第 5 章　Linux 驱动程序

本章主要介绍 Linux 驱动程序工作原理、设备分类、设备文件接口和驱动程序加载方法，总结归纳字符设备驱动程序使用到的重要数据结构和常用函数，最后通过几个实例介绍字符设备驱动程序的设计以及编译、安装和测试等过程。

5.1　Linux 驱动程序概述

不同的硬件设备需要不同的驱动程序，如网卡、声卡、键盘和鼠标等，它们的驱动程序都不一样。如果某个硬件设备在操作系统中没有对应的驱动程序，系统就无法对该硬件设备进行操作，这时就需要为该硬件设备开发驱动程序。驱动程序是应用程序与硬件之间的一个中间软件层，它为应用程序屏蔽了硬件的细节。Linux 操作系统预留了安装驱动程序的接口，可以非常方便地将驱动程序加载到操作系统。

大多数驱动程序都是用来控制某一个硬件设备，但不一定是某一个物理性的硬件设备，如/dev/null/、dev/random，这些设备与真实的硬件没有什么联系，只是从内核获取数据再送往应用程序。

在嵌入式系统开发过程中，因为系统不同，硬件结构也有所不同，所以很少有通用的驱动程序可以使用。因此，驱动程序开发是整个嵌入式系统设计过程中必不可少的一部分。

5.1.1　驱动程序

驱动程序的目标是屏蔽具体物理设备的操作细节，实现设备无关性。在嵌入式操作系统中，驱动程序是内核的重要部分，它为内核提供了一组统一的 I/O 接口，用户可以使用这些接口实现对设备的操作。

1. 驱动程序的功能

驱动程序作为操作系统最基本的组成部分，它的功能通常包括以下 3 部分。

(1) 对设备初始化和释放。驱动程序加载时，完成设备注册、中断申请、初始化等操作；当驱动程序卸载时，则将使用的设备号、中断和内存空间等资源释放出来。

(2) 数据传送。驱动程序最重要的功能就是在内核、硬件和应用程序之间传送数据，从而实现对设备的具体操作。

(3) 检测和处理设备出现的错误。驱动程序应能够对设备出现的一些常见错误具有检测和纠错等功能。

2. 驱动程序的组成

驱动程序通常由以下 3 部分组成。

(1) 自动配置和初始化子程序。这部分负责检测所要驱动的硬件设备是否存在以及是否工作正常。如果设备工作正常，则对这个设备及其相关的设备驱动程序和需要的软件状

态进行初始化。这部分驱动程序仅在系统初始化的时候被调用一次。

(2) 服务于 I/O 请求的子程序,又称为驱动程序的上半部分。调用这部分程序是系统调用的结果。这部分程序在执行的时候,系统仍认为是和进行调用的进程属于同一个进程,只是由用户态变成了内核态,具有进行此系统调用的用户程序的运行环境,因此可以在其中调用 sleep 等与进程运行环境有关的函数。

(3) 中断服务子程序,又称为驱动程序的下半部分。在 Linux 系统中,并不是直接从中断向量表中调用设备驱动程序的中断服务子程序,而是由 Linux 系统来接收硬件中断,再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候,因此在中断服务子程序被调用的时候,不能依赖于任何进程的状态,也就不能调用任何与进程运行环境有关的函数。因为设备驱动程序一般支持同一类型的若干设备,所以一般在系统调用中断服务子程序的时候都带有一个或多个参数,以唯一标识请求服务的设备。

3. 驱动程序与应用程序的区别

驱动程序与应用程序的区别主要表现在以下 3 个方面。

(1) 应用程序一般有一个 main 函数,并从头到尾执行一个任务;驱动程序没有 main 函数,它在加载时,通过调用 module_init 宏完成驱动设备的初始化和注册工作之后便停止,并等待被应用程序调用。

(2) 应用程序可以和 GLIBC 库连接,因此可以包含标准的头文件;驱动程序不能使用标准的 C 库,因此不能调用任何 C 库函数,比如输出函数不能使用 printf,只能使用内核的 printk,包含的头文件只能是内核的头文件,比如 Linux/module.h。

(3) 驱动程序运行在内核空间(又称内核态),比应用程序执行的优先级要高很多。应用程序则运行在最低级别的用户空间(又称用户态),在这一级别禁止对硬件的直接访问和对内存的未授权访问。

5.1.2　设备分类

1. 设备的类型

Linux 系统通常将设备分为 3 类,即字符设备(Character Devices)、块设备(Block Devices)和网络设备(Network Devices)。应用程序对不同类型设备的操作有一些差别,如图 5.1 所示。应用程序通过字符设备文件(又称设备节点)来操作字符设备,通过块设备文

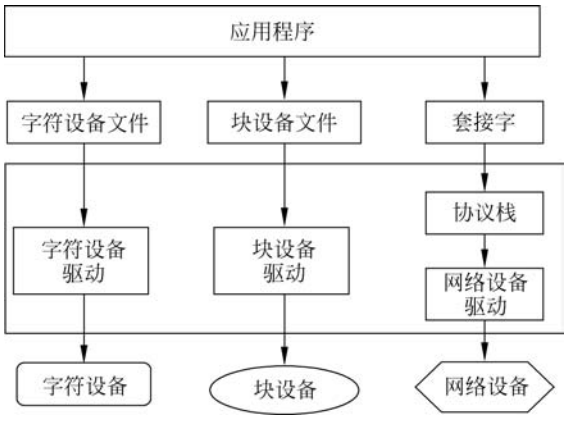


图 5.1　应用程序操作设备框图

件来操作块设备，通过套接字来操作网络设备。

1）字符设备

字符设备是指数据处理以字节为单位，并按顺序进行访问的设备，它一般没有缓冲区，不支持随机读写。嵌入式系统中的简单按键、触摸屏、鼠标、A/D 转换等设备都属于字符设备。

字符设备是 Linux 系统中最简单的设备，可以像文件一样被访问。初始化字符设备时，字符设备驱动程序向 Linux 系统登记，并在字符设备向量表 chrdevs 中增加一个 device_struct 数据结构条目，这个设备的主设备号用作这个向量表的索引，chrdevs[]数组的下标值就是主设备号，如图 5.2 所示。chrdevs 向量表中的每一个条目就是一个 device_struct 结构，这个结构的定义如下。

```
struct device_struct{
    const char  * name;                 //指向驱动程序名称
    struct file_operations    * fops;   //指向设备文件操作例程指针
}
```

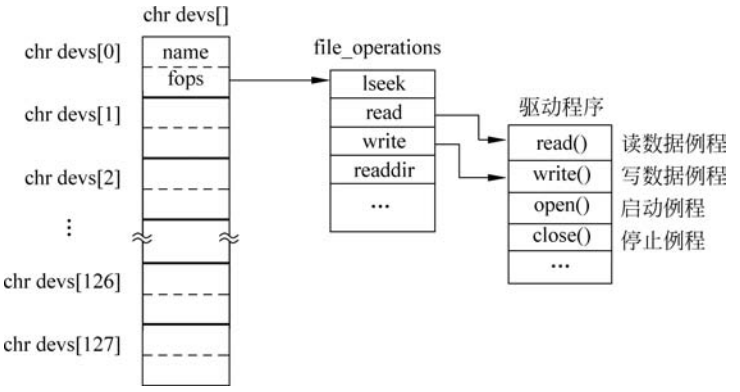


图 5.2　字符设备向量表

2）块设备

块设备是指在输入输出时以块为单位进行数据处理的设备，它一般都采用缓存技术，支持数据的随机读写。典型的块设备有硬盘、U 盘、内存、Flash、CD-ROM 等。

块设备是文件系统的物质基础，它也可以像文件一样被访问。Linux 系统用 blkdevs 向量表维护已经登记的块设备文件，它像 chrdevs 向量表一样使用设备的主设备号作为索引。blkdevs 向量表的条目也是 device_struct 结构，如图 5.3 所示。

块设备又分若干种类型，例如 SCSI 类和 IDE 类。块设备类向 Linux 内核登记并向内核提供文件操作，块设备类的驱动程序向这种类提供和类相关的接口。例如，SCSI 设备驱动程序必须向 SCSI 子系统提供接口，让 SCSI 子系统来对内核提供这种设备的文件操作。

对块设备文件进行读写时首先要对缓冲区进行操作，所以除了对文件进行操作的接口，块设备还必须提供缓冲区的接口。每一个块设备的驱动程序填充 blk_dev 向量表中的 blk_dev_struct 数据结构，这个向量表的索引还是设备的主设备号。blk_dev_struct 包含一个请

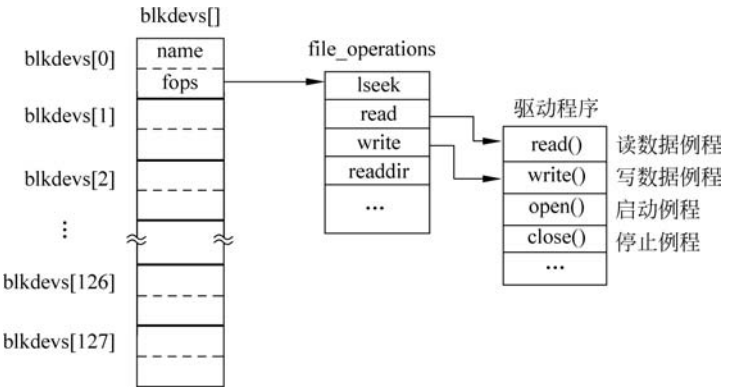


图 5.3　块设备向量表

求子程序和一个指向 request 结构的指针，每个 request 表示一个来自缓冲区的数据块读写请求。如果数据已存放在缓冲区内，则对缓冲区进行读写操作；否则系统将增加相应个数的 request 结构到对应的 blk_dev_struct 中，如图 5.4 所示。系统以中断方式调用 request 函数完成对块设备的读写，以响应请求队列，每个读写请求都有一个或多个 buffer_head 结构。对缓冲区进行读写操作时，系统可以锁定这个结构，这样会使得进程一直等待直到读写操作完毕。读写请求完成后，系统将相应的 buffer_head 从 request 中清除并解除锁定，等待进程被唤醒。

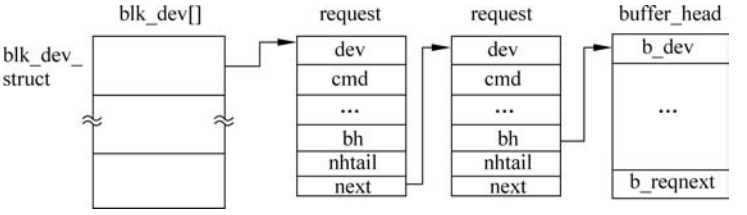


图 5.4　块设备缓冲区读写

3）网络设备

网络设备又称网络接口（Network Interface），用于网络通信。通常它们指的是硬件设备，但有时也可以是一个纯软件设备（如回环接口 loopback）。网络接口由内核中的网络子系统驱动，负责发送和接收数据包，而且它并不需要了解每一项事务如何映射到实际传送的数据包。因为它们的数据传送往往不是面向数据流（少数是，如 Telnet 和 FTP 就是面向数据流），所以不容易把它们映射到一个文件系统的节点上，所以 Linux 系统采用给网络接口设备分配一个唯一名字的方法来访问该设备。

2. 设备号

在传统的设备管理方式中，设备类型用于区分字符设备和块设备，字符设备用 C 表示，块设备用 B 表示。除了设备类型以外，内核还需要一对参数才能唯一标识某一设备，这对参数就是主设备号（Major Number）和次设备号（Minor Number）。

主设备号用于标识设备对应的驱动程序，主设备号相同的设备使用相同的设备驱动程序。一个主设备号可能有多个设备与之对应，这些设备在驱动程序内通过次设备号来进一

步区分。为了便于系统推广，Linux 操作系统对一些典型设备的主设备号进行了统一编号，例如，软驱的主设备号是 2，IDE 硬盘的主设备号是 3，并口的主设备号是 6。文件 include/linux/major.h 提供了当前正在使用的 Linux 系统中的全部主设备号清单。

次设备号是用来区分具体设备的实例(Instance)。例如，如果一台计算机上配有两个软驱，则两个软驱的主设备号都是 2，但次设备号不同，通常第一个软驱的次设备号为 0，第二个软驱的次设备号为 1。

在 Linux 2.4 版内核中有 256 个主设备号，更高版本的内核支持的主设备号更多。Linux 内核给设备分配主设备号的方法主要有两种，即静态申请和动态申请。静态申请是指由开发人员手工查看系统主设备号的使用情况，然后找到一个未被使用的主设备号，再向内核申请注册该主设备号。动态申请是指调用系统函数，向内核申请动态分配主设备号。

3. 设备文件

设备类型、主设备号、次设备号是内核与驱动程序通信时所使用的，但是对于开发应用程序的用户来说难以理解和记忆，所以 Linux 系统使用了设备文件的概念来统一对设备的访问接口。设备文件有时也称为设备节点，一般存放在/dev 目录下。正常情况下，/dev 目录下的每一个设备文件对应一个设备(包括虚拟设备)，设备文件的命名格式一般为“设备名＋数字或字母”，其中，数字或字母用于表示设备的子类，例如/dev/hda1、/dev/hda2 分别表示第一个 IDE 硬盘的第一个分区和第二个分区。

