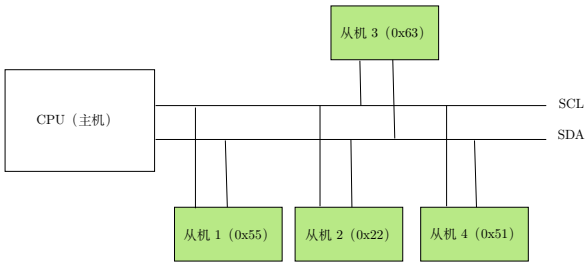

图 8.1: I^2C 总线和设备

时钟频率为 10 kHz 100 kHz,400 kHz 2 MHz 不等。本教程不涉及总线规范或总线驱动程序。然而，总线驱动程序应该管理总线并符合总线规范。i.MX6 芯片的总线驱动程序示例可在内核源代码 drivers/i2c/busses/i2c-imx.c 中找到。

本章集中介绍客户端驱动程序，以便处理该总线上的从设备。本章涉及以下主题。

- I^2C 客户驱动程序架构。
- 设备访问，从设备中读/写设备数据。
- 在 DT 中声明客户端设备。

8.1 驱动程序架构

当为其编写驱动程序的设备位于物理总线（被称作总线控制器）上时，它一定依赖总线的驱动程序，也就是控制器驱动程序，它负责在设备之间共享总线访问。控制器驱动程序在设备和总线之间提供抽象层。例如，当在 I^2C 或 USB 总线上执行事务（读取或写入）时，I^2C/USB 总线控制器将在后台透明地处理该事务。每个总线控制器驱动程序都提供一组函数，以简化位于该总线上设备驱动程序的开发。这适用于每个物理总线（I^2C、SPI、USB、PCI、SDIO 等）。I^2C 驱动程序在内核中表示为`struct i2c_driver`的实例。I^2C 客户端（代表设备本身）由`struct i2c_client`结构表示。

8.1.1 i2c_driver 结构

I^2C 驱动程序在内核中声明为`struct i2c_driver`实例，它看起来像下面这样:

i2c_driver

```
1 struct i2c_driver {
2     /* 标准驱动模型接口 */
```

```
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);
    /* 与枚举无关的驱动类型接口 */
    void (*shutdown)(struct i2c_client *);
    struct device_driver driver;
    const struct i2c_device_id *id_table;
};
```

`struct i2c_driver` 结构包含并描述通用访问例程，这些例程是处理声明驱动程序的设备所必需的，而 `struct i2c_cl` 包含设备特有的信息，如其地址。`struct i2c_client` 结构表示和描述 I^2C 设备。本章后面的部分将介绍如何填充这些结构。

1. probe 函数

probe 函数是 `struct i2c_driver` 结构的一部分，在 I^2C 器件实例化后随时执行。它负责以下任务。

- 检查设备是否是所期望的。
- 使用 `i2c_check_functionality` 函数检查 SoC 的 I^2C 总线控制器是否支持设备所需的功能。
- 初始化设备。
- 设置设备特定的数据。
- 注册合适的内核框架。

probe 函数的原型如下：

probe 函数的原型

```
static int foo_probe(struct i2c_client *client, const struct i2c_device_id *id)
```

各参数说明如下。

- `struct i2c_client` 指针：代表 I^2C 设备本身。该结构继承自 device 结构，由内核提供给 probe 函数。客户端结构在 `include/linux/i2c.h` 中定义。其定义如下：

i2c_client

```
struct i2c_client {
    unsigned short flags;
    /// 芯片地址：7位。地址被存储在addr的低7位。
    unsigned short addr;
    char name[I2C_NAME_SIZE];
    /// 适配器
    struct i2c_adapter *adapter;
    /// 设备结构
    struct device dev;
```

```
    /// 设备发出的中断请求
    intirq;
    struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE)
    /// 从设备的回调函数
    i2c_slave_cb_t slave_cb;
#endif
};
```

- 所有字段由内核基于注册客户端时所提供的参数进行填充。稍后会看到如何将设备注册到内核。
- `struct i2c_device_id` 指针：指向与正在探测的设备相匹配的 I^2C 设备 ID 项。

I^2C 内核使用户能够存储指向其所选任何数据结构的指针，把它作为设备特定数据。要存储或检索数据，请使用 I^2C 内核提供的以下函数：

I^2C 内核提供的函数

```
/// 设置数据
void i2c_set_clientdata(struct i2c_client *client, void *data);
/// 获取数据
void *i2c_get_clientdata(const struct i2c_client *client);
```

这些函数在内部调用 `dev_set_drvdata` 和 `dev_get_drvdata` 来更新或获取 `struct i2c_client` 结构中 `struct dev` 结构的 `void * driver_data` 字段的值。

8.2 总结

学会如何处理 I^2C 设备驱动程序之后，可以付诸实践了：在市场上选购任一款 I^2C 设备，编写相应的驱动程序，并支持 DT。

本章讨论了内核中的 I^2C 内核和相关的 API，包括设备树支持，介绍了与 I^2C 设备通信的必要技能。读者现在应该能够编写出高效的 probe 函数，并注册到 I^2C 内核中了。第 8 章将使用在这里学到的技能开发 SPI 设备驱动程序。

第二部分

Linux 移植

第九章 移植到 SD 卡

本次实验宿主机使用全新的 Ubuntu20.04 环境，并在完成实验后用户目录下有以下结构的文件夹。
首先需要创建架构。

文件结构

```
v3s-workspace
- linux
- partitions
-- boot
--- boot.scr
--- sun8i-v3s-licheepi-zero-dock.dtb
--- zImage
-- root
- u-boot
- modules
- buildroot
- boot.cmd
```

