华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象
华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《Android 系统移植和驱动开发》

作者：华清远见

专业始于专注 卓识源于远见

第 6 章 Android 驱动编程

本章目标

本章将进入到 Android 的内核空间，初步介绍嵌入式 Android 设备驱动的开发。驱动的开发流程相对于应用程序的开发是全新的，与读者以前的编程习惯完全不同，希望读者能尽快熟悉环境。本章主要内容：

- 设备驱动概述
- 字符设备驱动编程
- GPIO 驱动程序实例
- 4×4 扫描键盘驱动

6.1 Android 内核内核模块编程

1．设备驱动和内核模块

Android 内核中采用可加载的模块化设计（Loadable Kernel Modules，LKMs），一般情况下编译的 Android 内核是支持可插入式模块的，也就是将最基本的核心代码编译在内核中，其他的代码可以编译到内核中或者编译为内核的模块文件（在需要时动态加载）。

Android 内核设备驱动属于内核的一部分，Android 内核的一个模块可以以如下两种方式被编译和加载：

- 直接编译进 Android 内核内核，随同 Android 内核启动时加载。
- 编译成一个可加载和删除的模块，使用 insmod 加载（modprobe 和 insmod 命令类似，但依赖于相关的配置文件）、rmmod 删除。这种方式控制了内核的大小，而模块一旦被插入内核，它就与内核其他部分一样。

常见的驱动程序是作为内核模块动态加载的，如声卡驱动和网卡驱动等，而 Android 内核最基础的驱动，如 CPU、PCI 总线、TCP/IP、APM（高级电源管理）、VFS 等驱动程序则直接编译在内核文件中。有时也把内核模块称为驱动程序，只不过驱动的内容不一定是硬件罢了，如 Ext3 文件系统的驱动。因此，加载驱动就是加载内核模块。

2．模块相关命令

lsmod 列出了当前系统中加载的模块，其中左边第一列是模块名，第二列是该模块大小，第三列则是使用该模块的对象数目，代码如下：

```
$ lsmod
Module              Size     Used by
Autofs              12068    0  (autoclean) (unused)
eepro100            18128    1
iptable_nat         19252    0  (autoclean) (unused)
ip_conntrack        18540    1  (autoclean) [iptable_nat]
iptable_mangle      2272     0  (autoclean) (unused)
iptable_filter      2272     0  (autoclean) (unused)
ip_tables           11936    5  [iptable_nat iptable_mangle iptable_filter]
usb-ohci            19328    0  (unused)
usbcore             54528    1  [usb-ohci]
ext3                67728    2
jbd                 44480    2  [ext3]
aic7xxx             114704   3
sd_mod              11584    3
scsi_mod            98512    2  [aic7xxx sd_mod]
```

- rmmod 用于将当前模块卸载。
- insmod 和 modprobe 用于加载当前模块，但 insmod 不会自动解除依存关系，即如果要加载的模块引用了当前内核符号表中不存在的符号，则无法加载，也不会去查在其他尚未加载的模块中是否定义了该符号；modprobe 可以根据模块间依存关系，以及/etc/modules.conf 文件中的内容自动加载其他有依赖关系的模块。

/proc 文件系统是一个伪文件系统，它是一种内核和内核模块用来向进程发送信息的机制。这个伪文件系统让用户可以和内核内部数据结构进行交互，获取有关系统和进程的有用信息，在运行时通过改变内核参数来改变设置。与其他文件系统不同，/proc 存在于内存之中而不是在硬盘上。读者可以通过“ls”命令查看/proc 文件系统的内容。

表 6.1 列出了/proc 文件系统的主要目录内容。

表 6.1 /proc 文件系统的主要目录内容

目 录 名 称	目 录 内 容	目 录 名 称	目 录 内 容
apm	高级电源管理信息	locks	内核锁
cmdline	内核命令行	meminfo	内存信息

cpuinfo	CPU 相关信息	misc	杂项
devices	设备信息（块设备/字符设备）	modules	加载模块列表
dma	使用的 DMA 通道信息	mounts	加载的文件系统
filesystems	支持的文件系统信息	partitions	系统识别的分区表
interrupts	中断的使用信息	rtc	实时时钟
ioports	I/O 端口的使用信息	stat	全面统计状态表
kcore	内核映像	swaps	对换空间的利用情况
kmsg	内核消息	version	内核版本
ksyms	内核符号表	uptime	系统正常运行时间
loadavg	负载均衡	……	……

除此之外，还有一些是以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口。进程目录的结构如表 6.2 所示。

表 6.2 /proc 中进程目录结构

目录名称	目录内容	目录名称	目录内容
cmdline	命令行参数	cwd	当前工作目录的链接
environ	环境变量值	exe	指向该进程的执行命令文件
fd	一个包含所有文件描述符的目录	maps	内存映像
mem	进程的内存被利用情况	statm	进程内存状态信息
stat	进程状态	root	链接此进程的 root 目录
status	进程当前状态，以可读的方式显示出来	……	……

华清远见

用户可以使用 cat 命令来查看其中的内容。

可以看到，/proc 文件系统体现了内核及进程运行的内容，在加载模块成功后，读者可以通过查看/proc/device 文件获得相关设备的主设备号。每个内核模块程序可以在任何时候到/proc 文件系统中添加或删除自己的入口点（文件），通过该文件导出自己的信息。

但后来在新的内核版本中，内核开发者不提倡在/proc 下添加文件，而建议新的代码通过 sysfs 来向外导出信息。

3. Android 内核内核模块编程

1）内核模块的程序结构

一个 Android 内核内核模块主要由以下几个部分组成。

- 模块加载函数（必需）：当通过 insmod 或 modprobe 命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块的相关初始化工作。
- 模块卸载函数（必需）：当通过 rmmod 命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块加载函数相反的功能。
- 模块许可证声明（必需）：模块许可证（LICENSE）声明描述内核模块的许可权限，如果不声明 LICENSE，模块被加载时，将收到内核被污染（Kernel Tainted）的警告。在 Android 2.6 内核中，可接受的 LICENSE 包括“GPL”、“GPL v2”、“GPL and additional rights”、“Dual BSD/GPL”、“Dual MPL/GPL”和“Proprietary”。大多数情况下，内核模块应遵循 GPL 兼容许可权。Android 2.6 内核模块最常见的是以 MODULE_LICENSE("Dual BSD/GPL")语句声明模块采用 BSD/GPL 双许可。

- ❑ 模块参数（可选）：模块参数是模块被加载时可以被传递给它的值，它本身对应模块内部的全局变量。
- ❑ 模块导出符号（可选）：内核模块可以导出符号（symbol，对应于函数或变量），这样其他模块可以使用本模块中的变量或函数。
- ❑ 模块作者等信息声明（可选）。

2）模块加载函数

Android 内核模块加载函数一般以__init 标识声明，典型的模块加载函数的形式如下：

```
static int __init initialization_function(void)
{
 /* 初始化代码 */
}
module_init(initialization_function);
```

模块加载函数必须以“module_init(函数名)”的形式被指定。它返回整型值，若初始化成功，应返回 0。而在初始化失败时，应该返回错误编码。在 Android 内核中，错误编码是一个负值，在<Android 内核/errno.h>中定义，包含-ENODEV、-ENOMEM 之类的符号值。返回相应的错误编码是个非常好的习惯，因为只有这样，用户程序才可以利用 perror 等方法把它们转换成有意义的错误信息字符串。

在 Android 2.6 内核中，可以使用 request_module(const char *fmt, …)函数加载内核模块，驱动开发人员可以通过调用

```
request_module(module_name);
```

或

```
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

来加载其他内核模块。

在 Android 内核内核中，所有标识为__init 的函数在连接时都放在.init.text 这个区段内，此外，所有的__init 函数在区段.initcall.init 中还保存了一些函数指针，在初始化时内核会通过这些函数指针调用这些__init 函数，并在初始化完成后释放 init 区段（包括.init.text，.initcall.init 等）。

3）模块卸载函数

Android 内核模块卸载函数一般以__exit 标识声明，典型的模块卸载函数的形式如下：

```
static void __exit cleanup_function(void)
{
 /* 释放代码 */
}
module_exit(cleanup_function);
```

模块卸载函数在模块卸载时执行，不返回任何值，必须以“module_exit(函数名)”的形式来指定。

通常，模块卸载函数要完成与模块加载函数相反的功能，介绍如下。

- ❑ 若模块加载函数注册了 XXX，则模块卸载函数应该注销 XXX。
- ❑ 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。
- ❑ 若模块加载函数申请了硬件资源（如中断、DMA 通道、I/O 端口和 I/O 内存等）的占用，则模块卸载函数应释放这些硬件资源。
- ❑ 若模块加载函数开启了硬件，则卸载函数中一般要关闭硬件。

与__init 一样，__exit 也可以使对应函数在运行完成后自动回收内存。实际上，__init 和__exit 都是宏，其定义分别如下：

```
#define __init __attribute__ ((__section__ (".init.text")))
```

和

```
#ifdef MODULE
#define __exit __attribute__ ((__section__(".exit.text")))
#else
#define __exit __attribute_used__
__attribute__ ((__section__(".exit.text")))
```

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

```
#endif
```

数据也可以被定义为__initdata 和__exitdata，这两个宏分别如下：

```
#define __initdata __attribute__ ((__section__ (".init.data")))
```

和

```
#define __exitdata __attribute__ ((_ _section__(".exit.data")))
```

4）模块参数

我们可以用“module_param(参数名,参数类型,参数读/写权限)”为模块定义一个参数，下列代码定义了一个整型参数和一个字符指针参数：