用 ls -l 命令可查看设备文件的属性，命令示例如下。

```
#ls -l /dev
...
crw-------    1  root    root    5, 64  Jan  1 00:00  cua0
crw-------    1  root    root    5, 65  Jan  1 00:00  cua1
crw-------    1  root    root    4, 64  Jan  1 00:11  ttyS0
crw-------    1  root    root    4, 65  Jan  1 00:00  ttyS1
...
#
```

从命令运行结果可以看出，很多设备的主设备号相同，但它们的次设备号却没有重复，体现了主、次设备号的分工。运行结果共有 8 列，各列属性的含义如表 5.1 所示。

表 5.1　ls -l 命令下显示的文件各列属性含义

列号	含　义
1	文件的类型及文件的权限。第 1 位表示文件的类型，c 表示该文件为字符设备文件，b 表示该文件为块设备文件；第 2～10 位表示文件的权限，r 表示读，w 表示写，x 表示执行
2	文件硬连接数
3	文件所属的用户
4	文件所属的用户组
5	主设备号
6	次设备号
7	文件最后修改的时间
8	文件名

Linux-2.4 版本内核中引入了设备文件系统，所有设备文件都可以作为一个可以挂装的文件，这样就可以由文件系统统一管理，设备文件就可以挂装到任何需要的地方；设备文件命名规则也发生了变化，一般将主设备建立一个目录，再将具体的子设备文件建立在此目录下。例如某目标机中的 MTD 设备文件保存在/dev/mtdblock 目录下，该目录下就有两个设备文件 0 和 1。

设备文件的创建方式有两种，即自动创建和手动创建。自动创建是驱动程序在加载时，驱动程序内部调用自动创建设备文件的函数，由内核完成设备文件的创建工作。手动创建是用户通过 mknod 命令来创建设备文件。mknod 的语法格式如下。

```
# mknod name type major minor
```

其中，参数 name 是设备文件名；type 是设备类型；major 是主设备号；minor 是次设备号。

实例：创建一个字符设备文件。要求设备名为/dev/demo，主设备号是 100，次设备号是 0。创建命令如下。

```
# mknod /dev/demo c 100 0
```

5.1.3　设备文件接口

Linux 应用程序可以通过设备文件的一组固定的入口点来访问驱动程序，这组入口点是由每个设备的设备驱动程序提供的，也就是设备文件接口。一般来说，字符设备和块设备的驱动程序能够提供给应用程序的常用设备文件接口包括 open 入口点、close 入口点、read 入口点、write 入口点、ioctl 入口点。

1. open 入口点

对字符设备文件进行操作，都需要通过设备的 open 入口点调用。open 子程序，功能是为将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则 open 子程序必须设置一些标志以表示设备处于忙碌状态。open 子程序的调用格式如下。

```
int open(char * filename, int access);
```

其中，参数 filename 是设备文件名；access 为文件描述字，它包含基本模式和修饰符两部分内容，两部分内容可用 & 或|连接。修饰符可以有多个，但基本模式只能有一个。access 的规定如表 5.2 所示。

表 5.2　access 的规定

基本模式	含　义	修饰符	含　义
O_RDONLY	只读	O_APPEND	文件指针指向末尾
O_WRONLY	只写	O_CREAT	无文件时创建文件，属性按基本模式属性
O_RDWR	读写		
O_BINAR	打开二进制文件	O_TRUNC	若文件存在，将其长度缩为 0，属性不变
O_TEXT	打开文本文件		

如果 open 函数打开成功，则返回值就是文件描述字的值(非负值)，否则返回－1。

2. close 入口点

当设备操作结束后，需要通过 close 入口点调用 close 子程序关闭设备。如果是独占设备，则必须标记设备可再次使用。close 子程序的调用格式如下。

```
int close(int handle);
```

其中，参数 handle 为文件描述字(或称为设备文件句柄)。

3. read 入口点

当从设备上读取数据时，需要通过 read 入口点调用 read 子程序。read 子程序的调用格式如下。

```
int read(int handle, void * buf, int count);
```

其中，参数 handle 为文件描述字；buf 为存放数据的缓冲区；count 为读取数据的字节数。

若 read 函数返回值等于 count 参数值，则表示请求的数据传输成功；若返回值大于 0，但小于 count 参数值，则表明部分数据传输成功；若返回值等于 0，则表示到达文件的末尾；若返回值为负数，则表示出现错误，并会以错误号指明是何种错误，错误号的定义参见 <linux/errno.h>；在阻塞型 I/O 中，调用 read 子程序会出现阻塞。

4. write 入口点

向设备上写数据时，需要通过 write 入口点调用 write 子程序。write 子程序的调用格式如下。

```
int write(int handle, void * buf, int count);
```

其中，参数 handle 为文件描述字；buf 为存放待写数据的缓冲区；count 为向设备写数据的字节数。

若 write 函数返回值等于 count 参数值，则表示请求的数据传输成功；若返回值大于 0，但小于 count 参数值，则表明部分数据传输成功；若返回值等于 0，则表示没有写任何数据；若返回值为负数，表示出现错误，并且会以错误号指明是何种错误；在阻塞型 I/O 中，调用 write 子程序的会出现阻塞。

5. ioctl 入口点

ioctl 入口点主要用于对设备进行读写之外的其他操作，比如配置设备、进入或退出某种操作模式等，这些操作一般无法通过调用 read 或 write 子程序完成操作。比如：目标机中的 SPI 设备通道的选择操作，只有通过调用 ioctl 子程序操作才可以完成，调用格式如下。

```
int ioctl(int handle, int cmd, ...);
```

其中，…代表可变数目的参数表，如果只有一个可选参数，可以将其定义如下格式。

```
int ioctl(int handle, int cmd, char * argp);
```

其中，参数 handle 是文件描述符；cmd 是直接传递给驱动程序的一个值；可选参数 argp 用于定义无论用户应用程序使用的是指针还是其他类型值，都以 unsigned long 的形式传递给驱动程序。

6. 应用案例

下面通过一个实例来介绍字符设备文件接口的使用方法。

【程序 5.1】 编写应用程序,实现向串口发送字符串“ATD2109992”。

假设不需要对串口属性进行设置,则具体程序如下。

```
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#define MAX 20
int main()
{
    int fd,n;
    char buf[MAX]="ATD2109992";
    fd=open("/dev/ttyS0",O_RDWR); //open 入口点,ttyS0 是设备文件
    if( fd<0)
    {
        perror("Unable open /dev/ttyS0\n ");
        return 1;
    }
    n = write(fd,buf,strlen(buf));   //write 入口点
    if ( n<0 )
        printf( "write() of %d bytes failed!\n",strlen(buf));
    else
        printf( "write() of %d bytes ok!\n",strlen(buf));
    close(fd);                                  //close 入口点
}
```

5.1.4 驱动程序加载方法

1. 连接到内核

Linux 设备驱动程序属于内核的一部分,驱动程序连接到内核的方法有两种,即静态连接和动态连接。

静态连接是指将驱动程序源码保存在内核源码指定的位置,并修改内核相关参数,让它成为内核源码的一部分,然后重新编译内核,这时驱动程序将被编译到内核映像文件之中。

动态连接是指将驱动程序作为一个模块(module)单独编译,在需要它的时候再动态加载到内核中,如果不需要它,则可以将它从内核中删除。

在嵌入式系统开发阶段,为了方便调试,驱动程序一般采用动态连接的方法连接到内核;在产品发布阶段,驱动程序一般采用静态连接的方法连接到内核。

2. 模块化编程

设备驱动模块化编程一般分为加载、系统调用和卸载 3 个过程,如图 5.5 所示。

1) 加载

当执行 insmod 命令加载驱动程序时,首先调用驱动程序中的入口函数 module_init,该函数完成设备驱动的初始化工作,比如寄存器置位、中断申请、数据结构初始化等一系列工作。另外还有一个最要的工作就是向内核注册该设备,如果是字符设备,可以调用 register_chrdev 函数完成注册;如果是块设备,可以调用 register_blkdev 函数完成注册。注册成功后,该设备即获得了系统分配的主设备号、自定义的次设备号,并建立了与文件系统的关联。

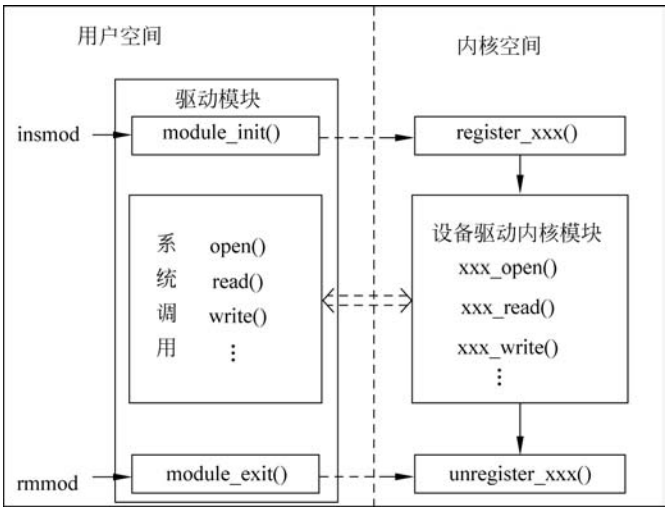


图 5.5　设备驱动加载、系统调用和卸载过程

2）系统调用

当驱动程序加载完成后，就一直等待应用程序来调用。应用程序可以利用设备文件对其进行操作，如调用 open、read、write、ioctl 和 close 等函数。

3）卸载

当执行 rmmod 命令卸载驱动程序时，会调用驱动程序中的 module_exit 函数，完成相应资源的回收，比如令设备的响应寄存器值复位并从系统中注销该设备。字符设备可调用 unregister_chrdev 函数完成注销，块设备可调用 unregister_blkdev 函数完成注销。

3. 驱动程序操作命令

1）lsmod 命令

功能：显示当前已加载的模块。

语法：

```
lsmod
```

【例 5.1】 查看系统当前已加载的模块。

```
#lsmod
Module          Size        Used by
ov511           67140        0 (unused)
videodev        5824         0 [ov511]
motor           1608         0 (unused)
ad              1712         0 (unused)
#
```

说明：从列表中可以看到系统已加载了 ov511、videodev、motor 和 ad 共 4 个驱动程序模块。

2）insmod 命令

功能：将驱动模块加载到操作系统内核。

语法：

```
insmod file_name
```

【例 5.2】 将数码管驱动程序(tube.ko)加载到内核。

```
#insmod tube.ko
Using tube.ko
Warning: loading tube will taintsthe kernel: no 3icense
  See hctp://www.tux.or2/lkml/#export-t4inted for information about tainted modules
0-numeric tube : Dprintk   device open
s3c2410-hc595 initialized
#
```

说明：以上显示是加载过程中的提示信息。

用户可以利用 lsmod 查看加载是否成功。

```
#lsmod
Module          Size       Used by
tube            2072       0 (unused)
ov511           67140      0 (unused)
videodev        5824       0 [ov511]
motor           1608       0 (unused)
ad              1712       0 (unused)
#
```

从以上列表中可以看到 tube 驱动模块已经加载到了内核。

3) rmmod 命令

功能：将驱动模块从内核中删除。

语法：

```
rmmod module_name
```

【例 5.3】 请将内核中的 tube 模块删除。

```
#rmmod tube
s3c2410-hc595 unloaded
#
```

说明：以上显示是删除过程中的提示信息。

5.1.5　设备驱动程序的重要数据结构

设备驱动程序所提供的入口点主要由 3 个重要的数据结构向系统进行说明，这 3 个数据结构分别是 file_operations、file 和 inode。

1. file_operations

内核内部通过 file 结构识别设备，通过 file_operations 数据结构提供文件系统的入口点函数，也就是访问设备驱动程序的函数。file_operations 定义在< linux/fs.h >头文件中，定义如下。

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

```
    ssize_t ( * write) (struct file * ,const char __user * ,size_t,loff_t * );
    ssize_t ( * aio_read) (struct kiocb * ,const struct iovec * ,unsigned long,loff_t);
    ssize_t ( * aio_write) (struct kiocb * ,const struct iovec * ,unsigned long,loff_t);
    int ( * iterate) (struct file * ,struct dir_context * );
    unsigned int ( * poll) (struct file * ,struct poll_table_struct * );
    long ( * unlocked_ioctl) (struct file * ,unsigned int,unsigned long);
    long ( * compat_ioctl) (struct file * ,unsigned int,unsigned long);
    int ( * mmap) (struct file * ,struct vm_area_struct * );
    int ( * open) (struct inode * ,struct file * );
    int ( * flush) (struct file * ,fl_owner_t id);
    int ( * release) (struct inode * ,struct file * );
    int ( * fsync) (struct file * ,loff_t,loff_t,int datasync);
    int ( * aio_fsync) (struct kiocb * ,int datasync);
    int ( * fasync) (int,struct file * ,int);
    int ( * lock) (struct file * ,int,struct file_lock * );
    ssize_t ( * sendpage) (struct file * ,struct page * ,int,size_t,loff_t * ,int);
    unsigned long ( * get_unmapped_area)(struct file * ,unsigned long,unsigned long,unsigned
long,unsigned long);
    int ( * check_flags)(int);
    int ( * flock) (struct file * ,int,struct file_lock * );
    ssize_t ( * splice_write)(struct pipe_inode_info * ,struct file * ,loff_t * ,size_t,unsigned int);
    ssize_t ( * splice_read)(struct file * ,loff_t * ,struct pipe_inode_info * ,size_t,unsigned int);
    int ( * setlease)(struct file * ,long,struct file_lock * * );
    long ( * fallocate)(struct file * file,int mode,loff_t offset,loff_t len);
    int ( * show_fdinfo)(struct seq_file * m,struct file * f);
};
```

file_operations 数据结构是整个 Linux 内核的重要数据结构，它也是 file、inode 结构的重要成员，其中的主要成员说明如表 5.3 所示。

