第 4 章　嵌入式 Linux 设备驱动开发

上一章讲解了 Linux 应用程序的开发，这些都是处于用户空间的内容。本章将进入内核空间，初步介绍嵌入式 Linux 设备驱动的开发。面对层出不穷的新硬件产品，必须有人不断地编写新的驱动程序以便让这些设备能够在 Linux 下正常工作，从这个意义上讲，讲述驱动程序的编写本身就是一件非常有意义的事情。

本章学习目标：

- 掌握 Linux 设备驱动的基本概念；
- 掌握设备驱动程序的基本功能；
- 掌握设备驱动运行过程、接口函数；
- 掌握字符设备驱动编程。

4.1　设备驱动概述

4.1.1　设备驱动简介及驱动模块

操作系统是通过各种驱动程序来驾驭硬件设备的，它为用户屏蔽了各种各样的设备，驱动硬件是操作系统最基本的功能，并且提供统一的操作方式。设备驱动程序是内核的一部分，硬件驱动程序是操作系统最基本的组成部分，在 Linux 内核源程序中也占有 60%以上。因此，熟悉驱动的编写是很重要的。

Linux 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 Linux 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）。

常见的驱动程序是作为内核模块动态加载的，比如声卡驱动和网卡驱动等，而 Linux 最基础的驱动，如 CPU、PCI 总线、TCP/IP 协议、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，比如 ext3 文件系统的驱动。因此，加载驱动就是加载内核模块。

下面首先列举一些模块相关的命令。

lsmod 命令列出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。如图 4.1.1 所示。

```
[root@WXL210 /]# lsmod
unifi_sdio 334981 0 - Live 0xbf0e5000
rt3070sta 661657 0 - Live 0xbf027000
wxl210_beep 1242 0 - Live 0xbf021000
wxl210_swzb 1735 0 - Live 0xbf01b000
wxl210_swrf 1763 0 - Live 0xbf015000
wxl210_adc1 2217 0 - Live 0xbf00f000
wxl210_leds 1269 0 - Live 0xbf009000
ft5x06_ts 8763 0 - Live 0xbf000000
[root@WXL210 /]#
```

图 4.1.1　lsmod 命令使用实例

rmmod 命令用于将当前模块卸载。

insmod 命令和 modprobe 命令用于加载当前模块，但 insmod 命令不会自动解决依存关系，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 命令可以根据模块间的依存关系以及/etc/modules.conf 文件中的内容自动加载其他有依赖关系的模块。

4.1.2 设备分类

Linux 的一个重要特点就是将所有的设备都当作文件进行处理，这一类特殊文件就是设备文件，可以使用前面提到的文件、I/O 相关函数进行操作，这样就大大方便了对设备的处理。它通常在/dev 下面存在一个对应的逻辑设备节点，这个节点以文件的形式存在。

Linux 系统的设备分为 3 类：字符设备、块设备和网络设备。

1．字符设备

字符设备通常指像普通文件或字节流一样，以字节为单位顺序读/写的设备，如并口设备、虚拟控制台等。字符设备可以通过设备文件节点访问，它与普通文件之间的区别在于普通文件可以被随机访问（可以前后移动访问指针），而大多数字符设备只能提供顺序访问，因为对它们的访问不会被系统所缓存。但也有例外，例如帧缓存（Frame Buffer）是一个可以被随机访问的字符设备。

2．块设备

块设备通常指一些需要以块为单位随机读/写的设备，如 IDE 硬盘、SCSI 硬盘、光驱等。块设备也是通过文件节点来访问的，它不仅可以提供随机访问，而且可以容纳文件系统（如硬盘、闪存等）。Linux 可以使用户态程序像访问字符设备一样每次进行任意字节的操作，只是在内核态内部中的管理方式和内核提供的驱动接口上不同。

通过文件属性可以查看它们是哪种设备文件（字符设备文件或块设备文件）。如图 4.1.2 所示，字母 c 开头表示字符设备，b 开头表示块设备。

```
crw-rw----    1 root     root       2, 175 Jan  1 08:00 ptyzf
brw-rw----    1 root     root       1,   0 Jan  1 08:00 ram0
```

图 4.1.2　文件属性

3．网络设备

网络设备通常是指通过网络能够与其他主机进行数据通信的设备，如网卡等。内核和网络设备驱动程序之间的通信需要调用一套数据包处理函数，它们完全不同于内核和字符以及块设备驱动程序之间的通信（read()、write()等函数）。Linux 网络设备不是面向流的设备，因此不会将网络设备的名字（如 eth0）映射到文件系统中去。