然后下载所有本次实验会使用到的代码和依赖。

安装依赖

```
# 安装依赖
sudo apt-get install flex bison gcc make gcc-arm-linux-gnueabihf libncurses-dev
     swig python-dev device-tree-compiler python3-setuptools python3-dev libssl
    -dev u-boot-tools g++ patch

# 下载Mainline Linux,你可以在https://www.kernel.org寻找最新的LTS版本
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.19.tar.xz
# extract是zsh提供的自动解压指令，可以替换成tar的对应格式解压指令
extract linux-6.1.19.tar.xz
mv linux-6.1.19 linux

# 下载U-Boot，我们这里没有使用LTS版本，你可以进入cd进去后切换到LTS分支
git clone git://git.denx.de/u-boot.git

# 下载Buildroot，你可以在https://buildroot.org/downloads寻找最新的版本
wget https://buildroot.org/downloads/buildroot-2023.02.tar.xz
extract buildroot-2023.02.tar.xz
```

```
16 mv buildroot-2023.02 buildroot
```

每一章节结束后请返回 `v3s-workspace` 目录。

9.1 编译 U-Boot

编译 *U-Boot*

```
1 cd u-boot
2 # 使用荔枝派Nano的默认配置
3 make CROSS_COMPILE=arm-linux-gnueabihf- LicheePi_Zero_defconfig
4 # 编译
5 make CROSS_COMPILE=arm-linux-gnueabihf-
6 # 拷贝U-Boot镜像
7 cp u-boot-sunxi-with-spl.bin ../partitions
```

接下来我们需要准备 U-Boot 启动所需的配置文件，将以下内容写入 boot.cmd。

s

```
1 etenv bootargs console=tty0 console=ttyS0,115200 panic=5 rootwait root=/dev/
      mmcblk0p2 rw
2 load mmc 0:1 0x43000000 sun8i-v3s-licheepi-zero-dock.dtb
3 load mmc 0:1 0x42000000 zImage
4 bootz 0x42000000 - 0x43000000
```

接着编译配置文件。

编译配置文件

```
1 mkimage -C none -A arm -T script -d boot.cmd ./partitions/boot/boot.scr
```

9.2 编译 Linux 内核

配置编译

```
cd linux
# 使用linux-sunxi项目的默认配置，该项目主要包含全志各芯片的硬件支持文档和手册
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- sunxi_defconfig
# 进入内核配置菜单
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

在内核配置菜单中，将 `Networking support > Wireless` 中的选项全部选中。

我们需要修改 `arch/arm/boot/dts/sun8i-v3s-licheepi-zero.dts` 以启用以太网和 USB 支持。

设备树文件

```
/dts-v1/;
#include "sun8i-v3s.dtsi"
#include "sunxi-common-regulators.dtsi"

/ {
    model = "Lichee Pi Zero";
    compatible = "licheepi,licheepi-zero", "allwinner,sun8i-v3s";

    aliases {
        serial0 = &uart0;
        ethernet0 = &emac; /* 添加这一行 */
    };

    chosen {
        stdout-path = "serial0:115200n8";
    };

    leds {
        compatible = "gpio-leds";

        blue_led {
            label = "licheepi:blue:usr";
            gpios = <&pio 6 1 GPIO_ACTIVE_LOW>; /* PG1 */
        };

```

```
26          green_led {
27              label = "licheepi:green:usr";
28              gpios = <&pio 6 0 GPIO_ACTIVE_LOW>; /* PG0 */
29              default-state = "on";
30          };
31
32          red_led {
33              label = "licheepi:red:usr";
34              gpios = <&pio 6 2 GPIO_ACTIVE_LOW>; /* PG2 */
35          };
36      };
37
38    /* 添加以下soc部分 */
39      soc {
40          ehci0: usb@01c1a000 {
41              compatible = "allwinner,sun8i-v3s-ehci", "generic-ehci";
42              reg = <0x01c1a000 0x100>;
43              interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
44              clocks = <&ccu CLK_BUS_EHCI0>, <&ccu CLK_BUS_OHCI0>;
45              resets = <&ccu RST_BUS_EHCI0>, <&ccu RST_BUS_OHCI0>;
46              status = "okay";
47          };
48
49          ohci0: usb@01c1a400 {
50              compatible = "allwinner,sun8i-v3s-ohci", "generic-ohci";
51              reg = <0x01c1a400 0x100>;
52              interrupts = <GIC_SPI 73 IRQ_TYPE_LEVEL_HIGH>;
53              clocks = <&ccu CLK_BUS_EHCI0>, <&ccu CLK_BUS_OHCI0>,
54              <&ccu CLK_USB_OHCI0>;
55              resets = <&ccu RST_BUS_EHCI0>, <&ccu RST_BUS_OHCI0>;
56              status = "okay";
57          };
58      };
59  };
60
61  &mmc0 {
62      broken-cd;
63      bus-width = <4>;
64      vmmc-supply = <&reg_vcc3v3>;
65      status = "okay";
66  };
```

```
67
68 &uart0 {
69     pinctrl-0 = <&uart0_pb_pins>;
70     pinctrl-names = "default";
71     status = "okay";
72 };
73
74 &usb_otg {
75     dr_mode = "host";
76     status = "okay";
77 };
78
79 &usbphy {
80     usb0_id_det-gpios = <&pio 5 6 GPIO_ACTIVE_LOW>;
81     status = "okay";
82 };
83
84 /* 添加emac部分 */
85 &emac {
86     allwinner,leds-active-low;
87     status = "okay";
88 };
```

接着执行以下指令。

编译内核

```
1  # 编译内核，-j4的4可以修改为你的CPU核心数
2  ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j4 zImage
3  # 编译DTB文件，本文件用于Kernel识别外设，是 Mainline Kernel不可缺少的部分，-j4的4可
       以修改为你的CPU核心数
4  ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j4 dtbs
5  # 编译Modules，-j4的4可以修改为你的CPU核心数
6  ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j4 modules
7  # 安装模块
8  ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=../modules make
       modules modules_install
9  # 拷贝生成的zImage内核镜像和DTB文件
10 cp arch/arm/boot/zImage ../partitions/boot
11 cp arch/arm/boot/dts/sun8i-v3s-licheepi-zero-dock.dtb ../partitions/boot
```

9.3 编译 Buildroot

我们使用 Buildroot 默认的 busybox 程序和 glibc，如果需要剪裁大小，可以选择其他的 C 支持库。

配置菜单