表 5.3　file_operations 数据结构的主要成员

成　　员	功　　能
owner	模块的拥有者
llseek	重新定位读写位置
read	从设备中读取数据
write	向设备中写入数据
unlocked_ioctl	控制设备，除读写操作外的其他控制命令
mmap	将设备内存映射到进程地址空间，通常只用于块设备
open	打开并初始化设备
flush	清除内容，一般只用于网络文件系统中
release	关闭设备并释放资源
fsync	实现内存与设备的同步，如将内存数据写入硬盘
fasync	实现内存与设备之间的异步通信
lock	文件锁定，用于文件共享时的互斥访问

目前，系统开发中对此结构体采用“标记化”的方法进行赋值，主要是应对由此结构日益庞大而带来的赋值冗余问题，即可能只使用其中的部分成员。例如在嵌入式系统开发中，一般只需实现其中几个接口函数，如 read、write、unlocked_ioctl、open、release 等，就可以完成

应用系统需要的功能，没有必要采用传统的方法对其所有变量全部进行赋值。例如对此结构体变量 fs4412_fops 用“标记化”的方法进行赋值，程序如下。

```
static struct file_operations fs4412_fops={
    .owner: THIS_MODULE,
    .open: fs4412_ts_open,
    .read: fs4412_ts_read,
    .write: fs4412_ts_write,
    . unlocked_ioctl: fs4412_ts_ unlocked_ioctl,
    .release: fs4412_ts_release,
};
```

可以看出，file_operations 就是一个函数指针数组，它们对应于一些系统调用，起名为 open、read、write、ioctl、close 等，后面的 fs4412_ts_open、fs4412_ts_read、fs4412_ts_write、fs4412_ts_unlocked_ioctl、fs4412_ts_release 等才是真正与硬件设备完成交互的函数。当应用程序中执行 open 函数时，系统会通过 file_operations 结构将其映射到 fs4412_ts_open 函数。

2. file

file 数据结构主要用于与文件系统对应的驱动程序。file 数据结构代表一个打开的文件，系统中每个打开的文件在内核空间都有一个关联的 file。file 数据结构是由内核在打开(open)文件时创建的，并且在关闭文件之前作为参数传递给操作在设备上的函数。在文件关闭后，内核将释放这个数据结构。

file 数据结构是由系统默认生成的，驱动程序从不去填写它，只是简单地访问。在内核源代码中，指向 struct file 的指针通常称为 file 或 filp(文件指针)。file 数据结构定义如下。

```
struct file {
    union {
        struct llist_node      fu_llist;        //文件对象链表
        struct rcu_head        fu_rcuhead;      //释放之后的 RCU 链表
    } f_u;
    struct path            f_path;              //描述文件路径的结构体
#define f_dentry      f_path.dentry
    struct inode           * f_inode;           //缓存值
    const struct file_operations      * f_op;   //执行文件操作的指针
    spinlock_t             f_lock;              //自旋锁
    atomic_long_t          f_count;             //使用该结构的进程数
    unsigned int           f_flags;             //文件标志,阻塞/非阻塞型操作时检查
    fmode_t                f_mode;              //标识文件的读写权限
    struct mutex           f_pos_lock;          //互斥锁结构体
    loff_t                 f_pos;               //文件当前位置
    struct fown_struct     f_owner;             //文件拥有者
    const struct cred      * f_cred;            //文件的信任状
    struct file_ra_state   f_ra;                //预读状态

    u64                    f_version;           //版本号
#ifdef CONFIG_SECURITY
    void                   * f_security;        //安全模块
#endif
```

```
    void                * private_data;          //私有数据

#ifdef CONFIG_EPOLL
    struct list_head    f_ep_links;             //事件池链表
    struct list_head    f_tfile_llink;
#endif
    struct address_space    * f_mapping;        //页缓存映射
#ifdef CONFIG_DEBUG_WRITECOUNT
    unsigned long f_mnt_write_state;
#endif
} __attribute__((aligned(4)));
```

3. inode

inode 数据结构用来记录文件的物理上的信息。一个文件可以对应多个 file 数据结构，但只能对应一个 inode 数据结构。inode 数据结构定义如下。

```
struct inode {
    umode_t             i_mode;                 //文件类型
    unsigned short      i_opflags;
    kuid_t              i_uid;                  //文件拥有者的标识号
    kgid_t              i_gid;                  //文件拥有者所在组的标识号
    unsigned int        i_flags;                //文件系统的安装标志

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl    * i_acl;
    struct posix_acl    * i_default_acl;
#endif

    const struct inode_operations    * i_op;    //索引节点操作
    struct super_block    * i_sb;               //指向该文件系统超级块的指针
    struct address_space    * i_mapping;        //把所有可交换的页面管理起来

#ifdef CONFIG_SECURITY
    void                * i_security;
#endif

    unsigned long           i_ino;              //节点号
    union {
        const unsigned int i_nlink;             //硬链接个数
        unsigned int __i_nlink;
    };
    dev_t               i_rdev;                 //实际设备号,编写驱动程序时需要
    loff_t              i_size;                 //文件大小
    struct timespec         i_atime;            //文件最后访问的时间(类型有变化)
    struct timespec         i_mtime;            //文件最后修改的时间
    struct timespec         i_ctime;            //节点最后修改的时间
    spinlock_t          i_lock;                 //i_blocks,i_bytes,maybe i_size
    unsigned short              i_bytes;        //文件中位于最后一个块的字节数
    unsigned int          i_blkbits;            //块的位数
    blkcnt_t            i_blocks;               //文件所占用的块数
```

```
#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t          i_size_seqcount;                //对 i_size 进行串行计数
#endif

    /* Misc */
    unsigned long           i_state;                    //inode 的状态
    struct mutex            i_mutex;                    //保护 inode 的互斥锁

    unsigned long           dirtied_when;               //inode 第一次为脏的时间

    struct hlist_node   i_hash;                         //指向哈希链表的指针
    struct list_head    i_wb_list;                      //备用设备 I/O 链表
    struct list_head    i_lru;                          //inode LRU 链表
    struct list_head    i_sb_list;                      //超级块链表
    union {
        struct hlist_head       i_dentry;               //所有引用该 inode 的目录项形成的链表
        struct rcu_head         i_rcu;
    };
    u64                 i_version;                      //版本号
    atomic_t            i_count;                        //引用计数
    atomic_t            i_dio_count;
    atomic_t            i_writecount;                   //写进程的引用计数
    const struct file_operations        *i_fop;         //former ->i_op->default_file_ops
    struct file_lock        *i_flock;                   //指向文件加锁链表的指针
    struct address_space        i_data;
#ifdef CONFIG_QUOTA
    struct dquot            *i_dquot[MAXQUOTAS];        //inode 磁盘限额
#endif
    struct list_head    i_devices;
    union {
        struct pipe_inode_info      *i_pipe;            //如果文件是一个管道则使用它
        struct block_device     *i_bdev;                //如果文件是块设备则使用它
        struct cdev             *i_cdev;                //如果文件是字符设备则使用它
    };

    __u32               i_generation;

#ifdef CONFIG_FSNOTIFY
    __u32               i_fsnotify_mask;                //目录通知事件掩码
    struct hlist_head       i_fsnotify_marks;
#endif

#ifdef CONFIG_IMA
    atomic_t            i_readcount;                    //打开 R0 的结构文件
#endif
    void                *i_private;                     //文件或设备的私有信息
};
```

inode 数据结构包含大量关于文件的信息。作为一个通用的规则，这个结构只有两个成员对编写驱动程序非常重要，一个是 dev_t i_rdev，对应于代表设备文件的节点，这个成员包含实际的设备号；另一个是 struct cdev * i_cdev，它是内核的内部结构，代表字符设备，当节点指的是一个字符设备文件时，这个成员包含一个指向这个数据结构的指针。

5.1.6　驱动程序常用函数

在 Linux 系统中，驱动程序能够使用的库函数很多，常用的函数说明如下。

1. 字符设备注册及注销函数

字符设备驱动程序可通过 register_chrdev_region 函数向内核注册设备，又可通过 unregister_chrdev_region 函数从内核注销设备。这两个函数存放在 linux/fs.h 文件中，它们的定义如下。

```
int register_chrdev_region(dev_t first, unsigned int count, char * name);
void unregister_chrdev_region(dev_t from, unsigned int count);
```

其中，参数 first 是分配的起始设备号，次设备号部分常常是 0，但是没有严格要求；count 是请求连续设备号的总数。注意，如果 count 太大，可能溢出到下一个主设备号，但是只要要求的编号范围可用，都仍然会正常工作。name 是设备的名称。unregister_chrdev_region 函数中的 from 参数与 first 参数意义相同。

若 register_chrdev_region 函数返回值为 0，则表示函数执行成功；返回值为－INVAL，则表示申请的主设备号非法；返回值为－EBUSY，则表示申请的主设备号正在被其他驱动程序使用。如果动态分配主设备号成功，则此函数返回值就是所分配的主设备号。如果 register_chrdev_region 函数操作成功，则设备名就会出现在/proc/devices 文件中。

2. 中断申请和释放函数

驱动程序可通过 request_irq 函数向内核申请中断，又可通过 free_irq 函数释放中断。它们的定义如下。

```
int request_irq(unsigned int irq,
                void ( * handler)(int, void * , struct pt_regs * ),
                unsigned long flags,
                const char * dev_name,
                void * dev_id);
void free_irq(unsigned int irq, void * dev_id);
```

其中，参数 irq 表示所要申请的中断号；handler 为向系统登记的中断处理子程序，中断产生时由系统来调用；flags 是申请中断时的选项，它决定中断处理程序的一些特性，设置 SA_INTERRUPT，表示中断处理程序是快速处理程序，设置 SA_SHIRQ，表示中断可以在设备之间共享；dev_name 为设备名，申请中断成功后会出现在/proc/interrupts 文件里；dev_id 为申请中断时告诉系统的设备标识。

若 request_irq 函数返回值为 0，则表示函数执行成功；返回值为－INVAL，则表示 irq＞15 或 handler＝＝NULL；返回值为－EBUSY，则表示中断已被占用且不能共享。

一般应该在设备第一次打开时使用 request_irq 函数，在设备最后一次关闭时使用 free_irq。

编写中断处理函数时需要注意，中断处理程序与普通 C 语言程序没有太大不同，不同的是中断处理程序在中断期间运行，不能向用户空间发送或接收数据，不能执行有睡眠操作的函数，不能调用调度函数。

3. 阻塞型 I/O 操作函数

当对设备进行读写操作时，如果驱动程序无法立刻满足请求，应当如何响应呢？驱动程序应当（默认）阻塞进程，使它进入睡眠直到请求可继续，即阻塞型 I/O 操作。

调用以下函数可以让进程进入睡眠状态。

```
void sleep_on(struct wait_queue * * q);
void interruptible_sleep_on(struct wait_queue * * q);
void wait_event_interruptible(struct wait_queue wq,condition);
```

调用以下函数可以唤醒进程。

```
void wake_up(struct wait_queue * * q);
void wake_up_interruptible(struct wait_queue * * q);
```

sleep_on 和 interruptible_sleep_on 的区别是 sleep_on 不能被信号取消，但是 interruptible_sleep_on 可以，也就是说，前者适用于不可中断进程，后者都适用于可中断进程。wake_up 和 wake_up_interruptible 的区别类似。

4. 并发处理函数

在编写驱动程序时，还需要考虑进程并发处理的情况。当一个进程请求内核驱动程序模块服务时，如果内核模块正忙，则可以先将进程放入睡眠状态直到驱动程序模块空闲。

调用以下函数可以完成并发处理。

```
void up(struct semaphore * sem);
void down(struct semaphore * sem);
int down_interruptible(struct semaphore * sem);
```

其中，sem 是信号，down_interruptible 是可以中断的，如果操作被中断，该函数会返回非零值，而调用者不会拥有该信号量，使用时需要始终检查返回值，并做出相应的响应。

5. 内核空间和用户空间的数据传递函数

Linux 系统运行在两种模式下，即内核模式和用户模式，又分别称为内核态和用户态。内核模式对应于内核空间，用户模式对应于用户空间。驱动程序运行于内核空间，应用程序运行在用户空间。这两个空间互相之间不能直接访问的数据，必须利用 copy_to_user 函数将内核空间的数据传递给用户空间；利用 copy_from_user 函数把用户空间的数据传递给内核空间。它们的定义如下。

```
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);
```