4.1.3 设备号

设备号是一个数字，它是设备的标志。就如前面所述，一个设备文件（也就是设备节点）可以通过 mknod 命令来创建，其中指定了主设备号和次设备号。主设备号表明设备的类型（如串口设备、SCSI 硬盘），与一个确定的驱动程序对应；次设备号通常用于标明不同的属性，例如不同的使用方法、不同的位置、不同的操作等，它标志着某个具体的物理设备。高字节为主设备号，低字节为次设备号。

例如，在系统中的块设备 IDE 硬盘的主设备号是 3，而多个 IDE 硬盘及其各个分区分别

赋予次设备号 1，2，3…，如图 4.1.3 所示，主设备号是 3，次设备号是 143。

```
crw-rw----    1 root     root        3, 143 Jan  1 08:00 ttyxf
```

图 4.1.3　设备号

4.1.4　驱动层次结构

Linux 下的设备驱动程序是内核的一部分，运行在内核模式下，也就是说设备驱动程序为内核提供了一个 I/O 接口，用户使用这个接口实现对设备的操作。图 4.1.4 显示了典型的 Linux 输入/输出系统中各层次结构和功能。

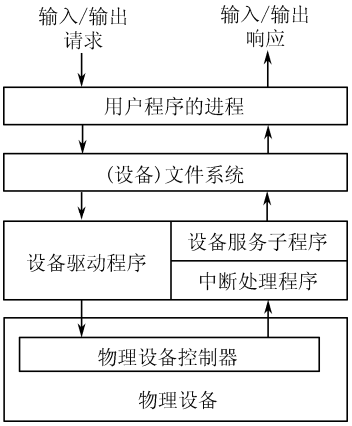

图 4.1.4　Linux 输入/输出系统

Linux 设备驱动程序包含中断处理程序和设备服务子程序两部分。设备服务子程序包含所有与设备操作相关的处理代码。它从面向用户进程的设备文件系统中接收用户命令，并对设备控制器执行操作。这样，设备驱动程序屏蔽了设备的特殊性，使用户可以像对待文件一样操作设备。

4.1.5　设备驱动程序与外界的接口

每种类型的驱动程序，不管是字符设备还是块设备都为内核提供相同的调用接口，因此内核能以相同的方式处理不同的设备。Linux 为每种不同类型的设备驱动程序维护相应的数据结构，以便定义统一的接口并实现驱动程序的可装载性和动态性。Linux 设备驱动程序与外界的接口可以分为如下 3 个部分。

① 驱动程序与操作系统内核的接口：这是通过数据结构 file_operations 来完成的。

② 驱动程序与系统引导的接口：这部分利用驱动程序对设备进行初始化。

③ 驱动程序与设备的接口：这部分描述了驱动程序如何与设备进行交互，这与具体设备密切相关。

它们之间的相互关系如图 4.1.5 所示。

4.1.6　设备驱动程序的特点

综上所述，Linux 中的设备驱动程序有如下特点。

① 内核代码：设备驱动程序是内核的一部分，如果驱动程序出错，则可能导致系统崩溃。

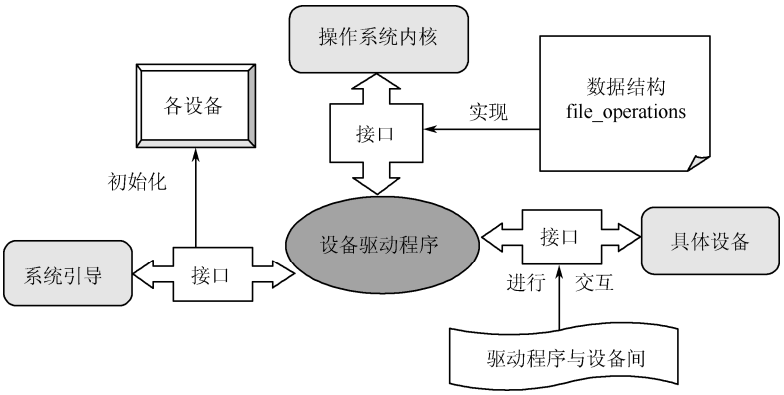

图 4.1.5　设备驱动程序与外界的接口

② 内核接口：设备驱动程序必须为内核或者其子系统提供一个标准接口。比如，一个终端驱动程序必须为内核提供一个文件 I/O 接口；一个 SCSI 设备驱动程序应该为 SCSI 子系统提供一个 SCSI 设备接口，同时 SCSI 子系统也必须为内核提供文件的 I/O 接口及缓冲区。