```
$ cd
$ make menuconfig
```

配置位置	操作	用途
Target options	Target Arch 设置为 ARM (little endian)	设置大小端
Target options	Target Arch Variant 设置为 Cortex-A7	设置 CPU 架构
Toolchain	Kernel Headers 设置为你下载的 LTS 版本内核对应的版本号	匹配内核版本
Target packages >Networking applications	hostapd	开启
Target packages >Networking applications	hostapd >Enable hostap driver	开启
Target packages >Networking applications	hostapd >Enable nl80211 driver	开启
Target packages >Networking applications	hostapd >Enable ACS	开启
Target packages >Networking applications	hostapd >Enable EAP	开启
Target packages >Networking applications	hostapd >Enable WPS	开启
Target packages >Networking applications	openssh	开启
Target packages >Networking applications	openssh >client	开启
Target packages >Networking applications	openssh >server	开启
Target packages >Networking applications	openssh >key utilities	开启
Target packages >Networking applications	openssh >use sandboxing	开启
Target packages >Networking applications	wireless tools	开启
Target packages >Networking applications	wireless tools >Install shared library	开启
Target packages >Networking applications	wpa_supplicant	开启
Target packages >Libraries >Crypto	openssl support	openssl support 的子菜单部件全部选中

执行以下命令

编译跟文件系统 *rootfs*

```
# 编译，Buildroot不支持多线程编译，所以不携带-j
make
# 拷贝解压文件系统和内核模块
cp output/images/rootfs.tar ../partitions/root
cd ../partitions/root
extract rootfs.tar
mv rootfs/* ./
rm -rf rootfs
rm rootfs.tar
cp -R ../../modules/* ./
```

同时修改 partitions/root/etc/fstab 文件中的 ext2 为 ext4。

9.4 写入 SD 卡

我的 SD 卡路径是/dev/sdb，可以通过 sudo fdisk -l 查看 SD 卡的路径。

将 *UBoot* 写入 *SD* 卡

```
cd partitions
# 清空分区表
sudo dd if=/dev/zero of=/dev/sdb bs=1M count=1
# 写入U-Boot
sudo dd if=u-boot-sunxi-with-spl.bin of=/dev/sdb bs=1024 seek=8
```

将分区表写入 *SD* 卡

```
# 写入分区表，请复制除了本行内的内容并执行
sudo blockdev --rereadpt /dev/sdb
cat <<EOT | sudo sfdisk /dev/sdb
1M,16M,c
,,L
EOT
```

将内核和根文件系统写入 *SD* 卡

```
# 格式化
sudo mkfs.vfat /dev/sdb1
sudo mkfs.ext4 /dev/sdb2
# 拷贝boot进入第一个分区
sudo mount /dev/sdb1 /mnt
sudo cp -R boot/* /mnt
sync
sudo umount /mnt
# 拷贝rootfs进入第二个分区
sudo mount /dev/sdb2 /mnt
sudo cp -R root/* /mnt
sync
sudo umount /mnt
```

9.5 将开发板作为 ssh 服务器来使用

连接串口，波特率设置为 115200 。
使用 putty 软件连接嵌入式 Linux 开发板。

启动 sshd 服务

```
vi /etc/ssh/sshd_config
# 将文件中的PermitRootLogin设置为yes
# PermitRootLogin yes
# 保存退出

# 使用passwd设置密码
passwd
# 启动sshd服务
/usr/sbin/sshd
```

这里建议重启一下板子，否则 /etc/ssh/sshd_config 不一定能够生效，我实测时就是需要重启才可以。

启动以太网服务

```
ifconfig eth0 up
udhcpc
```

9.6 安装无线网卡驱动

安装无线网卡驱动

```
cd v3s-workspace
git clone https://github.com/al177/esp8089.git
cd esp8089
ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -C ../linux/ M=${PWD} modules

# 172.17.0.58替换为你的电脑看到的Linux开发板的ip地址。
scp esp8089.ko root@172.17.0.58:/root/
ssh root@172.17.0.58
# 进入嵌入式Linux开发板之后的操作
# 加载wifi驱动
atguigu-pi>insmod esp8089.ko
atguigu-pi>ifconfig wlan0 up
atguigu-pi>wpa_passphrase your_SSID your_passphrase > your_SSID.conf
atguigu-pi>wpa_supplicant -B -i wlan0 -c your_SSID.conf
```

第三部分

驱动程序举例

第十章 V3S 按键驱动

我们的开发板上面，有 5 个按键本身不是通过 gpio 连接到 soc 上面的。它是通过 adc 的方法，连接到主芯片的。这个时候，不同的按键被按下的时候，就会生成不同的电压或者电流，那么完全可以根据对应的电信号，推算出当前是哪一个按键被按下去了。

简单看一下电路之后，下面就是去找设备树，对应的信号是什么、在哪里。

10.1 查找设备树

在 sun8i-v3s-licheepi-zero-dock.dts 文件当中，我们发现了这样的内容，

设备树

```
&lradc {
    vref-supply = <&reg_vcc3v0>;
    status = "okay";

    button@200 {
        label = "Volume Up";
        linux,code = <KEY_VOLUMEUP>;
        channel = <0>;
        voltage = <200000>;
    };

    button@400 {
        label = "Volume Down";
        linux,code = <KEY_VOLUMEDOWN>;
        channel = <0>;
        voltage = <400000>;
    };

    button@600 {
        label = "Select";
        linux,code = <KEY_SELECT>;
        channel = <0>;
        voltage = <600000>;
    };

    button@800 {
        label = "Start";
        linux,code = <KEY_OK>;
        channel = <0>;
```