```
static char *str_param = "Android 内核 Module Program";
static int num_param = 4000;
module_param(num_param, int, S_IRUGO);
module_param(str_param, charp, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为“insmode（或 modprobe）模块名 参数名=参数值”，如果不传递，参数将使用模块内定义的默认值。

参数类型可以是 byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool 或 invbool（布尔的反），在模块被编译时会将 module_param 中声明的类型与变量定义的类型进行比较，判断是否一致。

模块被加载后，在/sys/module/目录下将出现以此模块名命名的目录。当“参数读/写权限”为 0 时，表示此参数不存在 sysfs 文件系统下对应的文件节点，如果此模块存在“参数读/写权限”不为 0 的命令行参数，则在此模块的目录下还将出现 parameters 目录，包含一系列以参数名命名的文件节点，这些文件的权限值就是传入 module_param()的“参数读/写权限”，而文件的内容为参数的值。通常使用 <Android 内核/stat.h> 中定义的值来表示权限值，例如，使用 S_IRUGO 作为参数可以被所有人读取, 但是不能改变; S_IRUGO|S_IWUSR 允许 root 来改变参数。

除此之外，模块也可以拥有参数数组，形式为“module_param_array(数组名,数组类型,数组长,参数读/写权限)”。从 2.6.0 到 2.6.10 版本，需将数组长变量名赋给“数组长”，从 2.6.10 版本开始，需将数组长变量的指针赋给“数组长”，当不需要保存实际输入的数组元素个数时，可以设置“数组长”为 NULL。

运行 insmod 或 modprobe 命令时，应使用逗号分隔输入的数组元素。

5）导出符号

Android 2.6 的“/proc/kallsyms”文件对应着内核符号表，它记录了符号及符号所在的内存地址。模块可以使用如下宏导出符号到内核符号表中：

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出的符号将可以被其他模块使用，使用前声明一下即可。EXPORT_SYMBOL_GPL()只适用于包含 GPL 许可权的模块。

6）模块声明与描述

在 Android 内核内核模块中，可以用 MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_VERSION、MODULE_DEVICE_TABLE、MODULE_ALIAS 分别声明模块的作者、描述、版本、设备表和别名，例如：

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

对于 USB、PCI 等设备驱动，通常会创建一个 MODULE_DEVICE_TABLE。

7）模块的使用计数

Android 2.4 内核中，模块自身通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT 宏来管理自己被使用的计数。

华清远见

Android 2.6 内核提供了模块计数管理接口 try_module_get(&module)和 module_put (&module)，从而取代 Android 2.4 内核中的模块使用计数管理宏。模块的使用计数一般不必由模块自身管理，而且模块计数管理还考虑了 SMP 与 PREEMPT 机制的影响。

```
int try_module_get(struct module *module);
```

该函数用于增加模块使用计数；若返回为 0，则表示调用失败，希望使用的模块没有被加载或正在被卸载中。

```
void module_put(struct module *module);
```

该函数用于减少模块使用计数。

try_module_get ()与 module_put()的引入与使用与 Android 2.6 内核下的设备模型密切相关。Android 2.6 内核为不同类型的设备定义了 struct module *owner 域，用来指向管理此设备的模块。当开始使用某个设备时，内核使用 try_module_get(dev->owner)去增加管理此设备的 owner 模块的使用计数；当不再使用此设备时，内核使用 module_put(dev->owner)减少对管理此设备的 owner 模块的使用计数。这样，当设备在使用时，管理此设备的模块将不能被卸载。只有当设备不再被使用时，模块才允许被卸载。

在 Android 2.6 内核下，对于设备驱动工程师而言，很少需要亲自调用 try_module_get()与 module_put()，因为此时开发人员所写的驱动通常为支持某具体设备的 owner 模块，对此设备 owner 模块的计数管理由内核里更底层的代码（如总线驱动或此类设备共用的核心模块）来实现，从而简化了设备驱动的开发。

8）模块的编译

我们可以为 HelloWorld 模块程序编写一个简单的 Makefile，如下：

```
obj-m := hello.o
```

并使用如下命令编译 HelloWorld 模块：

```
$ make -C /usr/src/Android内核-2.6.15.5/ M=/driver_study/ modules
```

如果当前处于模块所在的目录，则以下命令与上述命令同等：

```
$ make -C /usr/src/Android内核-2.6.15.5 M=$(pwd) modules
```

其中，-C 后指定的是 Android 内核源代码的目录，而“M=”后指定的是 hello.c 和 Makefile 所在的目录，编译结果如下：

```
$ make -C /usr/src/Android内核-2.6.15.5/ M=/driver_study/ modules
make: Entering directory '/usr/src/Android内核-2.6.15.5'
  CC [M]  /driver_study/hello.o
/driver_study/hello.c:18:35: warning: no newline at end of file
  Building modules, stage 2.
  MODPOST
  CC      /driver_study/hello.mod.o
  LD [M]  /driver_study/hello.ko
make: Leaving directory '/usr/src/Android内核-2.6.15.5'
```

从中可以看出，编译过程中经历了这样的步骤：先进入 Android 内核所在的目录，并编译出 hello.o 文件，运行 MODPOST 会生成临时的 hello.mod.c 文件，然后根据此文件编译出 hello.mod.o，之后链接 hello.o 和 hello.mod.o 文件得到模块目标文件 hello.ko，最后离开 Android 内核内核所在的目录。

中间生成的 hello.mod.c 文件的源代码如下：

```
1      #include <Android内核/module.h>
2      #include <Android内核/vermagic.h>
3      #include <Android内核/compiler.h>
4
5      MODULE_INFO(vermagic, VERMAGIC_STRING);
6
7      struct module __this_module
8      __attribute__((section(".gnu.linkonce.this_module"))) = {
9       .name = KBUILD_MODNAME,
10      .init = init_module,
11     #ifdef CONFIG_MODULE_UNLOAD
12      .exit = cleanup_module,
```

```
13    #endif
14    };
15
16    static const char __module_depends[]
17    __attribute_used__
18    __attribute__((section(".modinfo"))) =
19    "depends=";
```

hello.mod.o 产生了 ELF（Android 内核所采用的可执行/可链接的文件格式）的两个节，即 modinfo 和.gun.linkonce.this_module。

如果一个模块包括多个.c 文件（如 file1.c、file2.c），则应该以如下方式编写 Makefile：

```
obj-m := modulename.o
module-objs := file1.o file2.o
```

9）模块与 GPL

对于自己编写的驱动等内核代码，如果不编译为模块则无法绕开 GPL，编译为模块后企业在产品中使用模块，则公司对外不再需要提供对应的源代码，为了使公司产品所使用的 Android 内核操作系统支持模块，需要完成如下工作。

- 在内核编译时应该选择“Enable loadable module support”，嵌入式产品一般不需要动态卸载模块，所以“可以卸载模块”不用选，当然选了也没关系，如图 6.1 所示。

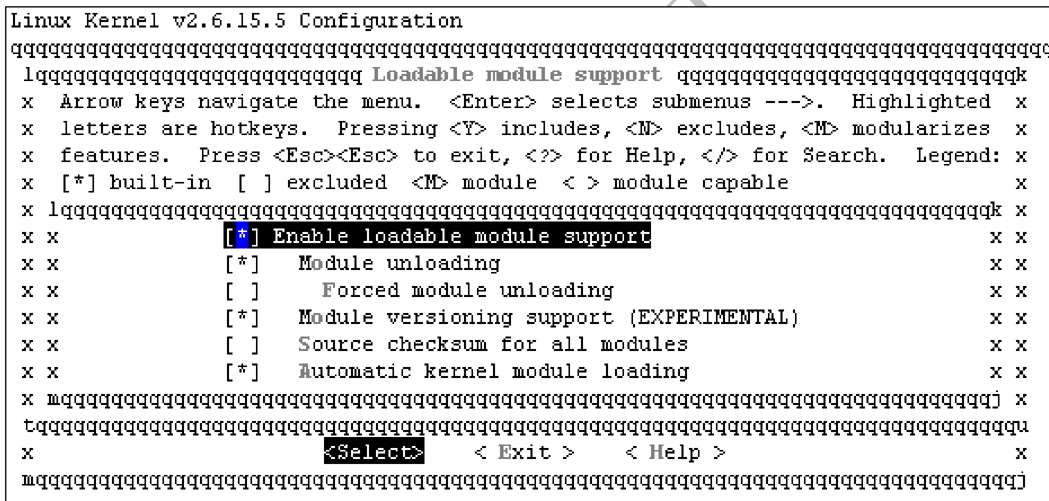

图 6.1 内核中支持模块的编译选项

如果有项目被选择“M”，则编译时除了制作镜像 make bzImage 以外，也要编译模块 make modules。

- 将编译的内核模块.ko 文件放置在目标文件系统的相关目录中。
- 产品的文件系统中应该包含了支持新内核的 insmod、lsmod、rmmod 等工具，由于嵌入式产品中一般不需要建立模块间依赖关系，所以 modprobe 可以不要，一般也不需要卸载模块，所以 rmmod 也可以不要。
- 在使用过程中用户可使用 insmod 命令手动加载模块，如 insmod xxx.ko。
- 但是一般而言，产品在启动过程中应该加载模块，在嵌入式 Android 内核的启动过程中，加载企业自己的模块的最简单方法是修改启动过程的 rc 脚本，增加 insmod /.../xxx.ko 命令。例如，某设备正在使用的 Android 内核系统中包含如下 rc 脚本：

```
mount /proc
mount /var
mount /dev/pts
mkdir /var/log
mkdir /var/run
mkdir /var/ftp
mkdir -p /var/spool/cron
mkdir /var/config
```

```
...
insmod /usr/lib/company_driver.ko 2> /dev/null
/usr/bin/userprocess
/var/config/rc
```