③ 内核机制和服务：设备驱动程序使用一些标准的内核服务，如内存分配等。

④ 可装载：大多数的 Linux 操作系统设备驱动程序都可以在需要时装载进内核，在不需要时从内核中卸载。

⑤ 可设置：Linux 操作系统设备驱动程序可以集成为内核的一部分，并可以根据需要把其中的某一部分集成到内核中，这只需要在系统编译时进行相应的设置即可。

⑥ 动态性：在系统启动且各个设备驱动程序初始化后，驱动程序将维护其控制的设备。如果该设备驱动程序控制的设备不存在也不影响系统的运行，那么此时的设备驱动程序只是多占用了一点系统内存罢了。

4.2　字符设备驱动编程

1．字符设备驱动编写流程

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 init_module()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 cleanup_module()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，用户的应用程序就可以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，在用户应用程序调用相应入口函数时执行相关的操作。init_module()入口点函数则不需要完成其他如 read()、write()之类功能。

上述函数之间的关系如图 4.2.1 所示。

2．重要数据结构

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<Linux/fs.h>中定义的 struct file_operations 结构，这是一个内核结构，不会出现在用户空间的程序中，它定义了常见文件 I/O 函数的入口，如下所示：

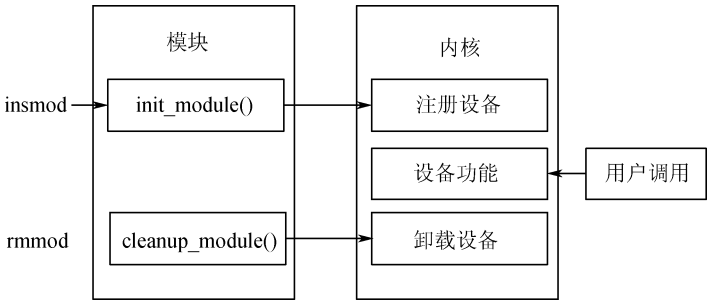

图 4.2.1　设备驱动程序流程图

```
struct file_operations
{
    loff_t (*lseek) (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 *);
};
```

