# 第十一章 异步 IO

这里要说的异步 IO 准确的说应该叫“信号驱动的异步 I/O”，也可以成为异步通知。前面两章说的阻塞和非阻塞 IO，他们都是同步 IO，需要应用程序不断的轮询设备是否可以被访问。而异步 IO 模型下，设备可被访问时，可以由驱动程序主动通知应用程序进行访问。他的形式类似于硬件层面的中断，可以理解为由软件实现的模拟中断机制。

## 11.1 Linux 中的异步 IO

### 11.1.1 信号

Linux 系统中，使用信号来实现异步 IO 机制。在头文件 arch/xtensa/include/uapi/asm/signal.h 的 34~72 行定义了所有 Linux 系统支持的信号，相当于是中断系统中的中断号。信号的使用和中断也很想色，一个信号对应一个回掉函数，收到信号时，就会执行对应的回掉。

对于应用程序来说，信号是一种模拟中断方法，调用接口使用即可，而驱动程序则需要实现相应的方法，提供底层的支持。分别来看应用程序和驱动程序中信号的用法和实现方法。

### 11.1.2 应用程序中信号的使用

应用程序中使用信号的步骤如下：

1) 指定信号并指定对应的信号处理函数

使用下面的函数来选择一个信号并指定对应的信号处理函数：

```
sighandler_t signal(int signum, sighandler_t handler)
```

参数说明：

**signum**：需要选择的信号，从文件 arch/xtensa/include/uapi/asm/signal.h 的 34~72 行宏定义中选一个。

**handler**：对应的信号处理函数。

**返回值**：成功返回信号的前一个处理函数，失败返回 SGI_ERR。

信号处理函数 sighandler_t 的原型如下：

```
typedef void (*sighandler_t)(int)
```

2) 设置即将接收信号的进程 ID

```
fcntl(fd, F_SETOWN, getpid());
```

fd 是设备文件句柄，F_SETOWN 命令表示设置将接收 SIGIO 或 SIGURG 信号的线程 ID，getpid()是获取当前线程 ID。

3) 获取当前线程状态，并使它进入 FASYNC 状态

使用下面的方法获取当前进程状态：

```
flag = fcntl(fd, F_GETFL);
```

flag 是整型，fd 是设备文件句柄，F_GETFL 命令表示获取当前线程状态。

获取到当前线程状态后，在当前状态的基础上，设置进程为 FASYNC 状态，即开启当前

线程的异步通知功能，使用下面的命令：

```
fcntl(fd, F_SETFL, flags | FASYNC);
```

fd 是设备文件句柄，F_SETFL 命令表示设置当前进程状态，flag | FASYNC 中 flag 是获取到的当前线状态，或上 FASYNC 即在当前的状态上增加 FASYNC 状态。当线程的 FASYNC 状态被设置时，对应驱动程序 file_pperations 操作函数中的 fasync 就会被调用。

### 11.1.3 驱动程序中信号的实现

在驱动程序中支持信号，需要以下步骤：

1) 在设备结构体中声明 fasync_struct 结构体指针：

```
struct xxx_dev {
    ......
    struct fasync_struct *fasync;
}
```

2) 实现 fasync 函数

fasync()函数的实现，一般只需将该函数的 3 个参数以及 fasync_struct 结构体指针传入 fasync_helper()函数就可以了，如下：

```
static int xxx_fasync(int fd,struct file *filp, int mode)
{
    struct xxx_dev *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev-> fasync);
}

static struct file_operations xxx_ops = {
    ......
    .fasync = xxx_fasync,
};
```

然后在 release 函数中调用 fasync 使用释放 fasync_struct 结构体，如下：

```
static int xxx_release(struct inode *inode, struct file *filp)
{
    ......
    return xxx_fasync(-1, filp, 0);
}
```