其中，参数 to 指向传递的目标地址；from 指向传递的起始地址；count 是传递的数据长度，函数的返回值为实际传递数据的长度。

6. 设备文件自动创建函数

通过调用 devfs_register 函数可完成设备的注册以及设备文件的自动创建，函数的定义如下。

```
devfs_register(devfs_handle_t dir, const char * name, unsigned int flags, unsigned int major, unsigned
int minor, umode_t mode, void * ops, void * info);
```

其中，参数 dir 是新创建的设备文件的父目录，如果为 NULL，则表示父目录为/dev；name 是新建的设备文件的名称；flags 是标志的位掩码；major 是为设备驱动程序向内核申请的主设备号；minor 是次设备号；mode 是设备的访问模式；ops 是设备文件的操作数据结构指针。

7. I/O 映射函数

通过调用 ioremap 函数可完成 I/O 映射，即将一个 I/O 地址空间映射到内核的虚拟地址空间上去，以便于访问。通过调用 iounmap 函数可以释放映射。两函数的定义如下。

```
void * __ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags);
void * __iounmap(unsigned long phys_addr);
```

其中，参数 phys_addr 是映射的起始 I/O 地址；size 是映射的空间的大小；flags 是映射的 I/O 空间与权限有关的标志。

8. I/O 读写操作函数

通过调用 readl 函数可完成 I/O 映射读操作，调用 writel 函数可完成 I/O 映射写操作。两函数的定义如下。

```
unsigned char readl (unsigned int addr );
void writel (unsigned char data, unsigned short addr );
```

其中，addr 是 I/O 地址；data 是要写入的数据。

5.2　虚拟字符设备 Demo 驱动程序设计

5.2.1　驱动程序编写方法

驱动程序一般是针对某一种具体硬件来编写的，所以编写驱动程序之前，必须对硬件的工作原理和流程了解得非常清楚，另外还要选择一种合适的驱动程序编写方法。驱动程序编写方法有 3 种，即传统方法、平台总线和设备树。这 3 种方法的核心都是一样的，即完成分配硬件资源、设置硬件工作环境以及注册 file_operations 结构体等工作，同时它们又有各自的特点。

(1) 传统方法。在驱动程序代码中写死硬件资源，代码简单，但不易扩展。

(2) 平台总线。把驱动程序分为两部分，即平台总线设备(Platform_device)和平台总线驱动(Platform_driver)。在平台总线设备中指定硬件资源，在平台总线驱动中完成分配硬件资源、设置硬件工作环境以及注册 file_operations 结构体等工作。这种方法易于扩展，

但是有很多冗余代码，如果硬件有变动，则需重新编译内核或驱动程序。

(3) 设备树。驱动程序也分为两部分，即设备树(Device Tree)和平台总线驱动。在设备树中指定硬件资源。每次系统启动时，设备树中的内容都会自动传给内核，当加载平台总线驱动时，内核会自动将平台总线驱动与设备树中的硬件资源进行匹配，如果匹配成功，则启动驱动程序，否则中止驱动程序加载。设备树方法的优点是易于扩展，没有冗余代码，当硬件有变动时，不需要重新编译内核或驱动程序，只需要修改设备树。

本节将介绍一个与硬件无关的虚拟字符设备驱动程序的设计示例，采用传统方法来编写，主要目的是帮助读者了解传统方法编写驱动程序的基本流程。

5.2.2　Demo 驱动程序设计

假设有一个简单的虚拟字符设备 Demo，该设备只在内核空间开辟一个大小为 40B 的缓冲区(drv_buf)。要求为该设备设计一个驱动程序，使该设备能够为应用程序提供读、写两种操作。

1. 程序功能

Demo 驱动程序的功能说明如下。

(1) 将驱动程序编译成模块，以模块方式动态加载。

(2) 模块加载时完成设备的注册，设备名为 demo，主设备号为 249，次设备号为 0。

(3) 打开和关闭设备时只显示提示信息。

(4) 读操作时将内核缓冲区中的数据读出。

(5) 写操作时将数据写入内核缓冲区。

(6) 模块卸载时完成设备的注销。

2. 程序设计

Demo 驱动程序的结构如图 5.6 所示。

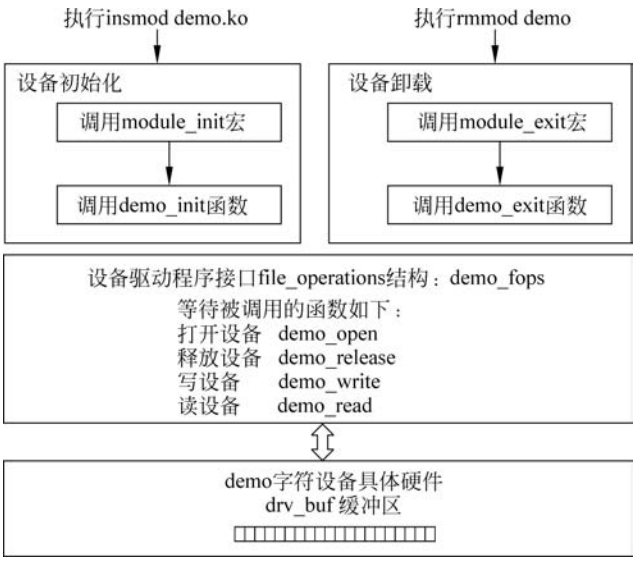


图 5.6　Demo 驱动程序结构示意图

【程序 5.2】 Demo 驱动程序 demo.c。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>

#include <asm/io.h>
#include <asm/uaccess.h>

MODULE_LICENSE("Dual BSD/GPL");

#define DEMO_MA 249          //主设备号
#define DEMO_MI 0            //次设备号
#define DEMO_NUM 1           //设备数量
static int MAX_BUF_LEN=40;  //缓冲区大小
static char drv_buf[40]="This is the initialization string";
struct cdev cdev;
/*****************************************/
static int demo_open(struct inode *inode,struct file *file)
{
    printk("This is demo_open function\n");
    return 0;
}
/*****************************************/
static int demo_release(struct inode *inode,struct file *file)
{
    printk("This is demo_release function\n");
    return 0;
}
/*****************************************/
static ssize_t demo_read(struct file *file,char *buffer,size_t count,loff_t *loff)
{
    if(count > MAX_BUF_LEN)
        count=MAX_BUF_LEN;
    /*将数据从内核态复制到用户态*/
    copy_to_user(buffer,drv_buf,count);
    printk("This is demo_read function\n");
    return count;
}
/*****************************************/
static ssize_t  demo_write(struct file *file,const char *buffer,size_t count)
{
    if(count > MAX_BUF_LEN)
        count=MAX_BUF_LEN;
    /*将数据从用户态复制到内核态*/
    copy_from_user(drv_buf ,buffer,count);
    printk("This is demo_write function\n");
    return count;
```

```
}
/*****************************************/
struct file_operations demo_fops = {
    .owner = THIS_MODULE,
    .open = demo_open,
    .release = demo_release,
    .read = demo_read,
    .write = demo_write,
};
/*****************************************/
static int demo_init(void)
{
    dev_t devno = MKDEV(DEMO_MA,DEMO_MI);
    int ret;
    /*注册字符设备*/
    ret = register_chrdev_region(devno,DEMO_NUM,"demo");
    if (ret<0) {
        printk("register_chrdev_region\n");
        return ret;
    }
    /*初始化 cdev 结构变量*/
    cdev_init(&cdev,&demo_fops);
    cdev.owner = THIS_MODULE;
    /*将字符设备信息添加到 cdev 结构变量*/
    ret = cdev_add(&cdev,devno,DEMO_NUM);
    if (ret<0) {
        printk("cdev_add\n");
        unregister_chrdev_region(devno,DEMO_NUM);
        return ret;
    }
    printk("Demo driver Init OK\n");
}
/*****************************************/
static void demo_exit(void)
{
    dev_t devno = MKDEV(DEMO_MA,DEMO_MI);
    /*将字符设备信息从 cdev 结构变量中删除*/
    cdev_del(&cdev);
    /*注销字符设备*/
    unregister_chrdev_region(devno,DEMO_NUM);
    printk("Demo driver exit\n");
}
/*****************************************/
module_init(demo_init);                    //安装驱动程序入口
module_exit(demo_exit);                    //移除驱动程序入口
```

3. 程序编译

驱动程序一般采用 Make 工具进行编译，编译时要使用到编译过的 Linux 内核源代码。本节设计的 Demo 驱动程序没有涉及具体的硬件电路，所以可以将其编译成宿主机（x86 平台）上的驱动程序，也可以编译成目标机（ARM）上的驱动程序。这两种编译使用到的内核

不同，所以编译过程有一些差异。

1）编译成宿主机上的驱动程序

（1）首先查询宿主机上 Linux 内核版本号，命令如下。

```
#uname -a
Linux ubuntu64-vm 3.2.0-24-generic #37-Ubuntu SMP Wed Apr 25 08:43:22 UTC 2012 x86_64
x86_64 x86_64 GNU/Linux
```

（2）找到 Linux 内核源代码保存的路径。通常内核源代码保存在/usr/src 中的某一个子目录，根据查询到的上述宿主机上的 Linux 内核版本号，最终确认宿主机上 Linux 的内核源代码保存的路径是/usr/src/linux-headers-3.2.0-24-generic/。

（3）编写 Makefile 文件，文件内容如下。

```
obj-m:=demo.o
KERNELDIR := /usr/src/linux-headers-3.2.0-24-generic
default:
        make -C $(KERNELDIR) M=$(shell pwd) modules
clean:
        rm -f *.o *.ko *.mod.* modules.* Mo*.*
```

（4）编译程序。执行 make 命令后，编译出驱动程序 demo.ko。

2）编译成目标机上的驱动程序

本书使用的目标机是 FS4412，假设移植好的 Linux 内核源代码保存在/home/linux/workdir/driver/linux-3.14-fs4412 目录下，并通过了内核编译。

编写 Makefile 文件，内容如下。

```
obj-m:=demo.o
KERNELDIR :=/home/linux/workdir/driver/linux-3.14-fs4412/
default:
    make -C $(KERNELDIR) M=$(shell pwd) modules
clean:
    rm -f *.o *.ko *.mod.* modules.* Mo*.*
```

执行 make 命令后，编译出驱动程序 demo.ko。可以用 file 命令查询 demo.ko 的运行平台。

4. 驱动程序加载和设备文件创建

宿主机和目标机的驱动程序加载和设备文件创建方法相同。加载之前，可以先查看一下系统已加载的驱动程序，一种方法是使用 lsmod 命令，它只能查看模块化的驱动程序；另一种方法是查看/proc/devices 文件内容，其中会包括系统已加载的所有驱动程序。

使用 insmod 命令可以加载驱动程序，格式如下。

```
#insmod demo.ko
```

出现 Demo driver Init OK 提示信息，即表示驱动程序加载成功。

应用程序要通过设备文件来访问驱动程序，所以要创建一个设备文件给应用程序使用。假设设备文件名为/dev/demo，则创建设备文件的命令如下。

```
#mknod /dev/demo c 249 0
```

执行完创建设备文件的命令后，系统会新建一个名为/dev/demo 的设备文件，用 ls 命令可以查看该文件的详细信息，命令如下。

```
#ls  -l  /dev/demo
```

```
cr-x------  1  root  demo  249 , 0  1月 3  09:47  /dev/demo
```

其中，c 表示该文件是字符设备文件；demo 是设备名称；249 是主设备号；0 是次设备号；/dev/demo 是设备文件名。

5.2.3　Demo 测试程序设计

1. 程序分析

编写一个应用程序来测试驱动程序是否正确，要求测试程序 test_demo.c 的功能是首先从设备上读出数据，并将数据显示在屏幕上，然后向设备写入数据，再从设备上读出数据，又将数据显示在屏幕上。

【程序 5.3】 Demo 测试程序 test_demo.c。

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <string.h>

void Delay(int t)
{
    int i;
    for(;t>0;t--)
        for(i=0;i<2000;i++);
}

int main(int argc,char **argv)
{
    int fd;
    char buf[40];
    char buf_write[40]="This is Write String!";
    /*打开设备文件*/
    fd = open("/dev/demo",O_RDWR);
    if (fd<0) {
        perror("open");
        exit(1);
    }
    /*从设备中读取数据*/
    read(fd,buf,40);
    Delay(1000);
    printf("Frist read data:%s\n",buf);
```

```
    Delay(1000);
    /* 向设备写入数据 */
    write(fd,buf_write,strlen(buf_write));
    Delay(1000);
    read(fd,buf,strlen(buf_write));
    Delay(1000);
    printf("Second read data:%s\n",buf);
    Delay(1000);
    /* 关闭设备 */
    close(fd);
}
```

2. 编译和运行

1）编译测试程序

宿主机和目标机使用的编译器不同，所以编译命令有一些差异。

编译成宿主机上运行的程序，命令如下。

```
#gcc test_demo.c -o test_demo
```

编译成目标机上运行的程序，命令如下。

```
#arm-none-linux-gnueabi-gcc test_demo.c -o test_demo
```

2）运行测试程序

宿主机和目标机运行测试程序方法相同，测试程序运行结果如下。

```
#./test_demo
```

```
This is demo_open function
This is demo_read function
First read data: This is the initialization string
This is demo_write function
This is demo_read function
Second read data: This is Write String!
This is demo_release function
```