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，struct file 结构提供关于被打开的文件信息，主要用于与文件系统对应的设备驱动程序使用。struct 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 */
};
```

3．设备驱动程序的主要组成

（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()，此后该设备就会从/proc/devices 里消失。其中主设备号和次设备号不能大于 255。

register_chrdev()函数格式见表 4.2.1。

表 4.2.1　register_chrdev()函数语法要点

所需头文件	#include <linux/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，返回所分配的主设备号，且设备名就会出现在/proc/devices 文件里
	出错：−1

unregister_chrdev()函数格式见表 4.2.2。

表 4.2.2　unregister_chrdev()函数语法要点

所需头文件	#include <linux/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 文件里消失
	出错：−1

（2）设备号相关函数

前面已经提到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公用同一个驱动程序，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 Linux2.6 的版本中，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。

分配设备号有静态和动态的两种方法。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()函数）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。

它们的函数格式见表 4.2.3。

表 4.2.3　设备号分配与释放函数语法要点

所需头文件	#include <linux/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 Linux 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 struct cdev 结构变量。首先使用 cdev_alloc()函数向系统申请分配 struct 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，这样新设备可以被用起来了。

如果要从系统中删除一个设备，则要调用 cdev_del()函数。具体函数格式见表 4.2.4。

表 4.2.4　最新版本的字符设备注册

所需头文件	#include <linux/fs.h>
函数原型	sturct cdev *cdev_alloc(void) void cdev_init(struct cdev *cdev,struct file_operations *fops) nt 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

（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。但通常情况下，在 open()函数接口中要完成如下工作：

- 递增计数器，检查错误；
- 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。

其中，递增计数器是用于设备计数的。由于设备在使用时通常会打开多次，也可以由不同的进程所使用，所以若有一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；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。

释放设备时要完成的工作如下：

- 递减计数器 MOD_DEC_USE_COUNT（最新版本已经不再使用）；
- 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）；
- 在最后一次释放设备操作时关闭设备。

（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里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，见表 4.2.5。

表 4.2.5　read()、write()函数接口语法要点

所需头文件	#include <linux/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

内核空间地址和用户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()函数和 copy_from_user()函数的格式见表 4.2.6。

表 4.2.6　copy_to_user()函数和 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

（7）ioctl()函数

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见表 4.2.7。

表 4.2.7　ioctl()函数接口语法要点

所需头文件	#include <linux/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

4.3　GPIO 驱动程序实例

4.3.1　LED 灯实验

【实验目的】

（1）熟悉 Linux 驱动实验原理；

（2）掌握 Linux 驱动加载和运行。

【实验设备】

（1）FS4412 开发板；

（2）USB 转串口下载线；

（3）计算机。

【实验内容】

编写 LED 驱动程序，在内核中编译成 ko 库文件，动态加载此驱动程序库。编写应用程序，调用此驱动函数驱动 LED 点亮或熄灭。

【实验原理】

如图 4.3.1 所示，LED 分别由 4 个 GPIO 口控制亮灭，实际控制 LED 灯亮灭就是控制 GPIO 的输出状态，见表 4.3.1。

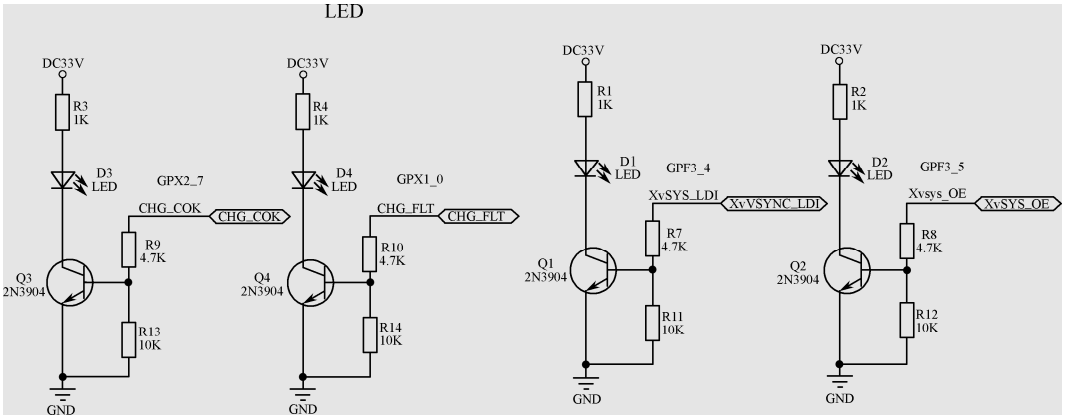

图 4.3.1　LED 驱动电路

表 4.3.1　LED 对应 GPIO 口

LED	对应的 I/O 寄存器名称
LED1	GPF3_4
LED2	GPF3_5
LED3	GPX2_7
LED4	GPX1_0

【实验步骤】

（1）实现 Windows 与 Ubuntu 下文件共享

单击“管理→虚拟机设置”命令，弹出“虚拟机设置”对话框，在“选项”页面单击“共享文件夹”项，勾选“总是启用”项，单击“添加”按钮，弹出“浏览文件夹”对话框，单击“Share”选项，单击“确定”按钮。如图 4.3.2、图 4.3.3 和图 4.3.4 所示。

图 4.3.2　Windows 与 Ubuntu 下文件共享设置

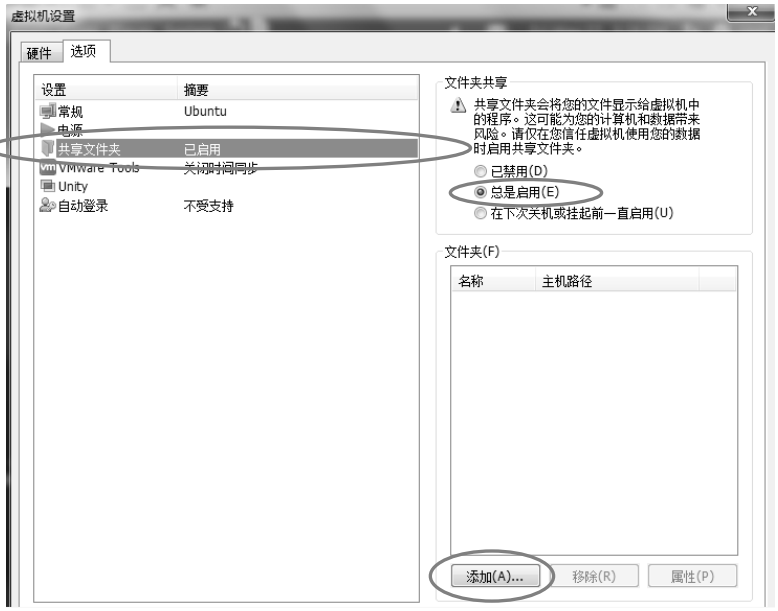

图 4.3.3　“选项”页面设置

图 4.3.4　浏览文件夹

添加完成之后，可以在 Ubuntu 终端输入如图 4.3.5 所示命令即可进入所选的共享文件夹。

```
gec@ubuntu:~$ cd /mnt/hgfs/
gec@ubuntu:/mnt/hgfs$ ls
share
gec@ubuntu:/mnt/hgfs$
```

图 4.3.5　进入共享文件夹

（2）编译环境准备

将“基于嵌入式系统的物联网实验开发光盘/实验代码/第 4 章/移植好的内核源码”路径下的压缩文件 Linux-3.14-fs4412.tar.xz 复制到共享文件夹下，如图 4.3.6 和图 4.3.7 所示。

图 4.3.6　压缩文件