```
30          voltage = <800000>;
31      };
32  };
```

很明显，每一个 button 都是和电路中的按键是一一对应的，这个没有问题。那么，我们不禁还有一个疑问，既然是 ad 转换得到的结果，那么肯定要知道 ad 相关的设备配置是恶还那么。仔细找了一下，可以在 sun8i-v3s.dtsi 文件发现这样的内容，

lradc 的配置

```
1  lradc: lradc@01c22800 {
2      compatible = "allwinner,sun4i-a10-lradc-keys";
3      reg = <0x01c22800 0x400>;
4      interrupts = <GIC_SPI 30 IRQ_TYPE_LEVEL_HIGH>;
5      status = "disabled";
6  };
```

看到这里，大家应该放心了，确实是有这么一个 ad 的驱动。兼容的设备是 sun4i-a10-lradc-keys，寄存器地址空间是 0x01c22800，长度是 0x400，中断是 GIC_SPI 类型，状态关闭。有了设备树，还有了兼容设备号，接下来的一步就是根据设备号 sun4i-a10-lradc-keys 找到对应的驱动文件。

10.2 查找驱动代码，准备测试程序

通过工具查找一下，不难发现，文件在这，即 sun4i-lradc-keys.c，

ad 驱动代码

```
1  static const struct of_device_id sun4i_lradc_of_match[] = {
2      { .compatible = "allwinner,sun4i-a10-lradc-keys", },
3      { /* sentinel */ }
4  };
5  MODULE_DEVICE_TABLE(of, sun4i_lradc_of_match);
6
7  static struct platform_driver sun4i_lradc_driver = {
8      .driver = {
9          .name   = "sun4i-a10-lradc-keys",
10         .of_match_table = of_match_ptr(sun4i_lradc_of_match),
11     },
```

```
12      .probe  = sun4i_lradc_probe,
13  };
14
15  module_platform_driver(sun4i_lradc_driver);
16
17  MODULE_DESCRIPTION("Allwinner sun4i low res adc attached tablet keys driver");
18  MODULE_AUTHOR("Hans de Goede <hdegoede@redhat.com>");
19  MODULE_LICENSE("GPL");
```

一般来说，如果按键 ok 的话，会在设备启动的时候生成个/dev/input/event0 节点，此时，如果编写一个应用程序，读写这些节点，就完全可以获取相关的按键信息。所以，我们还得准备一个 input.c 的读写程序，

input.c

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #include <errno.h>
9  #include <linux/input.h>
10 #include <linux/input-event-codes.h>
11
12 const char * path = "/dev/input/event0";
13
14 int main(char argc,char *argv[])
15 {
16     int ret;
17
18     int fd;
19     struct input_event event;
20
21     fd = open(path,O_RDONLY);
22     if(fd < 0)
23     {
24         perror(path);
25         exit(-1);
26     }
```

```
27
28      while(1)
29      {
30          ret = read(fd, &event, sizeof(struct input_event));
31          if(ret == sizeof(struct input_event))
32          {
33              if(event.type != EV_SYN)
34              {
35                  printf("Event: time %ld.%ld,", event.time.tv_sec, event.time.
                        tv_usec);
36                  printf("type:%d,code:%d,value:%d\n", event.type,event.code, event
                        .value);
37              }
38          }
39      }
40
41      close(fd);
42
43      return 0;
44  }
```

准备好了程序之后，下面就是交叉编译，下载到开发板上面。但是实际运行的时候，发现按键被按下的时候，有三个按键的数值居然是一样的，都是 352，另外一个是 114。这就非常蹊跷了。

10.3 解决问题

查看 sun4i-lradc-keys.c，惊讶地发现电压判断标准是根据 sun8i-v3s-licheepi-zero-dock.dts 中的 voltage 来验证的，这并不符合实际的情况。我们通过 printk&dmesg 打印，也验证了这一想法，所以如果需要得到正确的按键数值，只需要修正一下 sun4i-lradc-keys.c 中的判断逻辑就可以了，修改方法如下，具体的标定数值可以做实验来解决，

修改程序

```
1   #if 0
2           voltage = val * lradc->vref / 63;
3
4           for (i = 0; i < lradc->chan0_map_count; i++) {
5               diff = abs(lradc->chan0_map[i].voltage - voltage);
6               if (diff < closest) {
7                   closest = diff;
```

```
                keycode = lradc->chan0_map[i].keycode;
            }
        }
#else
        printk("val = %d\n", val);
        if(val >=9 && val <= 13)
            keycode = lradc->chan0_map[0].keycode;
        else if(val >=24 && val <= 29)
            keycode = lradc->chan0_map[1].keycode;
        else if(val >= 35 && val <= 40)
            keycode = lradc->chan0_map[2].keycode;
        else
            keycode = lradc->chan0_map[3].keycode;
#endif
```

经过这一次修改，我们重新编译 kernel 内核，烧入 zImage，启动后输入 key 程序，这样就得到了我们想要的最终结果，即稳定地输出按键值。

第十一章 GPIO 驱动示例-1

使用 spi 接口当成 gpio 口。

有一个 spi 接口，之前主要是用作 norflash 访问使用的。现在因为所有系统都保存在 sd 卡里面，因此完全可以用这个当成 gpio 使用。

修改 sun8i-v3s.dtsi 文件

首先注释掉之前 spi0_pins 这个部分，

修改设备树文件

```
/*spi0_pins: spi0 {
    pins = "PC0", "PC1", "PC2", "PC3";
    function = "spi0";
};*/
```