5.3　GPIO 应用实例

本节将介绍一个 GPIO 应用实例，即利用 GPIO 控制 4 个 LED 灯闪烁。

5.3.1　LED 灯控制电路概述

1. 硬件电路

控制 LED 灯的硬件电路如图 5.7 所示。Exynos 4412 处理器的 GPX2_7、GPX1_0、GPF3_4 和 GPF3_5 引脚分别连接到 Q3、Q8、Q9 和 Q10 三极管的基极，Q3、Q8、Q9 和 Q10 三极管的集电极分别通过电阻连接到 LED2、LED3、LED4 和 LED5 发光二极管的阴极，4 个发光二极管的阳极都连接到电源上。

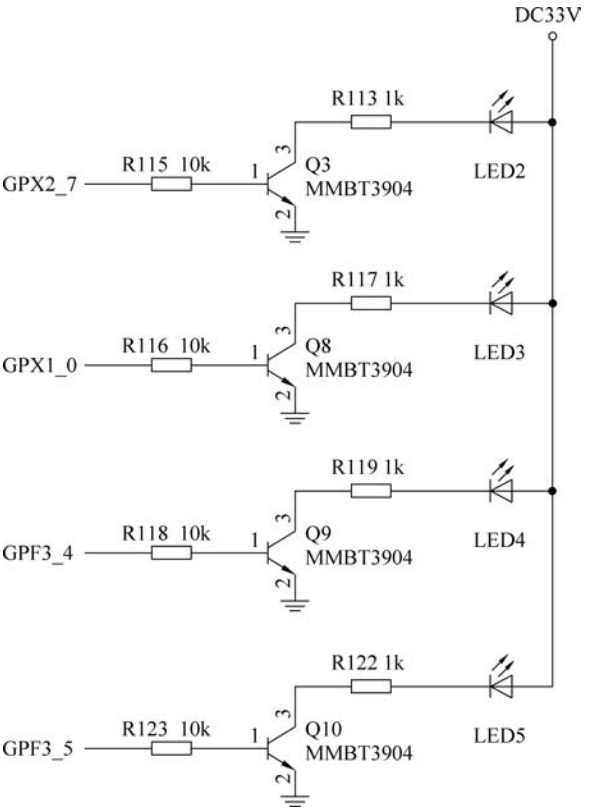


图 5.7　控制 LED 灯电路

2. 工作原理

以 LED2 为例，当 GPX2_7 引脚为高电平时，三极管 Q3 导通，这时电源、LED2、R113、Q3 和地之间形成一个通路，所以 LED2 灯亮；当 GPX2_7 引脚为低电平时，三极管 Q3 截止，不能形成通路，所以 LED2 灯灭。

3. 实现 LED 灯闪烁工作

(1) 设置 GPX2_7、GPX1_0、GPF3_4 和 GPF3_5 四个引脚都为输出方式。实现方法是设置 GPF3CON、GPX1CON 和 GPX2CON 控制寄存器信号为输出方式，具体置位请参考表 2.12～表 2.14。

(2) 向 GPX2_7、GPX1_0、GPF3_4 和 GPF3_5 四个引脚写数据。实现方法是向 GPF3DAT、GPX1DAT 和 GPX2DAT 数据寄存器对应的位写入数据，如果要点亮灯，则将对应位的值置 1，否则置 0。

(3) 按一定的时间间隔执行(2)的操作，让 LED 灯亮和灭，就可以实现闪烁效果。

5.3.2　LED 灯驱动程序设计

1. 程序功能

LED 灯驱动程序的功能说明如下。

(1) 采用传统方式将驱动程序编译成模块，以模块方式动态加载。

(2) 模块加载时完成设备的注册,设备名为 newled,主设备号为 500,次设备号为 0。

(3) 提供给系统 3 个 API 函数,即 open、close 和 ioctl。open 和 close 函数不写任何内容,ioctl 函数当选实现对 4 个 LED 灯的单独控制。

(4) 模块卸载时完成设备的注销。

2. 程序分析

因为 LED 驱动程序需要为系统提供 ioctl 函数,应用程序调用 ioctl 函数访问驱动程序时,需要使用内核中的_IO 宏函数对相关的结构体进行填充。为了方便应用程序编程,通常对 ioctl 函数中的 cmd 参数进行宏定义。现将这些宏定义保存在 fs4412_led.h 头文件中,头文件内容如下。

```
#ifndef FS4412_LED_HH
#define FS4412_LED_HH
#define LED_MAGIC 'L'
#define LED_ON      _IOW(LED_MAGIC,0,int)
#define LED_OFF     _IOW(LED_MAGIC,1,int)
#endif
```

LED 驱动程序源代码如程序 5.4。

【程序 5.4】 LED 驱动程序 fs4412_led.c。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include "fs4412_led.h"                    //cmd 参数的宏定义

MODULE_LICENSE("Dual BSD/GPL");

#define LED_MA 500
#define LED_MI 0
#define LED_NUM 1
/*寄存器的地址*/
#define FS4412_GPF3CON      0x114001E0
#define FS4412_GPF3DAT      0x114001E4
#define FS4412_GPX1CON      0x11000C20
#define FS4412_GPX1DAT      0x11000C24
#define FS4412_GPX2CON      0x11000C40
#define FS4412_GPX2DAT      0x11000C44

static unsigned int *gpf3con;
static unsigned int *gpf3dat;
static unsigned int *gpx1con;
static unsigned int *gpx1dat;
static unsigned int *gpx2con;
static unsigned int *gpx2dat;
```

```
struct cdev cdev;
/********** 点亮 LED 灯 **********/
void fs4412_led_on(int nr)
{
    switch(nr) {
        case 1:
            writel(readl(gpx2dat) | 1 << 7,gpx2dat);
            break;
        case 2:
            writel(readl(gpx1dat) | 1 << 0,gpx1dat);
            break;
        case 3:
            writel(readl(gpf3dat) | 1 << 4,gpf3dat);
            break;
        case 4:
            writel(readl(gpf3dat) | 1 << 5,gpf3dat);
            break;
    }
}
/********** 关闭 LED 灯 **********/
void fs4412_led_off(int nr)
{
    switch(nr) {
        case 1:
            writel(readl(gpx2dat) & ~(1 << 7),gpx2dat);
            break;
        case 2:
            writel(readl(gpx1dat) & ~(1 << 0),gpx1dat);
            break;
        case 3:
            writel(readl(gpf3dat) & ~(1 << 4),gpf3dat);
            break;
        case 4:
            writel(readl(gpf3dat) & ~(1 << 5),gpf3dat);
            break;
    }
}
/**********************************************/
static int fs4412_led_open(struct inode * inode,struct file * file)
{
    return 0;
}
/**********************************************/
static int fs4412_led_release(struct inode * inode,struct file * file)
{
    return 0;
}
/**********************************************/
static long fs4412_led_unlocked_ioctl(struct file * file,unsigned int cmd,unsigned long arg)
{
```

```
    int nr;

    if(copy_from_user((void *)&nr,(void *)arg,sizeof(nr)))
        return -EFAULT;
    if (nr < 1 || nr > 4)
        return -EINVAL;

    switch (cmd) {
        case LED_ON:
            fs4412_led_on(nr);
            break;
        case LED_OFF:
            fs4412_led_off(nr);
            break;
        default:
            printk("Invalid argument");
            return -EINVAL;
    }

    return 0;
}
/****************** I/O 映射 ******************************/
int fs4412_led_ioremap(void)
{
    int ret;

    gpf3con = ioremap(FS4412_GPF3CON,4);
    if (gpf3con == NULL) {
        printk("ioremap gpf3con\n");
        ret = -ENOMEM;
        return ret;
    }

    gpf3dat = ioremap(FS4412_GPF3DAT,4);
    if (gpf3dat == NULL) {
        printk("ioremap gpx2dat\n");
        ret = -ENOMEM;
        return ret;
    }

    gpx1con = ioremap(FS4412_GPX1CON,4);
    if (gpx1con == NULL) {
        printk("ioremap gpx2con\n");
        ret = -ENOMEM;
        return ret;
    }

    gpx1dat = ioremap(FS4412_GPX1DAT,4);
    if (gpx1dat == NULL) {
        printk("ioremap gpx2dat\n");
```

```
        ret = -ENOMEM;
        return ret;
    }
    gpx2con = ioremap(FS4412_GPX2CON,4);
    if (gpx2con == NULL) {
        printk("ioremap gpx2con\n");
        ret = -ENOMEM;
        return ret;
    }

    gpx2dat = ioremap(FS4412_GPX2DAT,4);
    if (gpx2dat == NULL) {
        printk("ioremap gpx2dat\n");
        ret = -ENOMEM;
        return ret;
    }

    return 0;
}
/*************** 释放映射 ********************************/
void fs4412_led_iounmap(void)
{
    iounmap(gpf3con);
    iounmap(gpf3dat);
    iounmap(gpx1con);
    iounmap(gpx1dat);
    iounmap(gpx2con);
    iounmap(gpx2dat);
}
/********* I/O 初始化 ***********************************/
void fs4412_led_io_init(void)
{
    writel((readl(gpf3con) & ~(0xff << 16)) | (0x11 << 16),gpf3con);
    writel(readl(gpx2dat) & ~(0x3 << 4),gpf3dat);

    writel((readl(gpx1con) & ~(0xf << 0)) | (0x1 << 0),gpx1con);
    writel(readl(gpx1dat) & ~(0x1 << 0),gpx1dat);

    writel((readl(gpx2con) & ~(0xf << 28)) | (0x1 << 28),gpx2con);
    writel(readl(gpx2dat) & ~(0x1 << 7),gpx2dat);
}
/*********************************************/
struct file_operations fs4412_led_fops = {
    .owner = THIS_MODULE,
    .open = fs4412_led_open,
    .release = fs4412_led_release,
    .unlocked_ioctl = fs4412_led_unlocked_ioctl,
};
/*********************************************/
static int fs4412_led_init(void)
```

```
{
    dev_t devno = MKDEV(LED_MA,LED_MI);
    int ret;

    ret = register_chrdev_region(devno,LED_NUM,"newled");
    if (ret < 0) {
        printk("register_chrdev_region\n");
        return ret;
    }

    cdev_init(&cdev,&fs4412_led_fops);
    cdev.owner = THIS_MODULE;
    ret = cdev_add(&cdev,devno,LED_NUM);
    if (ret < 0) {
        printk("cdev_add\n");
        goto err1;
    }
    ret = fs4412_led_ioremap();
    if (ret < 0)
        goto err2;
    fs4412_led_io_init();
    printk("Led init\n");
    return 0;
err2:
    cdev_del(&cdev);
err1:
    unregister_chrdev_region(devno,LED_NUM);
    return ret;
}
/*************************************************/
static void fs4412_led_exit(void)
{
    dev_t devno = MKDEV(LED_MA,LED_MI);

    fs4412_led_iounmap();
    cdev_del(&cdev);
    unregister_chrdev_region(devno,LED_NUM);
    printk("Led exit\n");
}
/*************************************************/
module_init(fs4412_led_init);
module_exit(fs4412_led_exit);
```

3. 程序编译和加载

编写 Makefile 文件，内容如下。

```
obj-m:=fs4412_led.o
KERNELDIR :=/home/linux/workdir/driver/linux-3.14-fs4412/
default:
    make -C $(KERNELDIR) M=$(shell pwd) modules
```

```
clean:
    rm -f *.o *.ko *.mod.* modules.* Mo*.*
```

编写完 Makefile 文件后，运行 make 命令，就可以生成 fs4412_led.ko 驱动程序。

将编译好的驱动程序下载到目标机上，然后使用 insmod fs4412_led.ko 加载驱动程序。同时，还要为应用程序创建一个设备文件，命令如下。

```
#mknod /dev/led c 500 0
```

执行完成后，会新建一个名为/dev/led 的文件。

5.3.3　LED 应用程序设计

应用程序实现 4 个 LED 灯从 LED2 到 LED5 逐个规律闪烁，程序内容如程序 5.5。

【程序 5.5】 LED 灯闪烁程序 test_led.c。

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>

#include "fs4412_led.h"                    //cmd 参数的宏定义

int main(int argc,char **argv)
{
    int fd;
    int i = 1;

    fd = open("/dev/led",O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    while(1)
    {
        ioctl(fd,LED_ON,&i);               //第 i 个 LED 灯亮
        usleep(500000);                    //延时
        ioctl(fd,LED_OFF,&i);              //第 i 个 LED 灯灭
        usleep(500000);
        if(++i == 5)
            i = 1;
    }

    return 0;
}
```

编译应用程序，具体命令如下。

```
# arm-none-linux-gnueabi-gcc test_led.c -o test_led
```

将应用程序下载到目标机上，然后运行，命令如下。

```
#./test_led
```

观察 LED 灯，可以看到 LED 灯实现了有规律闪烁。

5.4　PWM 应用实例

本节将通过实例介绍如何利用 PWM 定时器驱动蜂鸣器实现音乐播放器功能。

5.4.1　PWM 应用电路概述

1. 硬件电路

音乐播放器的硬件电路如图 5.8 所示，蜂鸣器 BZ1 一端连接到电源，另一端连接到三极管 Q11 的集电极，Q11 的基极连接到 Exynos 4412 的 TOUT0 引脚上。