```
gec@ubuntu:/mnt/hgfs$ cd share/
gec@ubuntu:/mnt/hgfs/share$ ls
linux-3.14-fs4412.tar.xz
```

图 4.3.7　共享文件夹下压缩文件

复制内核源码到 home 目录下，切换到 home 目录（不要在共享文件夹下编译内核源码），如图 4.3.8 所示。

```
gec@ubuntu:/mnt/hgfs/share$ cp ./linux-3.14-fs4412.tar.xz ~
gec@ubuntu:/mnt/hgfs/share$ cd ~
gec@ubuntu:~$ ls
Desktop    examples.desktop          Pictures   Videos
Documents  linux-3.14-fs4412.tar.xz  Public     workdir
Downloads  Music                     Templates
```

图 4.3.8　切换到 home 目录

解压内核源码，如图 4.3.9 所示。

```
gec@ubuntu:~$ tar xvf linux-3.14-fs4412.tar.xz
```

图 4.3.9　解压内核文件

进入源码目录，编译内核源码，如图 4.3.10 所示。

```
gec@ubuntu:~$ cd linux-3.14-fs4412/
gec@ubuntu:~/linux-3.14-fs4412$ ls
arch       Documentation     init      MAINTAINERS      REPORTING-BUGS  usr
block      drivers           ipc       Makefile         samples         virt
build.sh   firmware          Kbuild    mm               scripts
COPYING    fs                Kconfig   Module.symvers   security
CREDITS    fs4412_defconfig  kernel    net              sound
crypto     include           lib       README           tools
gec@ubuntu:~/linux-3.14-fs4412$ make uImage
```

图 4.3.10　编译内核源码

如果编译成功，则出现如图 4.3.11 所示信息。

```
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.14.0
Created:      Sat Aug 30 12:13:19 2014
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3027976 Bytes = 2957.01 kB = 2.89 MB
Load Address: 40008000
Entry Point:  40008000
  Image arch/arm/boot/uImage is ready
```

图 4.3.11　编译成功信息

（3）编译驱动程序及应用程序源码

将“基于嵌入式系统的物联网实验开发光盘/实验代码/第 4 章”路径下的 fs4412_led 文件复制到共享文件夹下，如图 4.3.12 和图 4.3.13 所示。

图 4.3.12　LED 实例源码

```
gec@ubuntu:~$ cd /mnt/hgfs/share/
gec@ubuntu:/mnt/hgfs/share$ ls
fs4412_led  linux-3.14-fs4412.tar.xz
```

图 4.3.13　共享文件夹下 LED 实例源码

进入“fs4412_led/fs4412_led_dev”目录，如图 4.3.14 所示。