4. 内核模块实例程序

下面列出一个简单的内核模块程序，它的功能是统计一个字符串中的各种字符（如英文字母、数字、其他符号）的数目。模块功能的演示如下：

```
$ insmod module_test.ko symbol_type=1 string="a1b+c=d4e5[6g7h,8i9}k/l."
$ dmesg|tail -n 10
… Total symbols module init /* 在模块加载时打印*/
… Digits: 7                  /* 字符串中有 7 个数字*/
```

第一个参数（symbol_type）表示统计类型（0 为英文字母，1 为数字，2 为其他字符，3 以上为任何字符），第二个参数（string）表示需要统计的字符串。

实现该功能的模块代码如下：

```
#include <Android 内核/init.h>
#include <Android 内核/module.h>
#include <asm/string.h>
#define TOTAL_LETTERS                  0
#define TOTAL_DIGITS                   1
#define TOTAL_SYMBOLS                  2
#define TOTAL_ALL             TOTAL_LETTERS | TOTAL_DIGITS | TOTAL_SYMBOLS
#define STR_MAX_LEN           256

static int symbol_type = TOTAL_ALL;
static char *string = NULL;
unsigned int total_symbols(char* string, unsigned int total_type)
{
 /* 该函数根据统计类型（total_type）计算相应字符的个数并返回*/
 /* 这部分代码也可以从网上下载 */
}
EXPORT_SYMBOL(total_symbols);
/* 导出的符号用'cat /proc/kallsyms |grep "total_symbols"'命令检查*/
static int total_symbols_init(void)
{
 char type_str[STR_MAX_LEN];
 unsigned int number_of_symbols = 0;
 switch(symbol_type)
 {
      case TOTAL_LETTERS: /* “字母”统计 */
      {
           strcpy(type_str, "Letters");
      }
      break;
      case TOTAL_DIGITS: /* “数字”统计 */
      {
           strcpy(type_str, "Digits");
      }
      break;
      case TOTAL_SYMBOLS: /* 其他符号”统计 */
      {
           strcpy(type_str, "Symbols");
      }
      break;
      default: /* 默认为总体统计 */
      case TOTAL_ALL:
      {
           strcpy(type_str, "Characters");
      }
 }
 number_of_symbols = total_symbols(string, symbol_type);
 printk("<1>Total symbols module init\n");
```

```
  printk("<1>%s: %d\n", type_str, number_of_symbols);
      return 0;
}
static void total_symbols_exit(void)
{
  printk("<1>Total symbols module exit\n");
}

module_init(total_symbols_init); /* 初始化设备驱动程序的入口 */
module_exit(total_symbols_exit); /* 卸载设备驱动程序的入口 */
module_param(symbol_type, uint, S_IRUGO);
module_param(string, charp, S_IRUGO);
MODULE_AUTHOR("David");
MODULE_DESCRIPTION("A simple module program");
MODULE_VERSION("V1.0");
```

6.2 字符设备驱动编程

6.2.1 字符设备驱动编写流程

在 6.1 节中已经提到，设备驱动程序可以使用模块的方式动态加载到内核中去。加载模块的方式与以往的应用程序开发有很大的不同。

以往在开发应用程序时都有一个 main()函数作为程序的入口点，而在驱动开发时却没有 main()函数，模块在调用 insmod 命令时被加载，此时的入口点是 module_init()函数，通常在该函数中完成设备的注册。同样，模块在调用 rmmod 命令时被卸载，此时的入口点是 module_exit()函数，在该函数中完成设备的卸载。

在设备完成注册加载之后，应用程序即对该设备进行一定的操作，如 open()、read()、write()等，而驱动程序就是用于实现这些操作，在应用程序调用相应入口函数时执行相关的操作。上述函数之间的关系如图 6.2 所示。

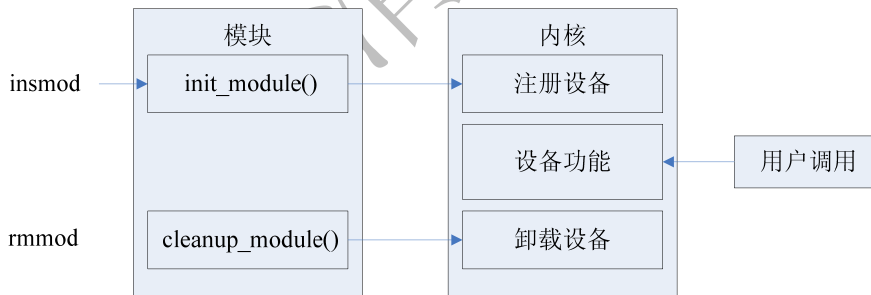

图 6.2 设备驱动程序流程图

6.2.2 重要数据结构

在 Android 内核驱动程序中，涉及 3 个重要的内核数据结构，分别是 file_operation、file 和 inode。在 Android 内核中 inode 结构用于表示文件，而 file 结构则表示打开的文件描述符，因为对于单个文件而言可能会有许多表示打开的文件描述符，因此就可能会对应有多个 file 结构，但它们都指向单个 inode 结构。此外，每个 file 结构都与一组函数相关联，这组函数是用 file_operations 结构来指示的。

用户应用程序调用设备的一些功能是在设备驱动程序中定义的，也就是设备驱动程序的入口点，它是一个在<Android 内核/fs.h>中定义的 struct file_operations 结构，file_operations 是 Android 内核驱动程序中最为重要的一个结构，它定义了一组常见文件 I/O 函数，这些函数在不同的驱动程序中会有不同的具体实现，其结构如下：

```
struct file_operations
{
  loff_t (*llseek) (struct file *, loff_t, int);
```

```
  ssize_t (*read) (struct file *filp,
char *buff, size_t count, loff_t *offp);
  ssize_t (*write) (struct file *filp,
const char *buff, size_t count, loff_t *offp);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *,
struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *);
  int (*fasync) (int, struct file *, int);
  int (*check_media_change) (kdev_t dev);
  int (*revalidate) (kdev_t dev);
  int (*lock) (struct file *, int, struct file_lock *);
};
```

这里定义的很多函数是否与第 5 章中文件 I/O 的系统调用类似？其实当时的系统调用函数通过内核，最终调用对应的 file_operations 结构的接口函数（例如，open()文件操作是通过调用对应文件的 file_operations 结构的 open 函数接口而被实现）。当然，每个设备的驱动程序不一定要实现其中所有的函数操作，若不需要定义实现，将其设为 NULL 即可。

struct inode 结构提供了关于设备文件/dev/driver（假设此设备名为 driver）的信息，file 结构提供关于被打开的文件信息，主要供与文件系统对应的设备驱动程序使用。file 结构较为重要，这里列出了它的定义：

```
struct file
{
  mode_t f_mode;/*标识文件是否可读或可写，FMODE_READ 或 FMODE_WRITE*/
  dev_t f_rdev; /* 用于/dev/tty */
  off_t f_pos; /* 当前文件位移 */
  unsigned short f_flags; /* 文件标志，如 O_RDONLY、O_NONBLOCK 和 O_SYNC */
  unsigned short f_count; /* 打开的文件数目 */
  unsigned short f_reada;
  struct inode *f_inode; /*指向 inode 的结构指针 */
  struct file_operations *f_op;/* 文件索引指针 */
};
```

6.2.3 设备驱动程序主要组成

1．早期版本的字符设备注册

早期版本的设备注册使用函数 register_chrdev()，调用该函数后就可以向系统申请主设备号，如果 register_chrdev()操作成功，设备名就会出现在/proc/devices 文件中。在关闭设备时，通常需要解除原先的设备注册，此时可使用函数 unregister_chrdev()，然后该设备就会从/proc/devices 中消失。其中主设备号和次设备号不能大于 255。

当前不少的字符设备驱动代码仍然使用这些早期版本的函数接口，但在未来内核的代码中，将不会出现这种编程接口机制。因此，应尽量使用后面讲述的编程机制。

register_chrdev()函数语法要点如表 6.3 所示。

表 6.3 register_chrdev()函数语法要点

所需头文件	#include <Android 内核/fs.h>
函数原型	int register_chrdev(unsigned int major, const char *name,struct file_operations *fops)
函数传入值	major：设备驱动程序向系统申请的主设备号，如果为 0，则系统为此驱动程序动态地分配一个主设备号
	name：设备名
	fops：对各个调用的入口点

函数返回值	成功：如果是动态分配主设备号，则返回所分配的主设备号。且设备名会出现在/proc/devices 文件中
	出错：–1

unregister_chrdev()函数语法要点如表 6.4 所示。

表 6.4 unregister_chrdev()函数语法要点

所需头文件	#include <Android 内核/fs.h>
函数原型	int unregister_chrdev(unsigned int major, const char *name)
函数传入值	major：设备的主设备号，必须和注册时的主设备号相同
	name：设备名
函数返回值	成功：0，且设备名从/proc/devices 文件中消失
	出错：–1