接着注释掉 spi0，

修改设备树文件

```
/*spi0: spi@1c68000 {
    compatible = "allwinner,sun8i-h3-spi";
    reg = <0x01c68000 0x1000>;
    interrupts = <GIC_SPI 65 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&ccu CLK_BUS_SPI0>, <&ccu CLK_SPI0>;
    clock-names = "ahb", "mod";
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins>;
    resets = <&ccu RST_BUS_SPI0>;
    status = "disabled";
    #address-cells = <1>;
    #size-cells = <0>;
};*/
```

注释掉这两部分呢，重新编译成 sun8i-v3s-licheepi-zero-dock.dtb 文件就可以了。细心的同学也许会看到 sun8i-v3s-licheepi-zero.dts 和 sun8i-v3s-licheepi-zero-dock.dts 这两个文件中均有 leds 的配置，是不是 status 设置为 okay 就好了？要注意它们的状态都是写死的，后期不能通过命令和配置的方法来解决，虽然启动后也可以在/sys/kernel/debug/gpio 下面看到映射关系，这个需要注意下。

重启开发板

重启开发板之后，首先需要查看一下端口使用情况，没有 debug 信息，先要 mount debugfs 系统，

```shell
mount -t debugfs debugfs /sys/kernel/debug
# 加载好了之后，就可以看看端口的使用情况了，
cat /sys/kernel/debug/gpio
```

创建通道，开始设备外设

看过上面一篇文章的同学，对于/sys/class/pwm 里面的 export 不会陌生。但是 pwm 只有两个，分别是 0 和 1，gpio 这么多，我们怎么把这些 pin 和通道 bind 在一起呢？其实这里面是有规律的。首先我们找到一个 pin，但不知道它的序号是多少，那可以先找到名称，比如 SPI_CS，接着看`Allwinner_V3s_Datasheet_V1.0.pdf`中的第 54 页，获取引脚名称，

找到了这个信号叫 PC2，下面就好办了。所有的端口一般都是 $channel = 32 \times x + y$ 来实现的。PA、PB、PC...，这些代表 x，分别是 0、1、2...。而 PC2 中的 2 就代表 y，如果是 PB9，那么 y 就是 9。所以对于 PC2 来说，$channel = 32 \times 2 + 2$，也就是 66，就是这么简单。那么，刚才说的 PB9 呢，它的 $channel = 32 \times 1 + 9$，应该就是 41。

说了这么多，下面开始实验，

```shell
echo 66 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio66/direction
echo 1 > /sys/class/gpio/gpio66/value
echo 0 > /sys/class/gpio/gpio66/value
```

四条命令依次解释下，第一条创建 channel 66。第二条呢，设定 chanel 66 的方向为输出。第三条，设置高电平，与此相对的，第四条就是设置低电平。

为了验证设置的电平是不是正确，一个靠谱的办法就是在 spi_cs 处于高电平和低电平的时候都测量下，这样就知道电压有没有设置对了。

第十二章 GPIO 驱动示例-2

gpio_driver.c

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/gpio.h>

/* Meta Information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Atguigu GNU/Linux");
MODULE_DESCRIPTION("A simple gpio driver for setting a LED and reading a button
    ");

/* Variables for device and device class */
static dev_t my_device_nr;
static struct class *my_class;
static struct cdev my_device;

#define DRIVER_NAME "my_gpio_driver"
#define DRIVER_CLASS "MyModuleClass"

/**
 * @brief 将数据从缓冲区中读出
 */
static ssize_t driver_read(struct file *File, char *user_buffer, size_t count,
    loff_t *offs) {
    int to_copy, not_copied, delta;
    char tmp[3] = " \n";

    /* Get amount of data to copy */
    to_copy = min(count, sizeof(tmp));

    /* Read value of button */
    printk("Value of button: %d\n", gpio_get_value(17));
    tmp[0] = gpio_get_value(17) + '0';

    /* Copy data to user */
    not_copied = copy_to_user(user_buffer, &tmp, to_copy);
```

```
37
38      /* Calculate data */
39      delta = to_copy - not_copied;
40
41      return delta;
42  }
43
44  /**
45   * @brief 将数据写入缓冲区
46   */
47  static ssize_t driver_write(struct file *File, const char *user_buffer, size_t
        count, loff_t *offs) {
48      int to_copy, not_copied, delta;
49      char value;
50
51      /* Get amount of data to copy */
52      to_copy = min(count, sizeof(value));
53
54      /* Copy data to user */
55      not_copied = copy_from_user(&value, user_buffer, to_copy);
56
57      /* Setting the LED */
58      switch(value) {
59          case '0':
60              gpio_set_value(4, 0);
61              break;
62          case '1':
63              gpio_set_value(4, 1);
64              break;
65          default:
66              printk("Invalid Input!\n");
67              break;
68      }
69
70      /* Calculate data */
71      delta = to_copy - not_copied;
72
73      return delta;
74  }
75
76  /**
```

```
77  * @brief 当设备文件打开时，调用这个函数
78  */
79 static int driver_open(struct inode *device_file, struct file *instance) {
80     printk("dev_nr - open was called!\n");
81     return 0;
82 }
83 
84 /**
85  * @brief 当设备文件关闭时，调用这个函数
86  */
87 static int driver_close(struct inode *device_file, struct file *instance) {
88     printk("dev_nr - close was called!\n");
89     return 0;
90 }
91 
92 static struct file_operations fops = {
93     .owner = THIS_MODULE,
94     .open = driver_open,
95     .release = driver_close,
96     .read = driver_read,
97     .write = driver_write
98 };
99 
100 /**
101  * @brief 当模块加载进内核时，调用这个函数
102  */
103 static int __init ModuleInit(void) {
104     printk("Hello, Kernel!\n");
105 
106     /* Allocate a device nr */
107     if( alloc_chrdev_region(&my_device_nr, 0, 1, DRIVER_NAME) < 0) {
108         printk("Device Nr. could not be allocated!\n");
109         return -1;
110     }
111     printk("read_write - Device Nr. Major: %d, Minor: %d was registered!\n",
        my_device_nr >> 20, my_device_nr && 0xfffff);
112 
113     /* 创建设备类 */
114     if((my_class = class_create(THIS_MODULE, DRIVER_CLASS)) == NULL) {
115         printk("Device class can not be created!\n");
116         goto ClassError;
```