3) 设备可访问时，发出信号

应用程序开启异步通知之后，就是在等待设备可操作的信号，驱动程序需要在设备可操作时发出信号，使用函数：

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
```

参数说明：

**fp**：目标 fasync_struct 结构体变量指针的地址。

**sig**：发送的信号类型，范围是文件 arch/xtensa/include/uapi/asm/signal.h 的 34~72 行宏定义，需要和应用程序的需求一致。

**band**：设备可写时设为 POLL_IN，可读时为 POLL_OUT。

## 11.2 实验

驱动程序在上一章的基础上，增加异步 IO 的实现。然后再完成对应的测试程序。

### 11.2.1 原理图

led 部分和章节 1.3.1 相同。
key 部分和章节 6.1 相同。

### 11.2.2 设备树

和章节 6.2 相同。

### 11.2.3 驱动代码

使用 petalinux 新建名为”ax-fasync-drv”的驱劢程序，并执行 petalinux-config -c rootfs 命令选上新增的驱动程序。

在 ax-fasync-drv.c 文件中输入下面的代码：

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/init.h>
4.  #include <linux/ide.h>
5.  #include <linux/types.h>
6.  #include <linux/errno.h>
7.  #include <linux/cdev.h>
8.  #include <linux/of.h>
9.  #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <linux/wait.h>
20. #include <linux/poll.h>
21. #include <linux/fcntl.h>
22. #include <asm/uaccess.h>
23. #include <asm/mach/map.h>
```

```c
#include <asm/io.h>

/* 设备节点名称 */
#define DEVICE_NAME       "fasync_led"
/* 设备号个数 */
#define DEVID_COUNT       1
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR_U
/* 次设备号 */
#define MINOR_U           0

/* 把驱动代码中会用到的数据打包进设备结构体 */
struct alinx_char_dev {
    dev_t               devid;          //设备号
    struct cdev         cdev;           //字符设备
    struct class        *class;         //类
    struct device       *device;        //设备
    struct device_node  *nd;            //设备树的设备节点
    int                 alinx_key_gpio; //gpio 号
    atomic_t            key_sts;        //记录按键状态，为 1 时被按下
    unsigned int        irq;            //中断号
    struct timer_list   timer;          //定时器
    wait_queue_head_t   wait_q_h;       //等待队列头
    struct fasync_struct *fasync;       //异步信号
};
/* 声明设备结构体 */
static struct alinx_char_dev alinx_char = {
    .cdev = {
        .owner = THIS_MODULE,
    },
};

/* 中断服务函数 */
static irqreturn_t key_handler(int irq, void *dev_in)
{
    /* 按键按下或抬起时会进入中断 */
    struct alinx_char_dev *dev = (struct alinx_char_dev *)dev_in;
    /* 开启 50 毫秒的定时器用作防抖动 */
    dev->timer.data = (volatile long)dev_in;
    mod_timer(&dev->timer, jiffies + msecs_to_jiffies(50));
    return IRQ_RETVAL(IRQ_HANDLED);
}
```

```
68.
69. /* 定时器服务函数 */
70. void timer_function(unsigned long arg)
71. {
72.     struct alinx_char_dev *dev = (struct alinx_char_dev *)arg;
73.     /* value 用于获取按键值 */
74.     unsigned char value;
75.     /* 获取按键值 */
76.     value = gpio_get_value(dev->alinx_key_gpio);
77.     if(value == 0)
78.     {
79.         /* 按键按下，状态置 1 */
80.         atomic_set(&dev->key_sts, 1);
81.         /* fasync 有没有初始化过 */
82.         if(dev->fasync)
83.         {
84.             /* 初始化过说明应用程序调用过 */
85.             kill_fasync(&dev->fasync, SIGIO, POLL_OUT);
86.         }
87.         else if((current->state & TASK_INTERRUPTIBLE) != 0)
88.         {
89.             /* 是等待队列，需要唤醒进程 */
90.             wake_up_interruptible(&dev->wait_q_h);
91.         }
92.         else
93.         {
94.             /* do nothing */
95.         }
96.     }
97.     else
98.     {
99.         /* 按键抬起 */
100.     }
101. }
102.
103. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