2. 设备号相关函数

设备号是一个数字，它是设备的标志。设备号有主设备号和次设备号，其中主设备号表示设备类型，对应于确定的驱动程序，具备相同主设备号的设备之间共用同一个驱动程序，而用次设备号来标识具体物理设备。因此，在创建字符设备之前，必须先获得设备的编号（可能需要分配多个设备号）。

在 Android 2.6 中，用 dev_t 类型来描述设备号（dev_t 是 32 位数值类型，其中高 12 位表示主设备号，低 20 位表示次设备号）。用两个宏 MAJOR 和 MINOR 分别获得 dev_t 设备号的主设备号和次设备号，而且用 MKDEV 宏来实现逆过程，即组合主设备号和次设备号而获得 dev_t 类型设备号。

```
#include <Android 内核/kdev.h>
MAJOR(dev_t dev); /*获得主设备号*/
MINOR(dev_t dev); /*获得次设备号*/
MKDEV(int major, int minor);
```

分配设备号有静态和动态两种方法。静态分配（register_chrdev_region()函数）是指在事先知道设备主设备号的情况下，通过参数函数指定第一个设备号（它的次设备号通常为 0）而向系统申请分配一定数目的设备号。动态分配（alloc_chrdev_region()）是指通过参数仅设置第一个次设备号（通常为 0，事先不会知道主设备号）和要分配的设备数目而系统动态分配所需的设备号。

通过 unregister_chrdev_region()函数释放已分配的（无论是静态的还是动态的）设备号。

它们的函数格式如表 6.5 所示。

表 6.5 设备号分配与释放函数语法要点

所需头文件	#include <Android 内核/fs.h>
函数原型	int register_chrdev_region (dev_t first, unsigned int count, char *name) int alloc_chrdev_region (dev_t *dev, unsigned int firstminor, unsigned int count, char *name) void unregister_chrdev_region (dev_t first, unsigned int count)
函数传入值	first：要分配的设备号的初始值 count：要分配（释放）的设备号数目 name：要申请设备号的设备名称（在/proc/devices 和 sysfs 中显示） dev：动态分配的第一个设备号
函数返回值	成功：0（只限于两种注册函数）
	出错：–1（只限于两种注册函数）

3．最新版本的字符设备注册

在获得了系统分配的设备号之后，通过注册设备才能实现设备号和驱动程序之间的关联。这里讨论 2.6 内核中的字符设备的注册和注销过程。

在 Android 内核中使用 struct cdev 结构来描述字符设备，在驱动程序中必须将已分配到的设备号及设备操作接口（即为 struct file_operations 结构）赋予 cdev 结构变量。首先使用 cdev_alloc()函数向系统申请分配 cdev 结构，再用 cdev_init()函数初始化已分配到的结构并与 file_operations 结构关联起来。最后调用 cdev_add()函数将设备号与 struct cdev 结构进行关联并向内核正式报告新设备的注册，这样新设备可以用了。

如果要从系统中删除一个设备，则要调用 cdev_del()函数。具体函数格式如表 6.6 所示。

表 6.6 最新版本的字符设备注册

所需头文件	#include <Android 内核/cdev.h>
函数原型	sturct cdev *cdev_alloc(void) void cdev_init(struct cdev *cdev, struct file_operations *fops) int cdev_add (struct cdev *cdev, dev_t num, unsigned int count) void cdev_del(struct cdev *dev)
函数传入值	cdev：需要初始化/注册/删除的 struct cdev 结构 fops：该字符设备的 file_operations 结构 num：系统给该设备分配的第一个设备号 count：该设备对应的设备号数量
函数返回值	成功： cdev_alloc：返回分配到的 struct cdev 结构指针 cdev_add：返回 0
	出错： cdev_alloc：返回 NULL cdev_add：返回 -1

2.6 内核仍然保留早期版本的 register_chrdev()等字符设备相关函数，其实从内核代码中可以发现，在 register_chrdev()函数的实现中用到 cdev_alloc()和 cdev_add()函数，而在 unregister_chrdev()函数的实现中用到 cdev_del()函数。因此很多代码仍然使用早期版本接口，但这种机制将来会从内核中消失。

前面已经提到字符设备的实际操作在 file_operations 结构的一组函数中定义，并在驱动程序中需要与字符设备结构关联起来。下面讨论 file_operations 结构中最主要的成员函数和它们的用法。

4．打开设备

打开设备的函数接口是 open，根据设备的不同，open 函数接口完成的功能也有所不同，其原型如下：

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

通常在 open 函数接口中要完成如下工作：

- 如果未初始化，则进行初始化。
- 识别次设备号，如果必要，更新 f_op 指针。
- 分配并填写被置于 filp->private_data 的数据结构。
- 检查设备特定的错误（如设备未就绪或类似的硬件问题）。

打开计数是 open 函数接口中常见的功能，它是用于计算自从设备驱动加载以来设备被打开过的次数。由于设备在使用时通常会多次被打开，也可以由不同的进程所使用，所以若有一个进程想要删除该设备，则必须保证其他设备没有使用该设备。因此，使用计数器就可以很好地完成这项功能。

5．释放设备

释放设备的函数接口是 release()。要注意释放设备和关闭设备是完全不同的。当一个进程释放设备时，其他进程还能继续使用该设备，只是该进程暂时停止对该设备的使用，并没有真正关闭该设备；而当一个进程关闭设备时，其他进程必须重新打开此设备才能使用它。

华清远见

释放设备时要完成的工作如下：

- 释放打开设备时系统所分配的内存空间（包括 filp->private_data 指向的内存空间）。
- 在最后一次关闭设备（使用 close()系统调用）时，才会真正释放设备（执行 release()函数）。即在打开计数等于 0 时的 close()系统调用才会真正进行设备的释放操作。

6．读/写设备

读/写设备的主要任务就是把内核空间的数据复制到用户空间，或者从用户空间复制到内核空间，也就是将内核空间缓冲区中的数据复制到用户空间的缓冲区中或者相反。这里首先解释一个 read()和 write()函数的入口函数，如表 6.7 所示。

表 6.7　read、write 函数接口语法要点

所需头文件	#include <Android 内核/fs.h>
函数原型	ssize_t (*read) (struct file *filp, char *buff, size_t count, loff_t *offp) ssize_t (*write) (struct file *filp, const char *buff, size_t count, loff_t *offp)
函数传入值	filp：文件指针
	buff：指向用户缓冲区
	count：传入的数据长度
	offp：用户在文件中的位置
函数返回值	成功：写入的数据长度

虽然这个过程看起来很简单，但是内核空间地址和应用空间地址是有很大区别的，其中一个区别是用户空间的内存是可以被换出的，因此可能会出现页面失效等情况。所以不能使用诸如 memcpy()之类的函数来完成这样的操作。在这里要使用 copy_to_user()或 copy_from_user()等函数，它们是用来实现用户空间和内核空间的数据交换的。

copy_to_user()和 copy_from_user()的格式如表 6.8 所示。

表 6.8　copy_to_user()和 copy_from_user()函数语法要点

所需头文件	#include <asm/uaccess.h>
函数原型	unsigned long copy_to_user(void *to, const void *from, unsigned long count) unsigned long copy_from_user(void *to, const void *from, unsigned long count)
函数传入值	to：数据目的缓冲区
	from：数据源缓冲区
	count：数据长度
函数返回值	成功：写入的数据长度 失败：-EFAULT

要注意，这两个函数不仅实现了用户空间和内核空间的数据转换，而且还会检查用户空间指针的有效性。如果指针无效，那么就不进行复制。

7．ioctl

大部分设备除了读/写操作，还需要硬件配置和控制（例如，设置串口设备的波特率）等很多其他操作。在字符设备驱动中 ioctl 函数接口给用户提供对设备的非读/写操作机制。

ioctl 函数接口的具体格式如表 6.9 所示。

表 6.9　ioctl 函数接口语法要点

所需头文件	#include <Android 内核/fs.h>
函数原型	int(*ioctl)(struct inode* inode, struct file* filp, unsigned int cmd, unsigned long arg)
函数传入值	inode：文件的内核内部结构指针

	filp：被打开的文件描述符
	cmd：命令类型
	arg：命令相关参数

8．获取内存

在应用程序中获取内存通常使用函数 malloc()，但在设备驱动程序中动态开辟内存可以以字节或页面为单位。其中，以字节为单位分配内存的函数有 kmalloc()，需要注意的是，kmalloc()函数返回的是物理地址，而 malloc()等返回的是线性虚拟地址，因此在驱动程序中不能使用 malloc()函数。与 malloc()不同，kmalloc()申请空间有大小限制。长度是 2 的整次方，并且不会对所获取的内存空间清零。

如果驱动程序需要分配比较大的空间，使用基于页的内存分配函数会更好些。

以页为单位分配内存的函数如下：

- get_zeroed_page()函数分配一个页大小的空间并清零该空间。
- __get_free_page()函数分配一个页大小的空间，但不清零空间。
- __get_free_pages()函数分配多个物理上连续的页空间，但不清零空间。
- __get_dma_pages()函数在 DMA 的内存区段中分配多个物理上连续的页空间。