```
117     }
118
119     /* 创建设备文件 */
120     if(device_create(my_class, NULL, my_device_nr, NULL, DRIVER_NAME) == NULL)
            {
121         printk("Can not create device file!\n");
122         goto FileError;
123     }
124
125     /* 初始化设备文件 */
126     cdev_init(&my_device, &fops);
127
128     /* 将设备注册到内核 */
129     if(cdev_add(&my_device, my_device_nr, 1) == -1) {
130         printk("Registering of device to kernel failed!\n");
131         goto AddError;
132     }
133
134     /* GPIO 4 init */
135     if(gpio_request(4, "rpi-gpio-4")) {
136         printk("Can not allocate GPIO 4\n");
137         goto AddError;
138     }
139
140     /* Set GPIO 4 direction */
141     if(gpio_direction_output(4, 0)) {
142         printk("Can not set GPIO 4 to output!\n");
143         goto Gpio4Error;
144     }
145
146     /* GPIO 17 init */
147     if(gpio_request(17, "rpi-gpio-17")) {
148         printk("Can not allocate GPIO 17\n");
149         goto Gpio4Error;
150     }
151
152     /* Set GPIO 17 direction */
153     if(gpio_direction_input(17)) {
154         printk("Can not set GPIO 17 to input!\n");
155         goto Gpio17Error;
156     }
```

```

    return 0;
Gpio17Error:
    gpio_free(17);
Gpio4Error:
    gpio_free(4);
AddError:
    device_destroy(my_class, my_device_nr);
FileError:
    class_destroy(my_class);
ClassError:
    unregister_chrdev_region(my_device_nr, 1);
    return -1;
}

/**
 * @brief 当模块从内核中移除时，调用这个函数
 */
static void __exit ModuleExit(void) {
    gpio_set_value(4, 0);
    gpio_free(17);
    gpio_free(4);
    cdev_del(&my_device);
    device_destroy(my_class, my_device_nr);
    class_destroy(my_class);
    unregister_chrdev_region(my_device_nr, 1);
    printk("Goodbye, Kernel\n");
}

module_init(ModuleInit);
module_exit(ModuleExit);
```

Makefile 文件如下

Makefile

```
obj-m += gpio_driver.o

all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

第十三章 PWM 驱动示例-1

要使能 pwm 功能，最主要就是修改设备树配置文件，

第一，在 sun8i-v3s.dtsi 中，添加 pwm0 和 pwm1 节点，

pwm 设备树

```
pwm0_pins: pwm0 {
    pins = "PB4";
    function = "pwm0";
};

pwm1_pins: pwm1 {
    pins = "PB5";
    function = "pwm1";
};
```

第二，在 sun8i-v3s-licheepi-zero.dts 中使能 pwm，

使能 *pwm*

```
&pwm {
    pinctrl-names = "default";
    pinctrl-0 = <&pwm0_pins>, <&pwm1_pins>;
    status = "okay";
};
```

修改了这两个文件，下面要做的就是把他们编译成 dtb，下载到 sd 卡里面，等待重启即可。注意，拷贝的 dtb 文件是 un8i-v3s-licheepi-zero-dock.dtb。

如果对驱动代码有兴趣，可以通过 sun8i-v3s-pwm 这个关键字去查找一下。查找后发现，相关的驱动文件名是 drivers/pwm/pwm-sun4i.c。

此外，之前这份驱动已经包含在了 zImage 里面，所以不需要重新编译内核。

前面如果大家做过实验，就可以发现，如果我们没有修改设备树文件，那么发现在/sys/class/pwm 节点下什么也没有。但是修改了之后，就会发现/sys/class/pwm 一下子多了很多的内容，

显示内容

```
# cd /sys/class
# cd pwm/
# ls
```

```
pwmchip0
# cd pwmchip0/
# ls
device   export   npwm     power    subsystem uevent  unexport
# ls -l
total 0
lrwxrwxrwx  1 root   root          0 Jan 1 00:31 device -> ../../../1c21400.pwm
--w-------  1 root   root       4096 Jan 1 00:31 export
-r--r--r--  1 root   root       4096 Jan 1 00:31 npwm
drwxr-xr-x  2 root   root          0 Jan 1 00:31 power
lrwxrwxrwx  1 root   root          0 Jan 1 00:31 subsystem ->
    ../../../../../../class/pwm
-rw-r--r--  1 root   root       4096 Jan 1 00:31 uevent
--w-------  1 root   root       4096 Jan 1 00:31 unexport
```

首先，我们可以通过 export 来使能通道，输入 0 就可以创建通道 0，输入 1 就可以创建通道 1，根据具体情况而定。

举例

```
echo 0 > /sys/class/pwm/pwmchip0/export
# 通道创建好了，就可以进入到通道里面，看看有哪些配置。以通道0为例，
cd pwm0
ls
```

简单来说，可以通过三个数值就可以实现最基本的 pwm 功能。其中 period 代表频率，duty_cycle 代表空占比，enable 代表使能开关，

设置 *pwm* 波的参数

```
echo 1000000 > /sys/class/pwm/pwmchip0/pwm0/period
echo 500000 > /sys/class/pwm/pwmchip0/pwm0/duty_cycle
echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable
echo 0 > /sys/class/pwm/pwmchip0/pwm0/enable
```

假设 cpu 频率是 1GHz，而我们希望得到的 pwm 频率是 1000，那么这里的 period 就是 1G/1000，而 duty_cycle 被设置成了 500000，代表空占比是 50%，enable 为 1 代表打开，0 则代表关闭。通道 1 也是这个道理，用同样的方法配置一下即可。