```
gec@ubuntu:/mnt/hgfs/share$ cd fs4412_led/
gec@ubuntu:/mnt/hgfs/share/fs4412_led$ ls
fs4412_led_app  fs4412_led_dev
gec@ubuntu:/mnt/hgfs/share/fs4412_led$ cd fs4412_led_dev/
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_dev$ ls
fs4412_led.c  fs4412_led.h  Makefile
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_dev$
```

图 4.3.14　进入 fs4412_led/fs4412_led_dev 目录

修改 makefile 文件第 3、4 行，如图 4.3.15 所示。

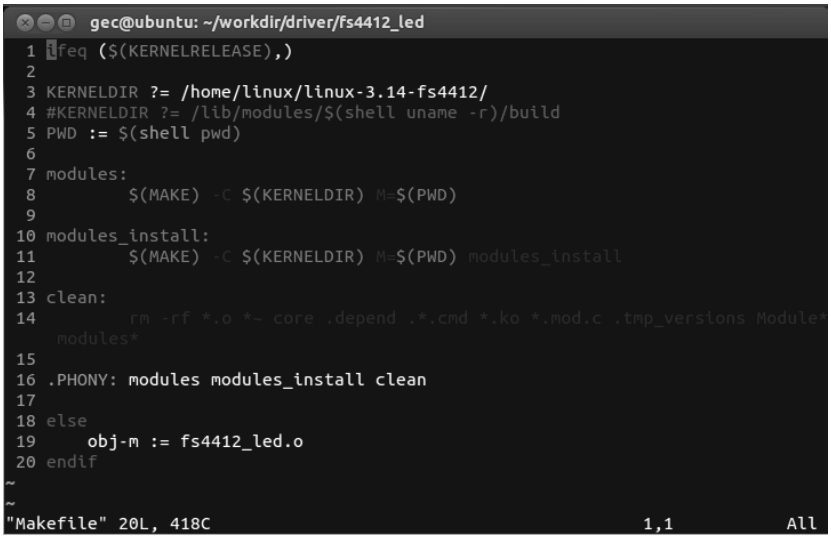

图 4.3.15　修改 makefile 文件

修改内核源码的路径和交叉编译工具链，如图 4.3.16 所示。

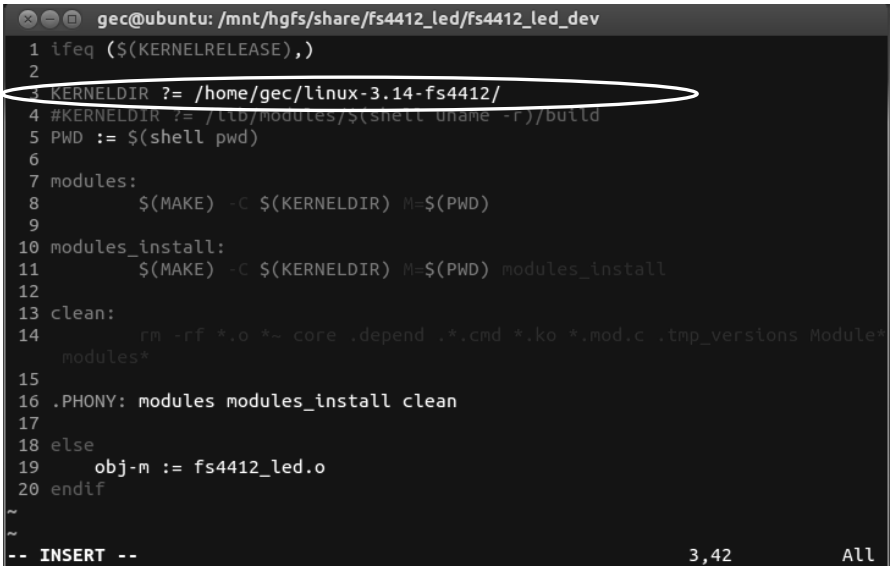

图 4. 3. 16　修改内核源码的路径和交叉编译工具链

保存退出。执行 make 命令编译源码，如图 4.3.17 所示。

图 4.3.17　执行 make 命令

查看生成的 ko 文件，如图 4.18 所示。

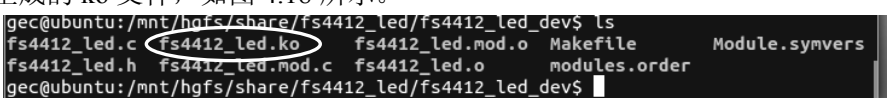

图 4.3.18　生成 ko 文件

进入 fs4412_led_app 目录，编译应用程序源码，如图 4.3.19 所示。

```
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_dev$ cd ../
gec@ubuntu:/mnt/hgfs/share/fs4412_led$ cd fs4412_led_app/
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_app$ ls
led.c
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_app$ arm-linux-gcc led.c -o led
```

图 4.3.19　编译应用程序源码

生成可执行文件，如图 4.3.20 所示。