与之相对应的释放内存的函数有 kfree()或 free_page 函数族。

表 6.10 给出了 kmalloc()函数的语法要点。

表 6.10 kmalloc()函数语法要点

所需头文件	#include <Android 内核/malloc.h>	
函数原型	void *kmalloc(unsigned int len,int flags)	
函数传入值	len：希望申请的字节数	
	flags	GFP_KERNEL：内核内存的通常分配方法，可能引起睡眠
		GFP_BUFFER：用于管理缓冲区高速缓存
		GFP_ATOMIC：为中断处理程序或其他运行于进程上下文之外的代码分配内存，且不会引起睡眠
		GFP_USER：用户分配内存，可能引起睡眠
		GFP_HIGHUSER：优先高端内存分配
		__GFP_DMA：DMA 数据传输请求内存
		__GFP_HIGHMEN：请求高端内存
函数返回值	成功：写入的数据长度 失败：-EFAULT	

华清远见

表 6.11 给出了 kfree()函数的语法要点。

表 6.11 kfree()函数语法要点

所需头文件	#include <Android 内核/malloc.h>
函数原型	void kfree(void * obj)
函数传入值	obj：要释放的内存指针
函数返回值	成功：写入的数据长度 失败：-EFAULT

表 6.12 给出了以页为单位的分配函数 get_free_ page 类函数的语法要点。

表 6.12 get_free_ page 类函数语法要点

所需头文件	#include <Android 内核/malloc.h>
函数原型	unsigned long get_zeroed_page(int flags) unsigned long __get_free_page(int flags) unsigned long __get_free_page(int flags,unsigned long order) unsigned long __get_dma_pages(int flags,unsigned long order)
函数传入值	flags：同 kmalloc()
	order：要请求的页面数，以 2 为底的对数
函数返回值	成功：返回指向新分配的页面的指针 失败：-EFAULT

表 6.13 给出了基于页的内存释放函数 free_ page 族函数的语法要点。

表 6.13　free_page 类函数语法要点

所需头文件	#include <Android 内核/malloc.h>
函数原型	unsigned long free_page(unsigned long addr) unsigned long free_pages(unsigned long addr, unsigned long order)
函数传入值	addr：要释放的内存起始地址
	order：要请求的页面数，以 2 为底的对数
函数返回值	成功：写入的数据长度 失败：-EFAULT

9．打印信息

就如同在编写用户空间的应用程序，打印信息有时是很好的调试手段，也是在代码中很常用的组成部分。但是与用户空间不同，在内核空间要用函数 printk()；而不能用平常的函数 printf()。printk()和 printf()很类似，都可以按照一定的格式打印消息，不同的是，printk()还可以定义打印消息的优先级。

表 6.14 给出了 printk()函数的语法要点。

表 6.14　printk 类函数语法要点

所需头文件	#include <Android 内核/kernel>	
函数原型	int printk(const char * fmt, …)	
函数传入值	fmt： 日志级别	KERN_EMERG：紧急时间消息
		KERN_ALERT：需要立即采取动作的情况
		KERN_CRIT：临界状态，通常涉及严重的硬件或软件操作失败
		KERN_ERR：错误报告
		KERN_WARNING：对可能出现的问题提出警告
		KERN_NOTICE：有必要进行提示的正常情况
		KERN_INFO：提示性信息
		KERN_DEBUG：调试信息
	…：与 printf()相同	
函数返回值	成功：0 失败：−1	

这些不同优先级的信息输出到系统日志文件（例如：“/var/log/messages”），有时也可以输出到虚拟控制台上。其中，对输出给控制台的信息有一个特定的优先级 console_loglevel。只有打印信息的优先级小于这个整数值，信息才能被输出到虚拟控制台上，否则，信息仅被写入到系统日志文件中。若不加任何优先级选项，则消息默认输出到系统日志文件中。

6.3 LCD 控制器

6.3.1 LCD 控制器介绍

1．液晶屏的分类

液晶显示屏按显示原理分为 STN 和 TFT 两种。

STN（Super Twisted Nematic，超扭曲向列）液晶屏：STN 液晶显示器与液晶材料、光线的干涉现象有关，因此显示的色调以淡绿色与橘色为主。STN 液晶显示器中，使用 *X*、*Y* 轴交叉的单纯电极驱动方式，即 *X*、*Y* 轴由垂直与水平方向的驱动电极构成，水平方向驱动电压控制显示部分为亮或暗，垂直方向的电极则负责驱动液晶分子的显示。STN 液晶显示屏加上彩色滤光片，并将单色显示矩阵中的每一像素分成 3 个子像素，分别通过彩色滤光片显示红、绿、蓝三原色，也可以显示出色彩。单色液晶屏及灰度液晶屏都是 STN 液晶屏。

TFT（Thin Film Transistor，薄膜晶体管）彩色液晶屏：随着液晶显示技术的不断发展和进步，TFT 液晶显示屏被广泛用于制作成计算机中的液晶显示设备。 TFT 液晶显示屏既可在笔记本电脑上应用（现在大多数笔记本电脑都使用 TFT 显示屏），也可用于主流台式显示器。

2．液晶屏的显示

液晶屏的显示要求设计专门的驱动与显示控制电路。驱动电路包括提供液晶屏的驱动电源和液晶分子偏置电压，以及液晶显示屏的驱动逻辑；显示控制部分可由专门的硬件电路组成，也可以采用集成电路（IC）模块，如 EPSON、Silicon Motion 的显卡驱动器等；还可以使用处理器外围 LCD 控制模块。

6.3.2 S5PC100 LCD 控制器介绍

S5PC100 处理器集成了 LCD 控制器，主要用于传输显示数据和产生控制信号。它并支持屏幕水平和垂直滚动显示。数据的传送采用 DMA（直接内存访问）方式，以达到最小的延迟。它可以支持多种液晶屏，如 STN LCD 显示器、TFT LCD 显示控制器。

STN LCD 显示器性能如下。

- ❑ 支持 3 种类型的扫描方式：4 位单扫描、4 位双扫描和 8 位单扫描。
- ❑ 支持 256 色和 4096 色彩色 STN LCD。
- ❑ 典型的实际屏幕大小是：640×480、320×240、160×160 等。
- ❑ 最大虚拟屏幕占内存大小为 4MB。
- ❑ 256 色模式下最大虚拟屏幕大小：4096×1024、2048×2048、1024×4096 等。

TFT LCD 显示控制器性能如下：

- ❑ 支持 1、2、4 或 8bpp 彩色调色显示。
- ❑ 支持 16bpp 和 24bpp 非调色真彩显示。
- ❑ 在 24bpp 模式下，最多支持 16M 种颜色。
- ❑ 支持多种屏幕大小。
- ❑ 典型的实际屏幕大小是：640×480、320×240、160×160 等。
- ❑ 最大虚拟屏幕占内存大小为 4MB。
- ❑ 64K 色模式下最大虚拟屏幕大小：2048×1024 等。

1．S5PC100 功能简述

S5PC100 的集成 LCD 控制器功能很强大，其中包含一个本地总线传输图像数据的逻辑模块，以及内置的图像处理单元。这些模块都可通过总线连接至外接的 LCD 接口，LCD 接口包含 3 种类型，有 RGB 接口、间接的-i80 接口、ITU-R BT.601/656 接口、显示控制器支持最多 5 个叠加图像窗口，每个窗口都支持多种图像格式，以及 256 灰度级绑定、颜色锁定、x-y 坐标控制、软件卷动、可变的窗体尺寸，等等。

华清远见

显示控制器支持多种颜色格式，例如，RGB（1bpp-24bpp）、YcbCr4:4:4（限于本地总线）、显示控制器可编程支持不同需求的图像像素、数据线宽度、时序、刷新率。

2．LCD 外部接口信号

S5PC100 的 LCD 控制器包括两个时序部分，一个针对 RGB 接口 ITU-R601/656 的时序；另一个针对间接 i80 接口的时序。本书重点介绍关于 RGB 接口的控制器部分。

RGB VIME 产生的控制信号有 VSYNC（垂直同步信号）、HSYNC（水平同步信号）、VDEN（数据有效信号）、VCLK（LCD 时钟），这些信号都可由寄存器配置，还有 VD[23:0]的数据输出口。如图 6.3 所示为 LCD-RGB 接口时序。