2. 工作原理

TOUT0 无信号时，三极管 Q11 截止，蜂鸣器 BZ1 不发音。当 TOUT0 输出 PWM 信号时，三极管 Q11 会按照 PWM 信号的频率在导通和截止之间进行切换，这时蜂鸣器 BZ1 就会发出声音。PWM 信号的频率(三极管 Q11 导通和截止之间的切换频率)决定了发出声音的音符，信号的时间长度决定了节奏。

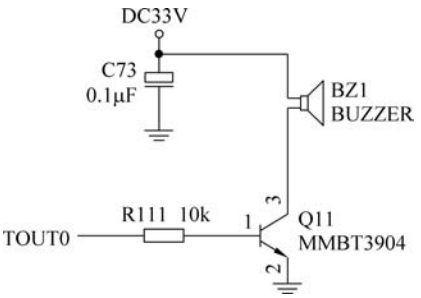


图 5.8　音乐播放器的硬件电路示意图

3. 实现音乐播放

实现音乐播放的工作流程如下。

(1) 设置 GPD0CON 控制寄存器[3:0]的值为 0x2，则 TOUT0 引脚输出 PWM 信号。

(2) 设计好哆、来、咪、发、索、啦、西等音符的工作频率。

(3) 通过设置 TCFG0、TCFG1 和 TCNTB0 寄存器得到相应频率的 PWM 信号。

(4) 有规律输出不同频率的 PWM 信号去驱动蜂鸣器，就可以实现音乐播放。

以上寄存器的设置可参考表 2.21～表 2.25。

5.4.2　PWM 驱动程序设计

1. 程序功能

PWM 驱动程序的功能说明如下。

(1) 采用传统方式将驱动程序编译成模块，以模块方式动态加载。

(2) 模块加载时完成设备注册，设备名为 pwm，主设备号为 501，次设备号为 0。

(3) 提供给系统 3 个 API 函数，即 open、close 和 ioctl，open 函数和 close 函数不写任何内容，ioctl 函数实现 PWM 信号的启动、关闭和频率的控制。

(4) 模块卸载时完成设备的注销。

2. 驱动程序分析

因为 PWM 驱动程序需要为系统提供 ioctl 函数，通常要对 ioctl 函数中的 cmd 参数进行宏定义。现将这些宏定义保存在 fs4412_pwm.h 头文件中，文件内容如下。

```
#ifndef FS4412_PWM_HH
#define FS4412_PWM_HH
#define PWM_MAGIC 'K'
//need arg = 0/1/2/3
#define PWM_ON _IO(PWM_MAGIC,0)
#define PWM_OFF _IO(PWM_MAGIC,1)
#define SET_PRE _IOW(PWM_MAGIC,2,int)
#define SET_CNT _IOW(PWM_MAGIC,3,int)
#endif
```

PWM 驱动程序源代码见程序 5.6。

【程序 5.6】 PWM 驱动程序 fs4412_pwm.c。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include "fs4412_pwm.h"              //cmd 参数宏定义

MODULE_LICENSE("GPL");
/*寄存器偏移地址*/
#define TCFG0      0x00
#define TCFG1      0x04
#define TCON       0x08
#define TCNTB0     0x0C
#define TCMPB0     0x10
/*寄存器基地址*/
#define GPDCON             0x114000A0
#define TIMER_BASE         0x139D0000

static int pwm_major = 501;
static int pwm_minor = 0;
static int number_of_device = 1;

struct fs4412_pwm
{
    unsigned int *gpdcon;
    void __iomem *timer_base;
    struct cdev cdev;
};

static struct fs4412_pwm *pwm;
```

```
static int fs4412_pwm_open(struct inode * inode, struct file * file)
{
    writel((readl(pwm->gpdcon) & ~0xf) | 0x2, pwm->gpdcon);
    writel(readl(pwm->timer_base + TCFG0) | 0xff, pwm->timer_base + TCFG0);
    writel((readl(pwm->timer_base + TCFG1) & ~0xf) | 0x2, pwm->timer_base +
TCFG1);
    writel(300, pwm->timer_base + TCNTB0);
    writel(150, pwm->timer_base + TCMPB0);
    writel((readl(pwm->timer_base + TCON) & ~0x1f) | 0x2, pwm->timer_base +
TCON);
    //writel((readl(pwm->timer_base + TCON) & ~(0xf << 8)) | (0x9 << 8), pwm->timer
_base + TCON);
    return 0;
}

static int fs4412_pwm_rlease(struct inode * inode, struct file * file)
{
    writel(readl(pwm->timer_base + TCON) & ~0xf, pwm->timer_base + TCON);
    return 0;
}

static long fs4412_pwm_ioctl(struct file * file, unsigned int cmd, unsigned long arg)
{
    int data;

    if (_IOC_TYPE(cmd) != 'K')
        return -ENOTTY;

    if (_IOC_NR(cmd) > 3)
        return -ENOTTY;

    if (_IOC_DIR(cmd) == _IOC_WRITE)
        if (copy_from_user(&data, (void * )arg, sizeof(data)))
            return -EFAULT;

    switch(cmd)
    {
    case PWM_ON:
        writel((readl(pwm->timer_base + TCON) & ~0x1f) | 0x9, pwm->timer_base +
TCON);
        break;
    case PWM_OFF:
        writel(readl(pwm->timer_base + TCON) & ~0x1f, pwm->timer_base + TCON);
        break;
    case SET_PRE:
        writel(readl(pwm->timer_base + TCON) & ~0x1f, pwm->timer_base + TCON);
        writel((readl(pwm->timer_base + TCFG0) & ~0xff) | (data & 0xff), pwm->timer_
base + TCFG0);
        writel((readl(pwm->timer_base + TCON) & ~0x1f) | 0x9, pwm->timer_base +
TCON);
```

```
            break;
        case SET_CNT:
            writel(data, pwm->timer_base + TCNTB0);
            writel(data >> 1, pwm->timer_base + TCMPB0);
            break;
        }

        return 0;
    }

    static struct file_operations fs4412_pwm_fops = {
        .owner = THIS_MODULE,
        .open = fs4412_pwm_open,
        .release = fs4412_pwm_rlease,
        .unlocked_ioctl = fs4412_pwm_ioctl,
    };

    static int __init fs4412_pwm_init(void)
    {
        int ret;
        dev_t devno = MKDEV(pwm_major, pwm_minor);

        ret = register_chrdev_region(devno, number_of_device, "pwm");
        if (ret < 0) {
            printk("faipwm : register_chrdev_region\n");
            return ret;
        }

        pwm = kmalloc(sizeof( * pwm), GFP_KERNEL);
        if (pwm == NULL) {
            ret = -ENOMEM;
            printk("faipwm: kmalloc\n");
            goto err1;
        }
        memset(pwm, 0, sizeof( * pwm));

        cdev_init(&pwm->cdev, &fs4412_pwm_fops);
        pwm->cdev.owner = THIS_MODULE;
        ret = cdev_add(&pwm->cdev, devno, number_of_device);
        if (ret < 0) {
            printk("faipwm: cdev_add\n");
            goto err2;
        }

        pwm->gpdcon = ioremap(GPDCON, 4);
        if (pwm->gpdcon == NULL) {
            ret = -ENOMEM;
            printk("faipwm: ioremap gpdcon\n");
            goto err3;
        }
```

```
    pwm->timer_base = ioremap(TIMER_BASE,0x20);
    if (pwm->timer_base == NULL) {
        ret = -ENOMEM;
        printk("failed: ioremap timer_base\n");
        goto err4;
    }
    return 0;
err4:
    iounmap(pwm->gpdcon);
err3:
    cdev_del(&pwm->cdev);
err2:
    kfree(pwm);
err1:
    unregister_chrdev_region(devno,number_of_device);
    return ret;
}

static void __exit fs4412_pwm_exit(void)
{
    dev_t devno = MKDEV(pwm_major,pwm_minor);
    iounmap(pwm->timer_base);
    iounmap(pwm->gpdcon);
    cdev_del(&pwm->cdev);
    kfree(pwm);
    unregister_chrdev_region(devno,number_of_device);
}
module_init(fs4412_pwm_init);
module_exit(fs4412_pwm_exit);
```

3. 程序编译和加载

编写 Makefile 文件，内容如下。

```
obj-m:=fs4412_pwm.o
KERNELDIR :=/home/linux/workdir/driver/linux-3.14-fs4412/
default:
    make -C $(KERNELDIR) M=$(shell pwd) modules
clean:
    rm -f *.o *.ko *.mod.* modules.* Mo*.*
```

编写完 Makefile 文件后，运行 make 命令，就可以生成 fs4412_pwm.ko 驱动程序。

将编译好的驱动程序下载到目标机上，然后使用 insmod fs4412_pwm.ko 加载驱动程序。同时，还要为应用程序创建一个设备文件，命令如下。

```
#mknod /dev/pwm c 501 0
```

执行完成后，会新建一个名为/dev/pwm 的文件。

5.4.3　PWM 应用程序设计

通过 PWM 信号可以控制蜂鸣器播放一首歌曲，实现方法是先选择一首歌，然后分解出音符和节奏，并将这些数据保存在 pwm_music.h 头文件中，然后编写应用程序，读出头文件中的数据，并将其送到驱动程序即可。pwm_music.h 头文件内容如下。

```
typedef struct
{
    int pitch;     //保存频率(音符)
    int dimation; //保存延时(节奏)

}Note;
//各种音符的频率
//  1        2       3        4       5    6     7
//261.63  293.66  329.63   349.23   392   440  493.88
#define DO 262
#define RE 294
#define MI 330
#define FA 349
#define SOL 392
#define LA  440
#define SI  494
#define TIME 6000
/*保存《世上只有妈妈好》歌谱*/
Note MotherLoveMeOnceAgain[]={
    //6.                  //_5          //3          //5
    {LA,TIME+TIME/2},{SOL,TIME/2},{MI,TIME},{SOL,TIME},

    //1^              //6_            //_5        //6-
    {DO*2,TIME},{LA,TIME/2},{SOL,TIME/2} ,{LA,2*TIME},
    //3       //5_        //_6            //5
    {MI,TIME},{SOL,TIME/2},{LA,TIME/2},{SOL,TIME},
    //3         //1_        //_6,          //1
    {MI,TIME},{DO,TIME/2},{LA/2,TIME/2},
    //5_        //_3        //2-             //2.
    {SOL,TIME/2},{MI,TIME/2},{RE,TIME*2},{RE,TIME+TIME/2},
    //_3    //5             //5_             //_6
    {MI,TIME/2},{SOL,TIME},{SOL,TIME/2},{LA,TIME/2},
    //3        //2             //1-             //5.
    {MI,TIME},{RE,TIME},{DO,TIME*2},{SOL,TIME+TIME/2},
    //_3        //2_        //_1         //6,_
    {MI,TIME/2},{RE,TIME/2},{DO,TIME/2},{LA/2,TIME/2},
    //_1        //5,--
    {DO,TIME/2},{SOL/2,TIME*3}
};
```

播放歌曲的应用程序见程序 5.7。

【程序 5.7】 PWM 应用程序 test_pwm.c。

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

#include "pwm_music.h"                                    //歌曲数据
#include "fs4412_pwm.h"                                   //cmd 参数宏定义

int main()
{
    int i = 0;
    int n = 2;
    int dev_fd;
    int div;
    int pre = 255;
    dev_fd = open("/dev/pwm",O_RDWR | O_NONBLOCK);
    if ( dev_fd == -1 ) {
        perror("open");
        exit(1);
    }
    ioctl(dev_fd,PWM_ON);                                 //控制 PWM 信号输出
    ioctl(dev_fd,SET_PRE,&pre);                           //设置预分频
    for(i = 0;i<sizeof(MotherLoveMeOnceAgain)/sizeof(Note);i++)
    {
        div = (PCLK/256/4)/(MotherLoveMeOnceAgain[i].pitch);
        ioctl(dev_fd,SET_CNT,&div);                       //控制频率(音符)
        usleep(MotherLoveMeOnceAgain[i].dimation * 50);//延时(节奏)
    }
    return 0;
}
```

编译应用程序,具体命令如下。

```
# arm-none-linux-gnueabi-gcc test_pwm.c  -o test_pwm
```

将应用程序下载到目标机上,然后运行,命令如下。

```
#./test_pwm
```

程序运行时,可以听到蜂鸣器播放出电子琴演奏的《世上只有妈妈好》歌曲,播放完成后程序结束。

5.5　ADC 应用实例

5.5.1　ADC 工作原理

ADC(Analog-to-Digital Converter,模数转换器)的任务是将连续变换的模拟信号转换为数字信号,以便计算机和数字系统进行存储、控制、显示和进行其他各种处理,建立起模拟信号源和 CPU 之间的连接,这在工业控制、数据采集及其他许多领域中都是不可缺少的。