```
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_app$ ls
led  led.c
gec@ubuntu:/mnt/hgfs/share/fs4412_led/fs4412_led_app$
```

图 4.3.20　生成可执行文件

（4）把驱动文件及可执行应用文件下载到实验箱

用 USB 转串口线把实验箱与 PC 相连，打开串口工具 SecureCRT，单击“文件→快速连接”命令，各选项选择如图 4.3.21 所示。

连接之后打开实验箱，如图 4.3.22 所示。

使用“rz”命令下载之前所生成的.ko 文件到实验箱，如图 4.3.23 所示。

图 4.3.21　串口工具连接设置

同样，使用“rz”命令下载之前所生成的 led 文件到实验箱。

修改 ko 文件权限，如图 4.3.24 所示。

（5）加载驱动，并执行应用程序

如图 4.3.25 所示，执行之后，可以看到 4 个 LED 间隔闪烁。

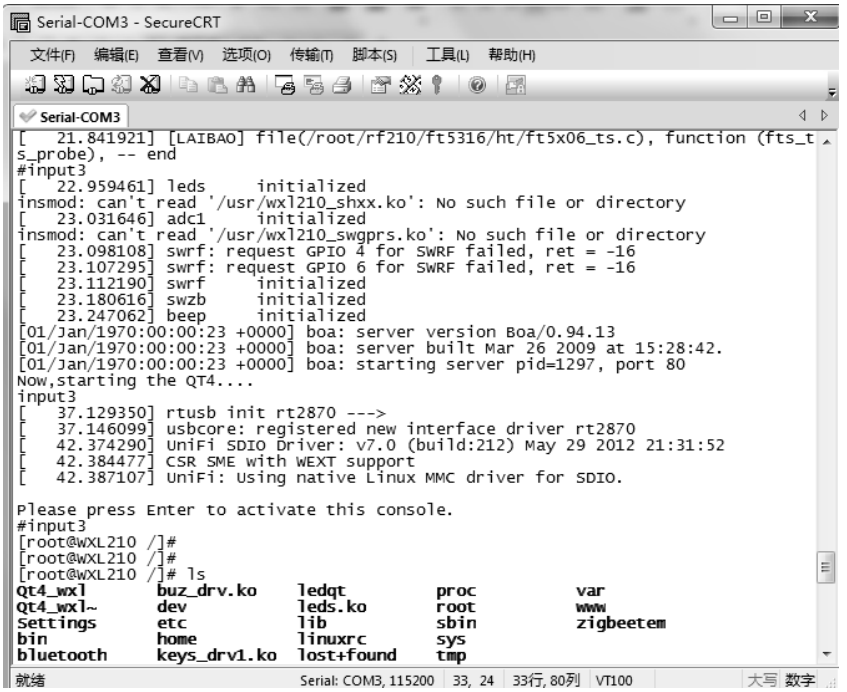

图 4.3.22　连接实验箱

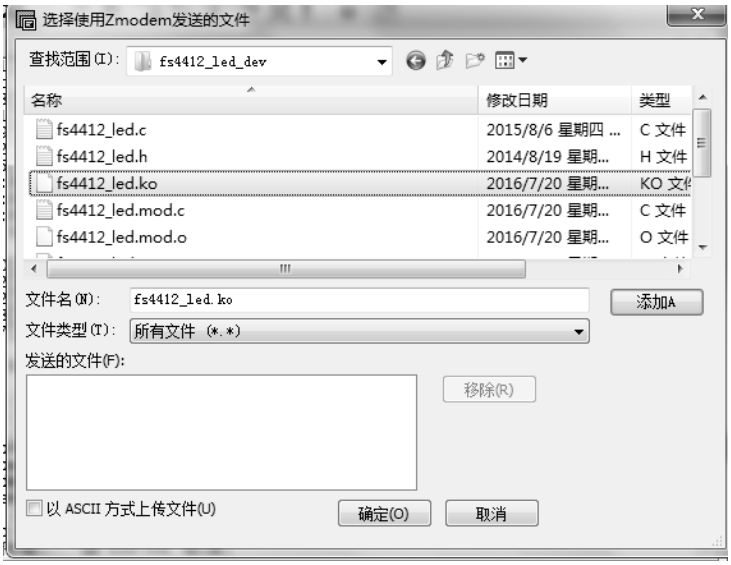

图 4.3.23　下载文件

```
[root@WXL210 /]# ls
Qt4_wxl          dev              lib              root             www
Qt4_wxl~         etc              linuxrc          sbin             zigbeetem
Settings         fs4412_led.ko    lost+found       sys
bin              home             mnt              tmp
bluetooth        led              mytest           usr
button_ctl       ledqt            proc             var
[root@WXL210 /]# chmod 777 fs4412_led.ko
[root@WXL210 /]# chmod 777 led
```

图 4.3.24　修改权限