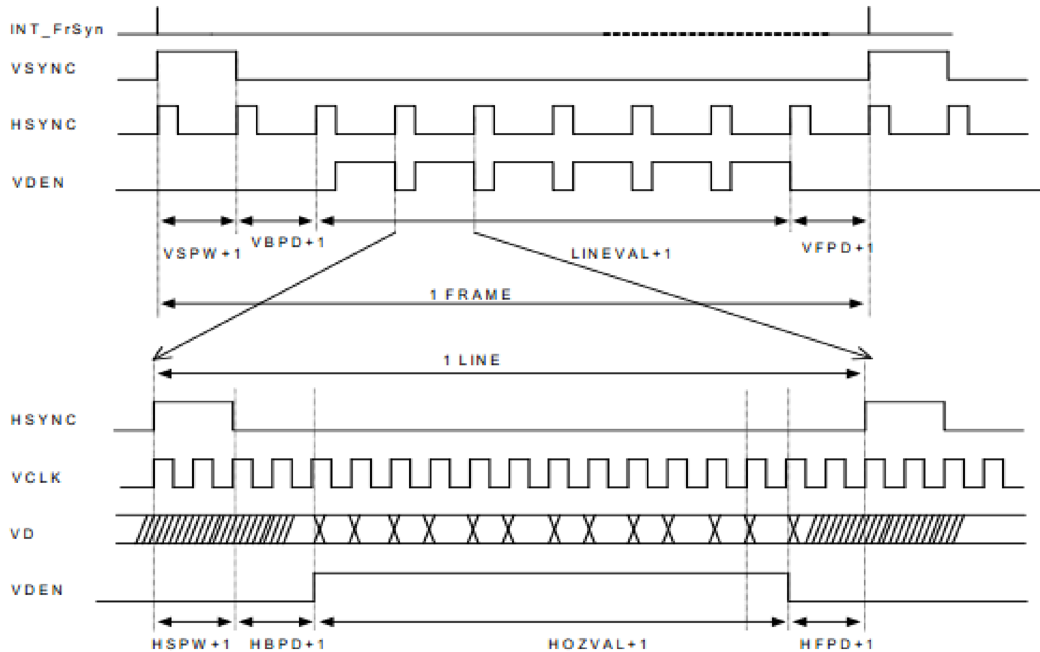

图 6.3　LCD RGB 接口时序图

6.3.3　S5PC100 LCD 控制器操作

基于前面介绍的 RGB 接口时序图，下面介绍 LCD 的控制流程。给出如下 3 个公式：

HOZVAL =（Horizontal display size -1）

LINEVAL=（Vertical display size -1）

VCLK(Hz)= HCLK/(CLKVAL+1) where CLKVAL >= 1

这些公式在配置寄存器的时候都需要使用到。下面简单解释一下 RGB 接口时序图的意义，在一帧画面的呈现中，涉及如下步骤：首先 LCD 控制器发出一次 VSYNC 信号，这时会伴随发出 HSYNC 信号，可以想象一下，LCD 屏的显示方式，先选中第一行（VSYNC 信号），然后从第一列开始顺序选中（HSYNC 信号），在每次的 HSYNC 中，会发生数据传输，这时是由 VCLK 来决定的。

在每一帧时钟信号中，还会有一些与屏显示无关的时钟出现，这就给确定行频和场频带来了一定的难度。如在 HSYNC 信号中先后会有水平同步信号前肩（HFPD）和水平同步信号后肩（HBPD）出现，在 VSYNC 信号中先后会有垂直同步信号前肩（VFPD）和垂直同步信号后肩（VBPD）出现，在这些信号时序内，不会有有效像素信号出现，另外，HSYNC 和 VSYNC 信号有效时，其电平要保持一定的时间，它们分别称为水平同步信号脉宽 HSPW 和垂直同步信号脉宽（VSPW），这段时间也不能有像素信号。因此，计算行频和场频时，一定要包括这些信号。HBPD、HFPD 和 HSPW 的单位是一个 VCLK 的时间，而 VSPW、VFPD 和 VBPD 的单位是扫描一行所用的时间。

在 S5PC100 中，还需要重点考虑 Alpha 绑定机制，因为它是 5 个窗口叠加共同呈像的原理，因此需要配置一下 Alpha 绑定方程，如图 6.4 所示。

集团官网：www.hqyj.com　嵌入式学院：www.embedu.org　企业学院：www.farsight.com.cn　北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org　物联网学院：www.topsight.cn　咨询电话：400-706-1880，15010390966

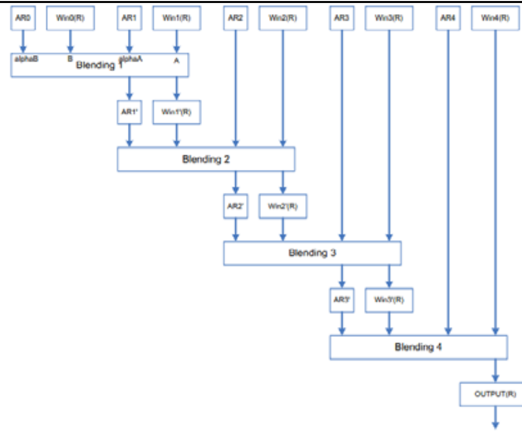

图 6.4　Alpha 绑定过程

从图中可以看到，每次叠加由两个窗口进行，窗口的顺序依次是：

（1）X0　= 窗口 0 与窗口 1。

（2）X1　=　X0 与窗口 2。

（3）X2　=　X1 与窗口 3。

（4）X4　=　X2 与窗口 4。

最后的 X4 为 LCD 所呈现的图像，如图 6.5 所示。

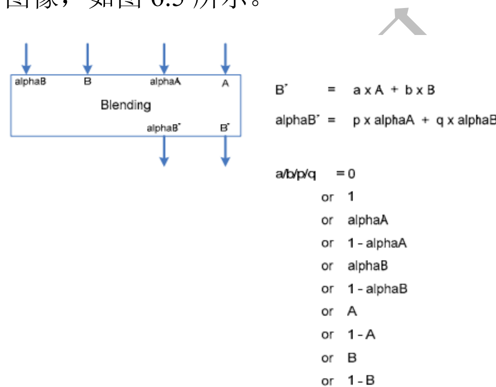

图 6.5　绑定方程

其中，B' 是一个色值函数，它由 A 的色值分量与 B 的色值分量线性叠加而成。此处，A 的色值来自 win(n+1)，B 的色值来自 win(n)，a、b 则是线性因子。我们只需要考虑 a、b 的值，就可以得到想要的结果。同理，alphaB' 是一个 B'色值的伴随灰度值函数，即一组 alpha 通道。

它的构成与前一个方程是同构的。p、q 也是线性因子，这样 4 个因子决定了两个窗口的最终叠加色值，以及伴随 alpha 的值（注意，a、b、p、q 是只能可选值，从图 6.5 中可以看出这一点）。

在 LCD 控制器中，只需要配置 WINn blending equation control register，并将具体的因子配好后，就可以实现了，注意 5 个窗口需要同时配置方程。

接下来重点考察 alphaB 灰度值，它是一个 8bit 的值，0~255 分别代表了不同的灰度级。从方程中可以看出，如果一个 alpha 值与一个色值相乘后，就会得到一个该色值的分量，这样两个窗口的叠加就靠不同的灰度来确定。换句话说，如果要使 win1~win4 窗体整体透明，就必须使 win1~win4 的灰度级最高，则使得 alpha value 为 0，并且色值方程配置 pn=0，qn=0，an=alphaA，bn=(1-alphaA)，其中 n={1,2,3,4}即可实现 window 0 显示，而其他窗口透明。

6.3.4　LCD 控制器寄存器

由于在 S5PC100 中 LCD 控制器的寄存器众多，这里只简单介绍一下我们要用到，并且很重要的寄存器，如果想了解更多的信息，请参看 S5PC100 手册。各寄存器及其参数含义如表 6.15 至表 6.24 所示。

表 6.15　VIDCON0，R/W,ADDRESS=0xEE000000

Field	Bit	Description	Reset Value
Reserved	[31]	Reserved (should be 0)	0
INTERLACE_F	[29]	逐行扫描方式或者间隔扫描方式 0 = 逐行扫描 1 = 间隔扫描方式(only ITU601/656 Interface)	0
VIDOUT	[27:26]	输出格式： 000: RGB I/F 001 = ITU601/656 010 = Indirect I80 I/F for LDI0 011 = Indirect I80 I/F for LDI1	000
PNRMODE	[18:17]	选择显示模式 (Where, VIDOUT[1:0] == 2'b00). 00 = RGB 格式 (RGB) 01 = RGB 格式 (BGR) 10 = Serial Format (R->G->B) 11 = Serial Format (B->G->R)	00
CLKVALUP	[16]	选择 CLKVAL_F 刷新时序的模式 0 = 总在刷新 1 =当一帧开始时	0
CLKVAL_F	[13:6]	决定 VCLK 的速率以及 CLKVAL[7:0] VCLK = HCLK / (CLKVAL+1) 且 CLKVAL >= 1 **Note.** 1. VCLK 最大值是 66MHz. 2. CLKSEL_F 寄存器选择的时钟源	0
VCLKFREE	[5]	VCLK 控制方式 0 =普通模式（ENVID） 1 = 自由模式	0
CLKDIR	[4]	选择时钟源的通道 0 = 直接获取时钟(VCLK = Clock source) 1 = 除以分频值	0
CLKSEL_F	[2]	选择时钟源 0 = HCLK 1 = SCLK_LCD	0
ENVID	[1]	数据输出使能位 0 = 禁止 1 = 使能	0
ENVID_F	[0]	当前帧结束使能位 0 = 禁止 1 = 使能 如果该位被设置，则该位直到一帧结束时才被禁止	0

表 6.16 **VIDCON1,R/W,ADDRESS=0XEE000004**

Field	Bit	Description	Reset Value
LINECNT (read only)	[26:16]	提供 line counter 的计数值(read only) 从 0 到 LINEVAL	0
FSTATUS	[15]	场状态 (read only). 0 = 非平坦场 1 = 平坦场	0
VSTATUS	[14:13]	垂直状态 read only). 00 = VSYNC 01 = 后肩 10 = ACTIVE 11 = 前肩	0
Reserved	[12:8]	Reserved	
IVCLK	[7]	VCLK 极性 0 = VCLK 下降沿取数据 1 = VCLK 上升沿取数据	0
IHSYNC	[6]	该位决定了 HSYNC 脉冲极性. 0 = 普通 1 = 反转	0
IVSYNC	[5]	该位决定了 VSYNC 脉冲极性 0 = 普通 1 = 反转	0
IVDEN	[4]	该位决定了 VDEN 信号极性 0 = 普通 1 = 反转	0
Reserved	[3:0]	Reserved	0x0

表 6.17 VIDTCON0,R/W,ADDRESS=0XEE000010

Field	Bit	Description	Reset
VBPD	[23:16]	垂直后肩所占周期	0x00
VFPD	[15:8]	垂直前肩所占周期	0x00
VSPW	[7:0]	垂直同步脉冲宽度	0x00