ADC 种类繁多,分类方法不一,按照工作原理可分为积分型、逐次逼近型、并行比较型、串并行型、电容阵列逐次比较型及压频变换型等。

1. 积分型 ADC

积分型 ADC 将输入电压转换成时间(脉冲宽度信号)或频率(脉冲频率),然后由定时器/计数器获得数字值。积分型 ADC 实际上是 V-T 方式电压对时间的转换,先对输入的量化电压以固定时间正向积分,然后再对基准电压反向积分,计数出对应的 ADC 结果值。

双积分型 ADC 是一种间接 A/D 转换技术,控制逻辑电路如图 5.9 所示,积分器是转换器的核心部分,它的输入端所接开关 S1 由定时信号控制,当定时信号为不同电平时,极性相反的输入电压 u_i 和参考电压 V_{REF} 将分别加到积分器的输入端,进行两次方向相反的积分,积分时间常数 $\tau=RC$。过零比较器用来确定积分器的输出电压 u_o 过零的时刻,当 $u_o>0$ 时,比较器输出电压为低电平;当 $u_o\leqslant 0$ 时,比较器输出电压为高电平。比较器的输出信号接至时钟控制门 G 作为关门和开门信号。可见双积分型 ADC 具有很强的抗干扰能力,所以采用双积分型 ADC 可大大降低对滤波电路的要求。

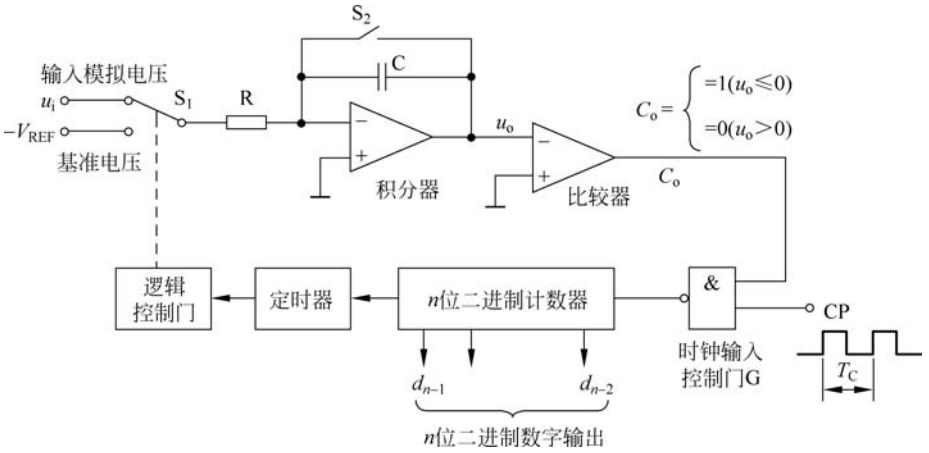


图 5.9　双积分型 ADC 的控制逻辑电路

2. 逐次逼近型 ADC

逐次逼近型 ADC 通常由比较器、DAC(Digital-to-Analog Converter,数模转换器)、寄存器和控制逻辑电路组成,如图 5.10 所示。

逐次逼近型 ADC 寄存器的数字量设置方法叫对分搜索法,转换过程如下所述。

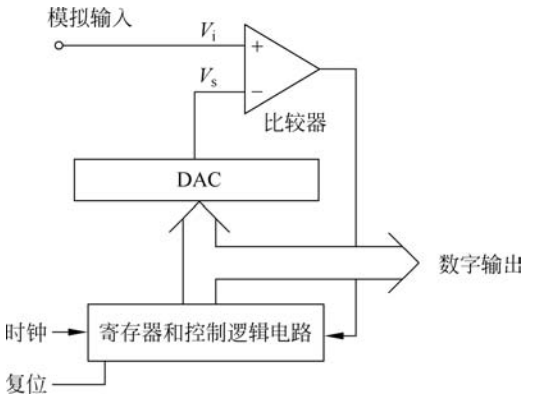


图 5.10　逐次逼近型 A/D 转换原理

(1) 初始化时,先将寄存器各位清零。

(2) 转换时,先将寄存器的最高位置 1,再将寄存器的数值送入 DAC,经数模转换后生成的模拟量送入比较器中与输入模拟量进行比较,若 $V_s < V_i$,则该位的 1 被保留,否则被清除; 然后再将次高位置 1,再将寄存器的数值送入 DAC,经数模转换后生成的模拟量送入比较器中与输入模拟量进行比较,若 $V_s < V_i$,则该位的 1 被保留,否则被清除; 重复上述过程,直至最低位,最后寄存器中的内容即为输入模拟值转换成的数字量。

对于 n 位逐次逼近型 ADC,要比较 n 次才能完成一次转换。因此,逐次逼近型 ADC 的转换时间取决于位数和时钟周期,转换精度取决于 DAC 的比较器的精度。逐次逼近型 ADC 可应用于许多场合,是应用最为广泛的一种 ADC。

5.5.2　ADC 的主要性能指标

1. 分辨率

分辨率(Resolution)是指 ADC 对输入电压的微小变化的响应能力的度量,它是数字输出的最低有效位(Least Significant Bit,LSB)所对应的模拟输入电平值。若输入电压的满刻度值用 VFS(Value of Full Scale)表示,转换器的位数为 n,则分辨率为$(1/2^n)$VFS。由于分辨率与转换器的位数 n 直接有关,所以常用位数来表示分辨率,如 8 位、10 位、12 位和 16 位等。

值得注意的是,ADC 的分辨率和精度是两个不同的概念。分辨率是指转换器所能分辨的模拟信号的最小变化值,精度是指转换器实际值与理论值之间的偏差。ADC 分辨率的高低取决于转换器位数的多少,但影响转换器精度的因素很多,分辨率高的 ADC,精度并不一定也高。

2. 绝对精度

绝对精度是指在输出端产生给定的数字量的条件下,实际需要的模拟输入值与理论上要求的模拟输入值之差。

3. 相对精度

相对精度是指满刻度值校准以后,任意数字输出所对应的实际模拟输入值(中间值)与理论值(中间值)之差。

4. 转换时间

转换时间是指完成一次 A/D 转换所需要的时间，即从启动信号开始到转换结束并得到稳定的数字输出量为止的总时间。一般来说，转换时间越短，转换速度越快。转换时间的倒数称为转换率。

5. 量程

量程指所能转换的输入电压范围，分单极性和双极性两种类型。例如，单极性量程为 0～＋5V、0～＋10V 等；双极性量程为－5～＋5V、－10～＋10V 等。

6. 积分线性误差

积分线性误差又称为线性误差，是指在没有偏移误差和增益误差的情况下，实际传输曲线与理想传输曲线之差。由于线性误差是由 ADC 特性随输入信号幅值变化而变化引起的，因此线性误差是不能进行补偿的，而且线性误差的数值会随温度的升高而增加。

7. 微分线性误差

微分线性误差是指实际代码宽度与理想代码宽度之间的最大偏差，以 LSB 为单位。微分线性误差也常用无失码分辨率表示。

在实际应用中，分辨率、绝对精度、相对精度、积分线性误差和微分线性误差等指标都用来衡量 ADC 测量精度，量程用于规定测量的电压范围，转换时间用于规定数据采集的最短时间间隔。

5.5.3　ADC 应用电路概述

本节将介绍实例——利用 Exynos 4412 内置 ADC 实现一个温度采集系统。Exynos 4412 内置 ADC 在本书 2.2.10 节已有比较详细的介绍。

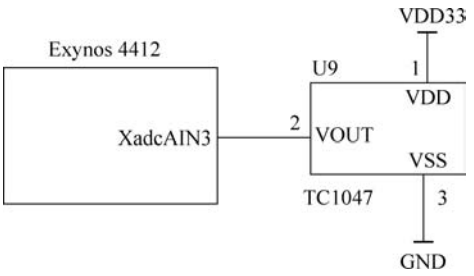


图 5.11　温度采集电路

1. 硬件电路

温度采集硬件电路如图 5.11 所示，TC1047 是温度传感器，该传感器有 3 个引脚，VDD33 是电源，VSS 是地，VOUT 是输出端；VOUT 连接到 Exynos 4412 自带 ADC 的第 3 个输入通道上，即 AIN3 引脚。

2. 工作原理

TC1047 温度传感器 VOUT 引脚的电压值与温度值存在一个对应关系，通过 Exynos 4412 内置 A/D 控制器的第 3 通道(AIN3)实时采集 VOUT 的电压值，再通过算法将电压值转换成温度值。

3. A/D 转换

A/D 转换的详细工作流程如下所述。

(1) 设置 A/D 转换输入为 AIN3，即设置 ADCMAX[3:0]的值为 0x3。

(2) 设置 A/D 转换精度，如果精度为 12 位，则设置 ADCCON[16]的值为 0x1。

(3) 设置预分频值，可以设置 ADCCON[13:6]的值为 0xFF。

(4) 允许进行预分频，设置 ADCCON[14]的值为 0x1。

(5) 启动 A/D 转换，设置 ADCCON[0]的值为 0x0。

(6) 每次 A/D 转换都需要一定的转换时间,在这段时间可以让程序休眠,当 A/D 转换结束后,ADC 会向系统发送一个中断,再让这个中断唤醒程序。

(7) 唤醒后,读出 A/D 转换结果,即读取 ADCDAT 寄存器前 12 位的值。

Exynos 4412 内置 ADC 涉及的寄存器可参见本书中的表 2.26～表 2.29。

5.5.4　温度采集驱动程序设计

本节将采用设备树编写驱动程序,包括两部分,即设备树和平台总线驱动(驱动程序)。采用设备树的驱动程序加载流程是先打开目标机电源,引导程序会将设备树的数据自动加载到 Linux 系统内核,当安装驱动程序时,内核会提取驱动程序中的设备信息(如. driver. of _match_table 下的. compatible 成员)与设备树中的设备信息(如. compatible 成员)进行匹配,如果匹配成功,则驱动程序可以调用 probe 函数继续加载驱动程序,否则(匹配失败)中止驱动程序加载。

1. *程序功能*

ADC 驱动程序的功能说明如下。

(1) 将驱动程序编译成模块,以模块方式动态加载。

(2) 模块加载时,完成设备的注册。设备名为 fs4412-adc,主设备号为 500,次设备号为 0。

(3) 提供给系统三个 API 函数,即 open、close 和 read。open 函数和 close 函数不写任何内容,read 函数实现读取 AIN3 通道的值。

(4) 模块卸载时,完成设备的注销。

2. *程序分析*

A/D 驱动程序的结构如图 5.12 所示,程序代码见程序 5.8。

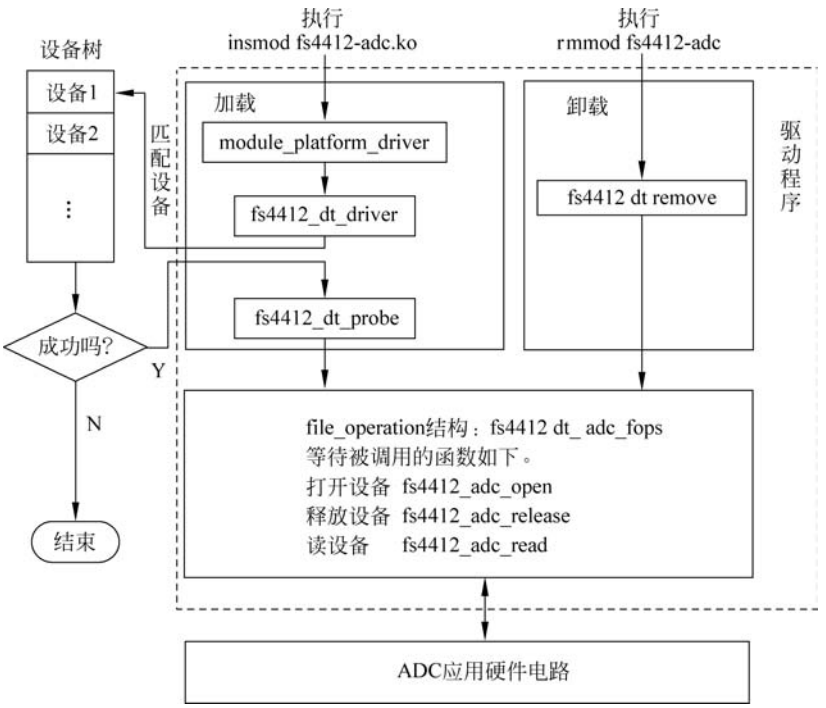


图 5.12　A/D 驱动程序的结构

【程序 5.8】 A/D 设备驱动程序 fs4412_adc.c。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/of.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <asm/uaccess.h>

#define FS4412_ADCCON           0x00
#define FS4412_ADCDAT           0x0C
#define FS4412_ADCCLRINT        0x18
#define FS4412_ADCMUX           0x1C

MODULE_LICENSE("GPL");

struct resource *mem_res;
struct resource *irq_res;
void __iomem *adc_base;
unsigned int adc_major = 500;
unsigned int adc_minor = 0;
struct cdev cdev;
int flags = 0;
wait_queue_head_t readq;

static int fs4412_adc_open(struct inode *inode, struct file *file)
{
    return 0;
}

static int fs4412_adc_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t fs4412_adc_read(struct file *file, char *buf, size_t count, loff_t *loff)
{
    int data = 0;

    if (count != 4)
        return -EINVAL;
    /*模数转换选择输入为第 3 通道*/
    writel(3, adc_base + FS4412_ADCMUX);
    /*设置模数转换参数,并启动模数转换*/
    writel(1<<0 | 1<<14 | 0xff<<6 | 0x1<<16, adc_base + FS4412_ADCCON);
    /*程序休眠,等待中断唤醒*/
```