```
# insmod fs4412_led.ko
# mknod /dev/led c 500 0
# ./test
```

图 4.3.25　加载驱动，执行应用程序

小练习

编写驱动程序和应用程序，实现 LED 流水灯功能。

4.3.2　按键驱动实例

【实验目的】

（1）熟悉 Linux 驱动实验原理；

（2）掌握 Linux 驱动加载和运行；

（3）利用按键驱动的编写掌握 Linux 下中断的实现。

【实验设备】

（1）FS4412 开发板；

（2）USB 转串口下载线；

（3）计算机。

【实验内容】

编写按键驱动程序，在内核中编译成 ko 库文件，动态加载此驱动程序库。编写应用程序调用此驱动函数采集按键值。

【实验原理】

如图 4.3.26 所示，3 个按键 K2，K3，K4 分别对应 3 个 GPIO，且 GPIO 与中断复用。当按键没有按下时 GPIO 处于高电平，当按键按下时 GPIO 处于低电平，所以本例将 GPIO 设置为中断功能，当按键按下时下降沿触发中断，CPU 通过中断确定按键状态。

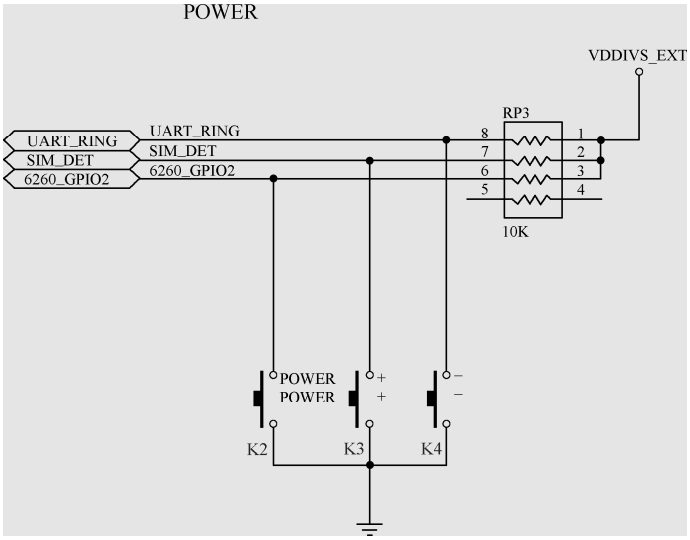

图 4.3.26　按键驱动电路

【实验步骤】

（1）编译生成驱动文件（参照 4.3.1 节实验步骤）

将“基于嵌入式系统的物联网实验开发光盘/实验代码/第 4 章”路径下的 fs4412_key 文件复制到共享文件夹下，如图 4.3.27 和图 4.3.28 所示。

图 4.3.27　按键实验源码文件

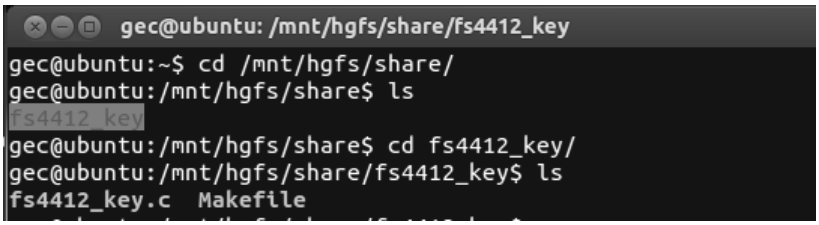

图 4.3.28　共享文件夹下按键实验源码文件

修改 makefile 文件的第 3，4 行，如图 4.3.29 所示。

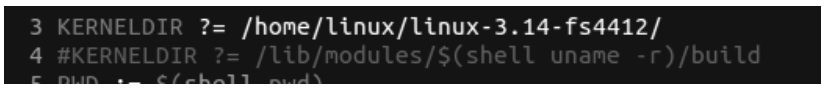

图 4.3.29　修改 makefile 文件

修改内核源码的路径和交叉工具链，如图 4.3.30 所示。

图 4.3.30　修改内核源码的路径和交叉工具链

保存退出。执行 make 命令编译源码，生成 ko 文件，如图 4.3.31 所示。

图 4.3.31　使用 make 命令编译源码

（2）修改设备树文件

进入 Linux 内核，如图 4.3.32 所示。

图 4.3.32　进入内核目录

打开设备树文件，如图 4.3.33 所示。

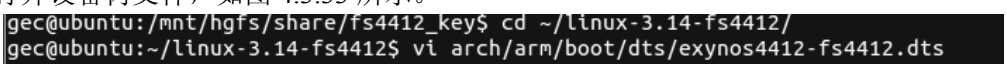

图 4.3.33　打开设备树文件

添加如下内容，如图 4.3.34 所示。

重新编译设备树文件，并复制到/tftpboot 目录下，如图 4.3.35 和图 4.3.36 所示。

```
fs4412-key{
        compatible = "fs4412,key";
        interrupt-parent = <&gpx1>;
        interrupts = <1 2>, <2 2>;
};
```

图 4.3.34　添加内容

```
gec@ubuntu:~/linux-3.14-fs4412$ make dtbs
```

图 4.3.35　重新编译设备树文件

```
gec@ubuntu:~/linux-3.14-fs4412$
gec@ubuntu:~/linux-3.14-fs4412$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb  /tftpboot
```

图 4.3.36　复制设备树文件/tftpboot 目录下

（3）下载并执行代码

参照 4.3.1 节步骤，启动开发板，并把 ko 文件下载到开发板，修改权限，并加载驱动。

加载驱动后屏幕会显示“match OK”的字样，否则说明设备树修改错误或设备树文件没有被正确加载，如图 4.3.37 所示。

```
[root@farsight ]# insmod fs4412_key.ko
[  107.725000] match OK
```

图 4.3.37　正确加载驱动

连续按下按键 K2 和 K3，屏幕会打印出按键对应的中断号，如图 4.3.38 所示。

```
[root@farsight ]# insmod fs4412_key.ko
[  107.725000] match OK
[root@farsight ]# [  112.745000] irqno = 168
[  114.005000] irqno = 169
[  114.570000] irqno = 168
[  115.180000] irqno = 169
[  115.695000] irqno = 168
[  115.835000] irqno = 168
[  116.215000] irqno = 169
[  116.640000] irqno = 168
[  117.095000] irqno = 169
```

图 4.3.38　运行结果

☞ 小练习

编写驱动程序和应用程序，实现一个按键控制一盏灯的亮灭功能。