表 6.18 VIDTCON1,R/W,ADDRESS=0XEE000014

Field	Bit	Description	Reset Value
HBPD	[23:16]	水平后肩所占周期	0x00
HFPD	[15:8]	水平前肩所占周期	0x00
HSPW	[7:0]	水平同步脉冲宽度	0x00

表 6.19 VIDTCON2,R/W,ADDRESS=0XEE000018

Field	Bit	Description	Reset Value
LINEVAL	[21:11]	该位决定了显示屏的垂直尺寸	0
HOZVAL	[10:0]	该位决定了显示屏的水平尺寸	0

表 6.20 VIDTCON2,R/W,ADDRESS=0XEE000020

Field	Bit	Description	Reset Value
ENLOCAL	[22]	数据存取路径	0

		0 = DMA 方式 1 = 本地方式（CAMIF 0 ）	
BUFSTATUS	[21]	缓冲区的编号（这是因为 Windows 中 0、1 可以有两个缓冲区）（Read Only） 0 = 缓冲区 0 1 = 缓冲区 1	0
BUFSEL	[20]	选择缓冲区（0/1） 0 = 缓冲区 0 1 = 缓冲区 1	0
BUFAUTOEN	[19]	双缓冲自动控制位 0 = BUFSEL, 1 = 自动改变由 Trigger Input 控制	0
BITSWP	[18]	位交换控制位 0 = 禁止交换 1 = 使能交换	0
BYTSWP	[17]	字节交换控制位 0 = 禁止交换 1 = 使能交换	0
HAWSWP	[16]	半字交换控制位 0 = 禁止交换 1 = 使能交换	0
WSWP	[15]	字交换控制位 0 = 禁止交换 1 = 使能交换	0
Reserved	[14]	Reserved	0
InRGB	[13]	输入的源图像的格式 （Only for ‘EnLcal’ enable） 0 = RGB 1 = YCbCr	0
Reserved	[12:11]	Reserved (should be 00)	0
BURSTLEN	[10:9]	DMA’s Burst 最大长度 00 = 16 字 01 = 8 字 10 = 4 字	0

续表

Field	Bit	Description	Reset Value
Reserved	[8:7]	Reserved	0
BLD_PIX	[6]	选择绑定的类型 0 = 平面绑定 1= 像素绑定	0
BPPMODE_F	[5:2]	BPP（Bits Per Pixel）模式 0000 = 1 bpp 0001 = 2 bpp 0010 = 4 bpp 0011 = 8 bpp (palletized) 0100 = 8 bpp (无调色盘, A: 1-R:2-G:3-B:2) 0101 = 16 bpp (无调色盘, R:5-G:6-B:5) 0110 = 16 bpp (无调色盘, A:1-R:5-G:5-B:5) 0111 = 16 bpp (无调色盘, I :1-R:5-G:5-B:5) 1000 = unpacked 18 bpp (无调色盘, R:6-G:6-B:6) 1001 = unpacked 18 bpp (无调色盘, A:1-R:6-G:6-B:5) 1010 = unpacked 19 bpp (无调色盘, A:1-R:6-G:6-B:6) 1011 = unpacked 24 bpp (无调色盘, R:8-G:8-B:8)	

		1100 = unpacked 24 bpp（ 无调色盘,A:1-R:8-G:8-B:7 ） *1101 = unpacked 25 bpp (无调色盘, A:1-R:8-G:8-B:8) *1110 = unpacked 13 bpp（ 无调色盘,A:1-R:4-G:4-B:4) 1111 = unpacked 15 bpp（ 无调色盘, R:5-G:5-B:5)	
ALPHA_SEL	[1]	选择 Alpha 值的方式 平面绑定时： 0 = using ALPHA0_R/G/B values 1 = using ALPHA1_R/G/B values 像素绑定时： 0 = AEN 使能位置 （细节请参看 S5PC100 手册）	
ENWIN_F	[0]	0 = 窗口禁止 1 = 窗口使能	

表 6.21 FRAME BUFFER ADDRESS 0,R/W,ADDRESS=0XEE0000A0

Field	Bit	Description	Reset Value
VBANK_F	[31:24]	[31:24] 决定系统内存中的 bank 地址	0
VBASEU_F	[23:0]	[23:0] 决定 frame buffer 的起始地址	0

表 6.22 FRAME BUFFER ADDRESS 1,R/W,ADDRESS=0XEE0000D0

Field	Bit	Description	Reset Value
VBASEL_F	[23:0]	[23:0] 决定了 frame buffer 的结束地址	0x0

表 6.23 FRAME BUFFER ADDRESS 2 ,R/W,ADDRESS=0XEE000100

Field	Bit	Description	Reset Value
OFFSIZE_F	[25:13]	虚拟屏幕偏移（B）	0
PAGEWIDTH_F	[12:0]	虚拟页宽度 (B)	0

表 6.24 WINDOW 1 BLENDING EQUATION CONTROL REGISTER ,R/W,ADDRESS=0XEE000244

Field	Bit	Description	Reset Value
reserved	[31:22]	reserved	0x000
Q_FUNC	[21:18]	alphaB 值为: 0000 = 0 (zero) 0001 = 1 (max) 0010 = **alphaA (alpha value of *foreground) 0011 = 1 – alphaA 0100 = alphaB 0101 = 1 – alphaB 011x = reserved 100x = reserved 1010 = A (foreground color data) 1011 = 1 – A 1100 = B (background color data) 1101 = 1 – B 111x = reserved	0x0
reserved	[17:16]	Reserved	00

P_FUNC	[15:12]	Alpha 值为: 同上	0x0
reserved	[11:10]	Reserved	00
B_FUNC	[9:6]	B 的值为: 同上	0x3
reserved	[5:4]	Reserved	00
A_FUNC	[3:0]	A 的值为: 同上	0x2

6.4 驱动程序

本部分共有两个文件，一个是 led_drv.ko，这是驱动程序；另一个是 main，它是 main.c 生成的测试程序，可以通过 ioctl 来控制驱动程序，测试驱动程序是否达到目标。下面列出了编写驱动时候要用到的原理图，主要有 S5PC100 芯片手册和 FS_S5PC100A LED 电路图。

S5PC100 芯片手册片段如下：

- GPIO
  - Controls 173 external interrupts
  - Support 203 multi-functional input/output ports
  - GPA0: 8 in/out port – 2xUART with flow control
  - GPA1: 5 in/out port – 2xUART without flow control or 1xUART with flow control, 1x IrDA
  - GPB: 8 in/out port – 2x SPI
  - GPC: 5 in/out port – I2S, PCM, AC97
  - GPD: 7 in/out port – 2xI2C, PWM, External DMA request, SPDIF
  - GPE0,1: 14 in/out port – Camera I/F 0, MMC channel1(GPE0 support only 4-bit mode MMC, if 8-bit mode is needed, you can use GPE1 for another 4-bit data channel)
  - GPF0,1,2,3: 28 in/out port – LCD I/F
  - GPG0,1,2,3: 25 in/out port – 3xMMC channel(channel 0 support 4-bit and 8-bit mode, but channel 1, channel 2 support only 4-bit mode), SPI, I2S, PCM, SPDIF
  - GPH0,1,2,3: 32 in/out port – CAM IF channel, Key pad, External Wake-up(up-to 32-bit)
  - GPI: 8 in/out port – Booting option, SPW IF
  - GPJ0,1,2,3,4: 33 in/out port – Modem IF, HSI, ATA, LCD IF1
  - GPK0,1,2,3: 30 in/out port – SROM, NF, CF, OneNAND
  - GPL: 37 in/out memory port – EBI
  - MP1: 71 in/out memory port – DRAM

Register	Address	R/W	Description	Reset Value
			up/down Register	
GPG3CON	0xE030_01C0	R/W	Port Group GPG3 Configuration Register	0x00000000
GPG3DAT	0xE030_01C4	R/W	Port Group GPG3 Data Register	-
GPG3PUD	0xE030_01C8	R/W	Port Group GPG3 Pull-up/down Register	0x1555
GPG3DRV	0xE030_01CC	R/W	Port Group GPG3 Drive strength control Register	0x0000
GPG3PDNCON	0xE030_01D0	R/W	Port Group GPG3 Power down mode Configuration Register	0x00
GPG3PDNPULL	0xE030_01D4	R/W	Port Group GPG3 Power down mode Pull-up/down Register	0x00

Field	Bit	Description	Reset Value
GPG3CON[0]	[3:0]	0000 = Input, 0001 = Output, 0010 = SD_2_CLK, 0011 = SPI_2_CLK, 0100 = I2S2_SCLK, 0101 = PCM_0_SCLK, 1111 = NWU_INTG14[0]	0000
GPG3CON[1]	[7:4]	0000 = Input, 0001 = Output, 0010 = SD_2_CMD, 0011 = SPI_2_nSS, 0100 = I2S2_CDCLK, 0101 = PCM_0_EXTCLK, 1111 = NWU_INTG14[1]	0000
GPG3CON[2]	[11:8]	0000 = Input, 0001 = Output, 0010 = SD_2_D[0], 0011 = SPI_2_MISO, 0100 = I2S2_LRCK, 0101 = PCM_0_FSYNC, 1111 = NWU_INTG14[2]	0000
GPG3CON[3]	[15:12]	0000 = Input, 0001 = Output, 0010 = SD_2_D[1], 0011 = SPI_2_MOSI, 0100 = I2S2_SDI, 0101 = PCM_0_SIN, 1111 = NWU_INTG14[3]	0000