使用逻辑分析仪来观察一下。

第十四章 PWM 驱动示例-2

驱动程序如下

pwm_driver.c

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/pwm.h>

/* Meta Information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes 4 GNU/Linux");
MODULE_DESCRIPTION("A simple driver to access the Hardware PWM IP");

/* Variables for device and device class */
static dev_t my_device_nr;
static struct class *my_class;
static struct cdev my_device;

#define DRIVER_NAME "my_pwm_driver"
#define DRIVER_CLASS "MyModuleClass"

/* Variables for pwm */
struct pwm_device *pwm0 = NULL;
u32 pwm_on_time = 500000000;

/**
 * @brief Write data to buffer
 */
static ssize_t driver_write(struct file *File, const char *user_buffer, size_t
    count, loff_t *offs) {
    int to_copy, not_copied, delta;
    char value;

    /* Get amount of data to copy */
    to_copy = min(count, sizeof(value));

    /* Copy data to user */
```

```
36      not_copied = copy_from_user(&value, user_buffer, to_copy);
37
38      /* Set PWM on time */
39      if(value < 'a' || value > 'j')
40          printk("Invalid Value\n");
41      else
42          pwm_config(pwm0, 100000000 * (value - 'a'), 1000000000);
43
44      /* Calculate data */
45      delta = to_copy - not_copied;
46
47      return delta;
48  }
49
50  /**
51   * @brief This function is called, when the device file is opened
52   */
53  static int driver_open(struct inode *device_file, struct file *instance) {
54      printk("dev_nr - open was called!\n");
55      return 0;
56  }
57
58  /**
59   * @brief This function is called, when the device file is opened
60   */
61  static int driver_close(struct inode *device_file, struct file *instance) {
62      printk("dev_nr - close was called!\n");
63      return 0;
64  }
65
66  static struct file_operations fops = {
67      .owner = THIS_MODULE,
68      .open = driver_open,
69      .release = driver_close,
70      .write = driver_write
71  };
72
73  /**
74   * @brief This function is called, when the module is loaded into the kernel
75   */
76  static int __init ModuleInit(void) {
```

```
77      printk("Hello, Kernel!\n");
78
79      /* Allocate a device nr */
80      if( alloc_chrdev_region(&my_device_nr, 0, 1, DRIVER_NAME) < 0) {
81          printk("Device Nr. could not be allocated!\n");
82          return -1;
83      }
84      printk("read_write - Device Nr. Major: %d, Minor: %d was registered!\n",
            my_device_nr >> 20, my_device_nr && 0xfffff);
85
86      /* Create device class */
87      if((my_class = class_create(THIS_MODULE, DRIVER_CLASS)) == NULL) {
88          printk("Device class can not be created!\n");
89          goto ClassError;
90      }
91
92      /* create device file */
93      if(device_create(my_class, NULL, my_device_nr, NULL, DRIVER_NAME) == NULL)
            {
94          printk("Can not create device file!\n");
95          goto FileError;
96      }
97
98      /* Initialize device file */
99      cdev_init(&my_device, &fops);
100
101     /* Regisering device to kernel */
102     if(cdev_add(&my_device, my_device_nr, 1) == -1) {
103         printk("Registering of device to kernel failed!\n");
104         goto AddError;
105     }
106
107     pwm0 = pwm_request(0, "my-pwm");
108     if(pwm0 == NULL) {
109         printk("Could not get PWM0!\n");
110         goto AddError;
111     }
112
113     pwm_config(pwm0, pwm_on_time, 1000000000);
114     pwm_enable(pwm0);
115
```

```
116     return 0;
117 AddError:
118     device_destroy(my_class, my_device_nr);
119 FileError:
120     class_destroy(my_class);
121 ClassError:
122     unregister_chrdev_region(my_device_nr, 1);
123     return -1;
124 }
125 
126 /**
127  * @brief This function is called, when the module is removed from the kernel
128  */
129 static void __exit ModuleExit(void) {
130     pwm_disable(pwm0);
131     pwm_free(pwm0);
132     cdev_del(&my_device);
133     device_destroy(my_class, my_device_nr);
134     class_destroy(my_class);
135     unregister_chrdev_region(my_device_nr, 1);
136     printk("Goodbye, Kernel\n");
137 }
138 
139 module_init(ModuleInit);
140 module_exit(ModuleExit);
```

Makefile 如下

Makefile

```
1 obj-m += pwm_driver.o
2 
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5 
6 clean:
7     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