```
    wait_event_interruptible(readq,flags == 1);
    /* 读取 A/D 数据寄存器的低 12 位的数据,它就是转换结果 */
    data = readl(adc_base + FS4412_ADCDAT) & 0xfff;
    if (copy_to_user(buf,&data,sizeof(data)))
        return -EFAULT;
    flags = 0;
    return count;
}
/* 发生中断时执行的函数 */
irqreturn_t adc_interrupt(int irqno,void *devid)
{
    flags = 1;
    writel(0,adc_base + FS4412_ADCCLRINT);    //清除中断
    wake_up_interruptible(&readq);            //唤醒程序
    return IRQ_HANDLED;
}

struct file_operations fs4412_dt_adc_fops = {
    .owner = THIS_MODULE,
    .open = fs4412_adc_open,
    .release = fs4412_adc_release,
    .read = fs4412_adc_read,

};

int fs4412_dt_probe(struct platform_device *pdev)
{
    int ret;
    dev_t devno = MKDEV(adc_major,adc_minor);
    printk("match OK\n");
    init_waitqueue_head(&readq);
    mem_res = platform_get_resource(pdev,IORESOURCE_MEM,0);
    irq_res = platform_get_resource(pdev,IORESOURCE_IRQ,0);
    if (mem_res == NULL || irq_res == NULL) {
        printk("No resource !\n");
        return -ENODEV;
    }

    printk("mem = %x: irq = %d\n",mem_res->start,irq_res->start);

    adc_base = ioremap(mem_res->start,mem_res->end - mem_res->start);
    if (adc_base == NULL) {
        printk("failed to ioremap address reg\n");
        return -EINVAL;
    };

    ret = request_irq(irq_res->start,adc_interrupt,IRQF_DISABLED,"adc",NULL);
    if (ret < 0) {
        printk("failed request irq: irqno = %d\n",irq_res->start);
        goto err1;
```

```
    }

    printk("major = %d,minor = %d,devno = %x\n",adc_major,adc_minor,devno);
    ret = register_chrdev_region(devno,1,"fs4412-adc");
    if (ret < 0) {
        printk("failed register char device region\n");
        goto err2;
    }

    cdev_init(&cdev,&fs4412_dt_adc_fops);
    cdev.owner = THIS_MODULE;
    ret = cdev_add(&cdev,devno,1);
    if (ret < 0) {
        printk("failed add device\n");
        goto err3;
    }

    return 0;

err3:
    unregister_chrdev_region(devno,1);
err2:
    free_irq(irq_res->start,NULL);
err1:
    iounmap(adc_base);
    return ret;
}

int fs4412_dt_remove(struct platform_device *pdev)
{
    dev_t devno = MKDEV(adc_major,adc_minor);
    printk("remove OK\n");
    cdev_del(&cdev);
    unregister_chrdev_region(devno,1);
    free_irq(irq_res->start,NULL);
    iounmap(adc_base);
    return 0;
}

static const struct of_device_id fs4412_dt_of_matches[] = {
    { .compatible = "fs4412,adc"},          //设备信息,要与设备树中的节点信息一致
    { /* nothing to be done! */},
};

MODULE_DEVICE_TABLE(of,fs4412_dt_of_matches);

struct platform_driver fs4412_dt_driver = {
    .driver = {
        .name = "fs4412-dt",
        .owner = THIS_MODULE,
```

```
        .of_match_table = of_match_ptr(fs4412_dt_of_matches),
    },
    .probe = fs4412_dt_probe,                    //驱动程序加载入口
    .remove = fs4412_dt_remove,                  //驱动程序删除入口
};
module_platform_driver(fs4412_dt_driver);        //平台总线驱动入口
```

3. 程序编译

编写 Makefile 文件，内容如下。

```
obj-m:=fs4412_adc.o
KERNELDIR :=/home/linux/workdir/driver/linux-3.14-fs4412/
default:
        make -C $(KERNELDIR) M=$(shell pwd) modules
clean:
        rm -f *.o *.ko *.mod.* modules.* Mo*.*
```

编写完 Makefile 文件后，运行 make 命令，就可以生成 fs4412_adc.ko 驱动程序。

将编译好的驱动程序下载到目标机上，然后使用 insmod fs4412_adc.ko 加载驱动程序。这时会报加载失败，因为没有修改设备树文件。

4. 修改设备树文件

设备树文件在内核目录下，文件名是 Exynos 4412-fs4412.dts，具体操作如下。

(1) 修改文件。打开文件，命令如下。

```
#cd /home/linux/workdir/driver/linux-3.14-fs4412/
#vi arch/arm/boot/dts/Exynos 4412-fs4412.dts
```

在文件的最后一行前，添加以下内容。

```
fs4412-adc@126c0000{
    compatible="fs4412,adc";                 //设备信息，要与驱动程序中的信息一致
    reg=<0x126c0000 0x20>;
    interrupt-parent=<&combiner>;
    interrupts=<10,3>;
};
```

(2) 编译。编译命令如下。

```
#make dtbs
```

编译完成后会在 arch/arm/boot/dts/目录下生成一个 Exynos 4412-fs4412.dtb 文件。参考 4.1.4 节介绍的内容，可将该文件写到目标机上。

5. 加载驱动程序及创建设备文件

将编译好的驱动程序下载到目标机上，然后使用 insmod fs4412_adc.ko 加载驱动程序。另外，还要为应用程序创建一个设备文件，命令如下。

```
#mknod /dev/adc c 500 0
```

执行完成后，会新建一个名为/dev/adc 的设备文件。

5.5.5　温度采集应用程序设计

1. 电压与温度转换模型

温度采集应用程序设计的关键是建立温度传感器的电压与温度转换模型。传感器 TC1047 输出电压与温度的关系如图 5.13 所示，电压与温度的转换公式如下。

$$T=(V-0.5)\times 100$$

其中，T 为温度值，单位为℃；V 为输出的电压值，单位为 V。

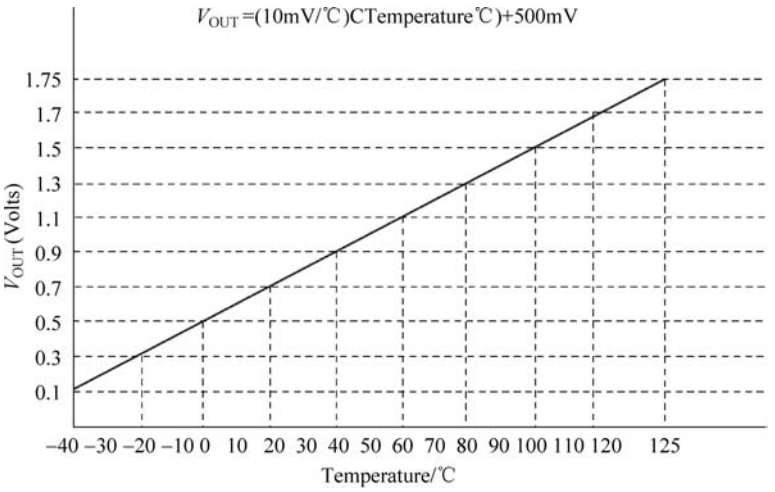


图 5.13　TC1047 温度与输出电压关系图

2. 程序分析

温度采集程序 temperature.c 的功能是首先读出第 3 通道的 A/D 值，然后转换成模拟电压，再将模拟电压转换成对应的温度值，最后将温度值显示在屏幕程序见程序 5.9。

【程序 5.9】 温度采集程序 temperature.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc,const char *argv[])
{
    int fd;
    int data;
    float d,tem;

    fd = open("/dev/adc",O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(1);
    }
        /*获取 A/D 转换结果*/
```

```
        read(fd,&data,sizeof(data));
        /*计算出模拟电压值*/
        d=1.8 * data / 4096;
        /*将模拟电压值转换成对应的温度值*/
        tem=(d-0.5)*100;
        printf("The temperature is: %4.2f ℃\n",tem);
        return 0;
}
```

3. 编译和运行

编译应用程序,具体命令如下。

```
#arm-none-linux-gnueabi-gcc temperature.c -o temperature
```

将应用程序下载到目标机上,然后运行./temperature,参考结果如下。

```
The temperature is: 21.08℃
```

5.6 练　习　题

1. 选择题

(1) 驱动程序的主要功能包括 3 个方面,但不包括(　　)。

A. 对设备初始化和释放　　B. 控制应用程序

C. 检测和处理设备出现的错误　　D. 数据传送

(2) 驱动程序主要由 3 部分组成,但不包括(　　)。

A. 自动配置和初始化子程序　　B. 服务于 I/O 请求的子程序

C. 中断服务子程序　　D. 服务于 CPU 子程序

(3) 字符设备提供给应用程序的入口点有很多,但(　　)不是。

A. ioctl　　B. read　　C. main　　D. open

(4) Linux 系统通常将设备分为 3 类,但不包括(　　)。

A. 输入设备　　B. 字符设备　　C. 块设备　　D. 网络设备

(5) Linux 系统用(　　)字母表示字符设备。

A. A　　B. B　　C. C　　D. N

(6) 设备文件包括了较多信息,但没有包括(　　)。

A. 设备类型　　B. 主设备号　　C. 次设备号　　D. 驱动程序名称

(7) 应用程序通过(　　)来操作字符设备。

A. 字符设备文件　　B. 块设备文件　　C. 网络设备文件　　D. 套接字

(8) 应用程序通过(　　)来操作网络设备。

A. 字符设备文件　　B. 块设备文件　　C. 网络设备文件　　D. 套接字

(9) 安装驱动程序的命令是(　　)。

A. insmod　　B. mknod　　C. rmmod　　D. lsmod

(10) 创建设备文件的命令是(　　)。

A. insmod　　B. mknod　　C. rmmod　　D. lsmod

(11) 内核内部通过(　　)数据结构识别设备。

A. file　　B. file_operations

C. inode　　D. device_struct

(12) 内核内部通过(　　)数据结构提供文件系统的入口点函数。

A. file　　B. file_operations

C. inode　　D. device_struct

(13) I/O 地址映射函数是(　　)。

A. copy_from_user　　B. copy_to_user

C. iounmap　　D. ioremap

(14) 从应用程序接收数据到内核态的函数是(　　)。

A. copy_from_user　　B. copy_to_user

C. iounmap　　D. ioremap

(15) 驱动程序编写有 3 种方法,但不包括(　　)。

A. 传统方法　　B. 现代方法　　C. 平台总线　　D. 设备树

2. 填空题

(1) 设备驱动程序的目标是屏蔽_____。

(2) 驱动程序运行在_____,应用程序运行在用户态。

(3) Linux 系统的设备一般分为三类,即:_____、_____和网络设备。

(4) 在 Linux 系统中,设备号包括两部分,即_____和_____设备号。

(5) Linux 驱动程序的编译方法有两种,即_____和_____。

(6) Linux 系统中用于删除模块化驱动程序的命令是_____。

(7) 字符设备的数据处理以_____为单位,块设备的数据处理以_____为单位。

(8) inode 结构用来记录文件的_____的信息。一个文件有_____个 inode 结构。

(9) 设备树编写驱动程序分为两部分,即_____和_____。

(10) 驱动程序编写有 3 种方法,即_____、_____和_____。

3. 简答题

(1) 简述驱动程序的主要功能。

(2) 简述驱动程序的组成。

(3) 简述驱动程序和应用程序的区别。

(4) 简述设备文件、驱动程序、主设备号和次设备号之间的关系。

(5) 简述字符设备驱动程序提供的常用入口点及其各自的功能。

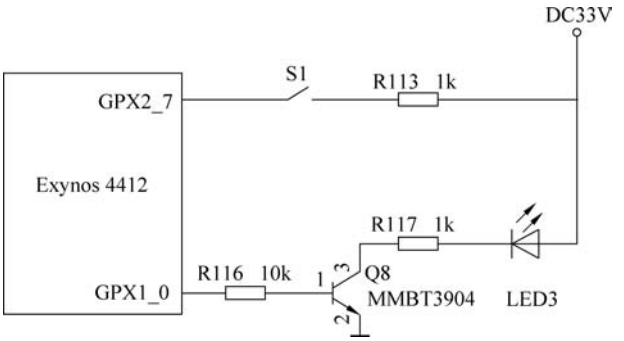
(6) 简述逐次逼近型 ADC 的结构及工作原理。

(7) 驱动程序编写有几种方法? 简述各种方法的特点。

4. 编程题

(1) Exynos 4412 的 GPX2_7、GPX1_0 端口分别连接到开关 S1 和 Q8 三极管,用开关 S1 来控制 LED 的亮和灭,具体电路如下图所示。请为该字符设备设计一个驱动程序和应

用程序，应用程序能够实现 S1 的开和关控制 LED 的亮和灭，也可以反过来。



（2）本书中的 A/D 驱动程序只能读取 AIN3 通道的数据，请再改写驱动程序，可以根据应用程序的需要读取 AIN0～AIN3 通道中任何一个通道的数据。