Field	Bit	Description	Reset Value
DAT[n] (n=0~7)	[n]	If the bit is configured as input, it represents the pin state. If the bit is configured as output, the pin state is the same as the value of the bit. If the port is configured as functional pin, an undefined value is read.	-

FS_S5PC100A LED 电路图如图 6.6 所示。

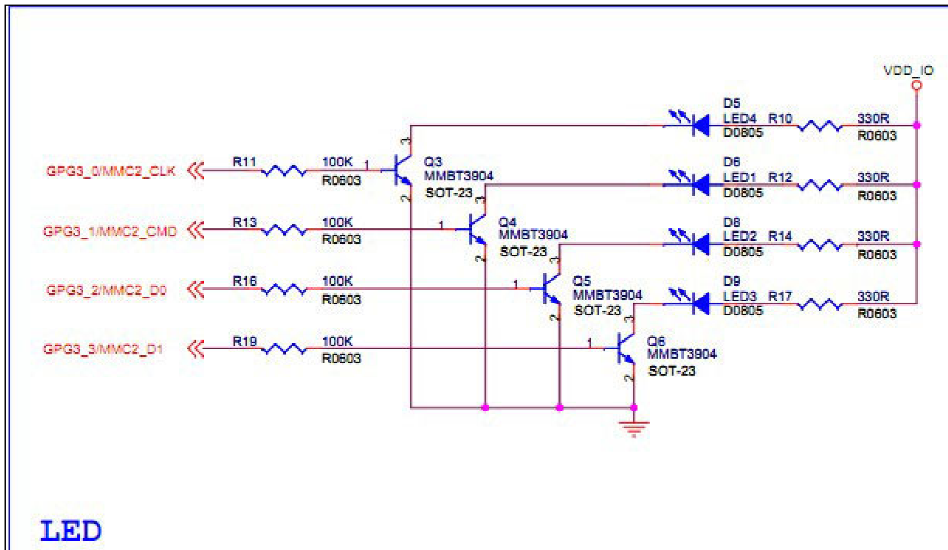

图 6.6　FS_S5PC100A LED 电路图

6.4.1　驱动程序初始化和退出

驱动程序初始化和退出的代码如下：

```
static int simple_major = 250;          //默认的设备号码，如果为 0 则尝试自动分配
……
/*
 * Set up the cdev structure for a device.
 */
static void simple_setup_cdev(struct cdev *dev, int minor,
      struct file_operations *fops)     //自编的函数，注册字符设备
{
 int err, devno = MKDEV(simple_major, minor);//建立设备号

 cdev_init(dev, fops);                  //初始化设备结构体 struct cdev *dev
 dev->owner = THIS_MODULE;
 dev->ops = fops;                       //关联 fops
 err = cdev_add (dev, devno, 1);        //注册一个字符设备
 /* Fail gracefully if need be */
 if (err)                               //注册失败处理
      printk (KERN_NOTICE "Error %d adding simple%d", err, minor);
}
/*
 * Our various sub-devices.
 */
/* Device 0 uses remap_pfn_range */
static struct file_operations simple_remap_ops = {  //定义设备的 fops
 .owner   = THIS_MODULE,
 .open    = simple_open,
 .release = simple_release,
 .read    = simple_read,
 .write   = simple_write,
 .ioctl   = simple_ioctl,
};

/*
 * We export two simple devices.  There's no need for us to maintain any
 * special housekeeping info, so we just deal with raw cdevs.
```

```
 */
static struct cdev SimpleDevs;

/*
 * Module housekeeping.
 */
static struct class *my_class;
static int simple_init(void)
{
  int result;
  dev_t dev = MKDEV(simple_major, 0);       //将设备号转化为 dev_t 的结构

  /* Figure out our device number. */
  if (simple_major)
       result = register_chrdev_region(dev, 1, "simple");//尝试申请主设备号
  else {
       //请求自动分配主设备号，起始值是 0，总共分配 1 个，设备名 simple
       result = alloc_chrdev_region(&dev, 0, 1, "simple");
       simple_major = MAJOR(dev);//将分配成功的设备号保存在 simple_major 变量中
  }
  if (result < 0) {//分配主设备号失败
       printk(KERN_WARNING "simple: unable to get major %d\n", simple_major);
       return result;
  }
  if (simple_major == 0)
       simple_major = result;

  /* Now set up two cdevs. */
  //调用自编的函数注册字符设备，有 Bug，没有返回注册是否成功。
  simple_setup_cdev(&SimpleDevs, 0, &simple_remap_ops);
  //Bug：打印前应该检查注册是否成功
  printk("simple device installed, with major %d\n", simple_major);

  //建立一个叫 simple 的内核 class，目的是下一步创建设备节点文件
  my_class= class_create(THIS_MODULE, "simple");
  device_create(my_class, NULL, MKDEV(simple_major, 0),
           NULL, "led");//创建设备节点文件
  return 0;
}


static void simple_cleanup(void)
{
  cdev_del(&SimpleDevs);  //删除字符设备
  unregister_chrdev_region(MKDEV(simple_major, 0), 1);//注销主设备号
  device_destroy(my_class,MKDEV(simple_major,0));//删除设备节点
  printk("simple device uninstalled\n");
}

module_init(simple_init);
module_exit(simple_cleanup);
```

6.4.2 驱动程序 Open and release 函数

驱动程序 Open and release 函数代码如下：

```
//寄存器地址，见 CPU 手册 70 页
#define pGPG3CON 0xE03001C0
#define pGPG3DAT 0xE03001C4
//寄存器操作指针
static void *vGPG3CON , *vGPG3DAT;
#define GPG3CON (*(volatile unsigned int *) vGPG3CON)
#define GPG3DAT (*(volatile unsigned int *) vGPG3DAT)
…
```

```
static int simple_major = 250;          //默认的主设备号
module_param(simple_major, int, 0);     //向内核申明一个参数，可以在 insmod 时传递给驱动程序
MODULE_AUTHOR("farsight");
MODULE_LICENSE("Dual BSD/GPL");

/*
 * Open the device; in fact, there's nothing to do here.
 */
int simple_open (struct inode *inode, struct file *filp)
{
 vGPG3CON=ioremap(pGPG3CON,0x10);//io remap 地址 pGPG3CON 到变量
 vGPG3DAT=vGPG3CON+0x04;      //计算 vGPG3DAT 寄存器的地址
 GPG3CON=0x1111;              //使用宏设定寄存器初始值
 GPG3DAT=0xff;                //使用宏设定寄存器初始值
 return 0;
}

ssize_t simple_read(struct file *file, char __user *buff, size_t count, loff_t *offp)
{
 return 0;
}

ssize_t simple_write(struct file *file, const char __user *buff, size_t count, loff_t *offp)
{
 return 0;
}

……

static int simple_release(struct inode *node, struct file *file)
{
 return 0;
}
```

6.4.3 驱动程序 ioctl 函数

驱动程序 ioctl 函数代码如下：

```
…
//ioctl 命令值，LED ON ，LED OFF
#define LED_ON 0x4800
#define LED_OFF 0x4801
…

void led_off( void )
{
 GPG3DAT=GPG3DAT|(1<<2);      //通过宏设定寄存器的值，置 1
 //printk("stop led\n");
}
void led_on( void )

{
 GPG3DAT=GPG3DAT&(~(1<<2));  //通过宏设定寄存器的值，置 0
 //printk("start led\n");
}

static int simple_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
 switch ( cmd )
 {
      case LED_ON:              //判断命令
           {
                led_on();      //执行命令
                break;
           }
```

```
        case LED_OFF:
            {
                led_off();
                break;
            }
        default:
            {
                break;
            }
    }
    return 0;
}
```

6.4.4 驱动测试程序 main.c

驱动测试程序 main.c 代码如下：

```
/*
 * main.c : test demo driver
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#define LED_ON 0x4800
#define LED_OFF 0x4801
int main()
{
 int i = 0;
 int dev_fd;
 dev_fd = open("/dev/simple",O_RDWR | O_NONBLOCK);//打开设备文件
 if ( dev_fd == -1 ) {
     printf("Cann't open file /dev/simple\n");
     exit(1);
 }
 while(1)
 {
 ioctl(dev_fd,LED_ON,0);      //发送 ioctl 命令 LED_ON
 //     printf("on\n");
 sleep(1);
 ioctl(dev_fd,LED_OFF,0);     //发送 ioctl 命令 LED_OFF
//     printf("off\n");
     sleep(1);
}
return 0;
}
```

6.5 小结

本章主要介绍了嵌入式 Android 内核设备驱动程序开发的基础。

首先介绍了设备驱动程序的基础知识、驱动程序与整个软硬件系统之间的关系，以及 Android 内核内核模块的基本编程。

接下来重点讲解了字符设备驱动程序的编写，这里详细介绍了字符设备驱动程序的编写流程、重要的数据结构、设备驱动程序的主要函数接口。然后又以 GPIO 驱动为例介绍了一个简单的字符驱动程序的编写步骤。最后，介绍了中断编程，并以编写完整的按键驱动程序为例进行讲解。

6.6 思考题

1．根据书中的提示，将本章所述的按键驱动程序进行进一步的改进，并在目标板上进行测试。
2．实现各种外设（包括 ADC、SPI、I^2C 等）的字符设备驱动程序。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067

华清远见