第十五章 LCD 驱动示例

lcd_driver.c

```c
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/gpio.h>
#include <linux/delay.h>

/* Meta Information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes 4 GNU/Linux");
MODULE_DESCRIPTION("A driver to write to a LCD text display");

/* Variables for device and device class */
static dev_t my_device_nr;
static struct class *my_class;
static struct cdev my_device;

#define DRIVER_NAME "lcd"
#define DRIVER_CLASS "MyModuleClass"

/* LCD char buffer */
static char lcd_buffer[17];

/* Pinout for LCD Display */
unsigned int gpios[] = {
    3, /* Enable Pin */
    2, /* Register Select Pin */
    4, /* Data Pin 0*/
    17, /* Data Pin 1*/
    27, /* Data Pin 2*/
    22, /* Data Pin 3*/
    10, /* Data Pin 4*/
    9, /* Data Pin 5*/
    11, /* Data Pin 6*/
    5, /* Data Pin 7*/
};
```

```
39
40 #define REGISTER_SELECT gpios[1]
41
42 /**
43  * @brief generates a pulse on the enable signal
44  */
45 void lcd_enable(void) {
46    gpio_set_value(gpios[0], 1);
47    msleep(5);
48    gpio_set_value(gpios[0], 0);
49 }
50
51 /**
52  * @brief set the 8 bit data bus
53  * @param data: Data to set
54  */
55 void lcd_send_byte(char data) {
56    int i;
57    for(i=0; i<8; i++)
58       gpio_set_value(gpios[i+2], ((data) & (1<<i)) >> i);
59    lcd_enable();
60    msleep(5);
61 }
62
63 /**
64  * @brief send a command to the LCD
65  *
66  * @param data: command to send
67  */
68 void lcd_command(uint8_t data) {
69     gpio_set_value(REGISTER_SELECT, 0); /* RS to Instruction */
70    lcd_send_byte(data);
71 }
72
73 /**
74  * @brief send a data to the LCD
75  *
76  * @param data: command to send
77  */
78 void lcd_data(uint8_t data) {
79     gpio_set_value(REGISTER_SELECT, 1); /* RS to data */
```

```
80      lcd_send_byte(data);
81  }
82
83
84  /**
85   * @brief Write data to buffer
86   */
87  static ssize_t driver_write(struct file *File, const char *user_buffer, size_t
        count, loff_t *offs) {
88      int to_copy, not_copied, delta, i;
89
90      /* Get amount of data to copy */
91      to_copy = min(count, sizeof(lcd_buffer));
92
93      /* Copy data to user */
94      not_copied = copy_from_user(lcd_buffer, user_buffer, to_copy);
95
96      /* Calculate data */
97      delta = to_copy - not_copied;
98
99      /* Set the new data to the display */
100     lcd_command(0x1);
101
102     for(i=0; i<to_copy; i++)
103         lcd_data(lcd_buffer[i]);
104
105     return delta;
106 }
107
108 /**
109  * @brief This function is called, when the device file is opened
110  */
111 static int driver_open(struct inode *device_file, struct file *instance) {
112     printk("dev_nr - open was called!\n");
113     return 0;
114 }
115
116 /**
117  * @brief This function is called, when the device file is opened
118  */
119 static int driver_close(struct inode *device_file, struct file *instance) {
```

```
120     printk("dev_nr - close was called!\n");
121     return 0;
122 }
123
124 static struct file_operations fops = {
125     .owner = THIS_MODULE,
126     .open = driver_open,
127     .release = driver_close,
128     .write = driver_write
129 };
130
131 /**
132  * @brief This function is called, when the module is loaded into the kernel
133  */
134 static int __init ModuleInit(void) {
135     int i;
136     char *names[] = {"ENABLE_PIN", "REGISTER_SELECT", "DATA_PIN0", "DATA_PIN1",
            "DATA_PIN2", "DATA_PIN3", "DATA_PIN4", "DATA_PIN5", "DATA_PIN6", "
        DATA_PIN7"};
137     printk("Hello, Kernel!\n");
138
139     /* Allocate a device nr */
140     if( alloc_chrdev_region(&my_device_nr, 0, 1, DRIVER_NAME) < 0) {
141         printk("Device Nr. could not be allocated!\n");
142         return -1;
143     }
144     printk("read_write - Device Nr. Major: %d, Minor: %d was registered!\n",
            my_device_nr >> 20, my_device_nr && 0xfffff);
145
146     /* Create device class */
147     if((my_class = class_create(THIS_MODULE, DRIVER_CLASS)) == NULL) {
148         printk("Device class can not be created!\n");
149         goto ClassError;
150     }
151
152     /* create device file */
153     if(device_create(my_class, NULL, my_device_nr, NULL, DRIVER_NAME) == NULL)
            {
154         printk("Can not create device file!\n");
155         goto FileError;
156     }
```

```
157
158     /* Initialize device file */
159     cdev_init(&my_device, &fops);
160
161     /* Regisering device to kernel */
162     if(cdev_add(&my_device, my_device_nr, 1) == -1) {
163         printk("lcd-driver - Registering of device to kernel failed!\n");
164         goto AddError;
165     }
166
167     /* Initialize GPIOs */
168     printk("lcd-driver - GPIO Init\n");
169     for(i=0; i<10; i++) {
170         if(gpio_request(gpios[i], names[i])) {
171             printk("lcd-driver - Error Init GPIO %d\n", gpios[i]);
172             goto GpioInitError;
173         }
174     }
175
176     printk("lcd-driver - Set GPIOs to output\n");
177     for(i=0; i<10; i++) {
178         if(gpio_direction_output(gpios[i], 0)) {
179             printk("lcd-driver - Error setting GPIO %d to output\n", i);
180             goto GpioDirectionError;
181         }
182     }
183
184     /* Init the display */
185     lcd_command(0x30); /* Set the display for 8 bit data interface */
186
187     lcd_command(0xf); /* Turn display on, turn cursor on, set cursor blinking
        */
188
189     lcd_command(0x1);
190
191     char text[] = "Hello World!";
192     for(i=0; i<sizeof(text)-1;i++)
193         lcd_data(text[i]);
194
195     return 0;
196 GpioDirectionError:
```

```
197     i=9;
198 GpioInitError:
199     for(;i>=0; i--)
200         gpio_free(gpios[i]);
201 AddError:
202     device_destroy(my_class, my_device_nr);
203 FileError:
204     class_destroy(my_class);
205 ClassError:
206     unregister_chrdev_region(my_device_nr, 1);
207     return -1;
208 }
209 
210 /**
211  * @brief This function is called, when the module is removed from the kernel
212  */
213 static void __exit ModuleExit(void) {
214     int i;
215     lcd_command(0x1); /* Clear the display */
216     for(i=0; i<10; i++){
217         gpio_set_value(gpios[i], 0);
218         gpio_free(gpios[i]);
219     }
220     cdev_del(&my_device);
221     device_destroy(my_class, my_device_nr);
222     class_destroy(my_class);
223     unregister_chrdev_region(my_device_nr, 1);
224     printk("Goodbye, Kernel\n");
225 }
226 
227 module_init(ModuleInit);
228 module_exit(ModuleExit);
```

Makefile

```
1 obj-m += lcd_driver.o
2 
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5 
```

```
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

第十六章 I2C 驱动程序示例