104. static int char_drv_open(struct inode *inode_p, struct file *file_p)
105. {
106.     printk("gpio_test module open\n");
107.     file_p->private_data = &alinx_char;
108.     return 0;
109. }
110.
111. /* read 函数实现，对应到 Linux 系统调用函数的 write 函数 */
```

```
112. static ssize_t char_drv_read(struct file *file_p, char __user *buf, size_t len, loff_t *lof
     f_t_p)
113. {
114.     unsigned int keysts = 0;
115.     int ret;
116.
117.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
118.
119.     /* 读取 key 的状态 */
120.     keysts = atomic_read(&dev->key_sts);
121.     /* 判断文件打开方式 */
122.     if(file_p->f_flags & O_NONBLOCK)
123.     {
124.         /* 如果是非阻塞访问，说明已满足读取条件 */
125.     }
126.     /* 判断当前按键状态 */
127.     else if(!keysts)
128.     {
129.         /* 按键未被按下(数据未准备好) */
130.         /* 以当前进程创建并初始化为队列项 */
131.         DECLARE_WAITQUEUE(queue_mem, current);
132.         /* 把当前进程的队列项添加到队列头 */
133.         add_wait_queue(&dev->wait_q_h, &queue_mem);
134.         /* 设置当前进成为可被信号打断的状态 */
135.         __set_current_state(TASK_INTERRUPTIBLE);
136.         /* 切换进程，是当前进程休眠 */
137.         schedule();
138.
139.         /* 被唤醒，修改当前进程状态为 RUNNING */
140.         set_current_state(TASK_RUNNING);
141.         /* 把当前进程的队列项从队列头中删除 */
142.         remove_wait_queue(&dev->wait_q_h, &queue_mem);
143.
144.         /* 判断是否是被信号唤醒 */
145.         if(signal_pending(current))
146.         {
147.             /* 如果是直接返回错误 */
148.             return -ERESTARTSYS;
149.         }
150.         else
151.         {
152.             /* 被按键唤醒 */
153.         }
154.     }
```

```
155.     else
156.     {
157.         /* 按键被按下(数据准备好了) */
158.     }
159.
160.     /* 读取 key 的状态 */
161.     keysts = atomic_read(&dev->key_sts);
162.     /* 返回按键状态值 */
163.     ret = copy_to_user(buf, &keysts, sizeof(keysts));
164.     /* 清除按键状态 */
165.     atomic_set(&dev->key_sts, 0);
166.     return 0;
167. }
168.
169. /* poll 函数实现 */
170. unsigned int char_drv_poll(struct file *file_p, struct poll_table_struct *wait)
171. {
172.     unsigned int ret = 0;
173.
174.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
175.
176.     /* 将应用程序添添加到等待队列中 */
177.     poll_wait(file_p, &dev->wait_q_h, wait);
178.
179.     /* 判断 key 的状态 */
180.     if(atomic_read(&dev->key_sts))
181.     {
182.         /* key 准备好了, 返回数据可读 */
183.         ret = POLLIN;
184.     }
185.     else
186.     {
187.
188.     }
189.
190.     return ret;
191. }
192.
193. /* fasync 函数实现 */
194. static int char_drv_fasync(int fd,struct file *file_p, int mode)
195. {
196.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
197.     return fasync_helper(fd, file_p, mode, &dev->fasync);
198. }
```

```
199.
200. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
201. static int char_drv_release(struct inode *inode_p, struct file *file_p)
202. {
203.     printk("gpio_test module release\n");
204.     return char_drv_fasync(-1, file_p, 0);
205. }
206.
207. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
208. static struct file_operations ax_char_fops = {
209.     .owner    = THIS_MODULE,
210.     .open     = char_drv_open,
211.     .read     = char_drv_read,
212.     .poll     = char_drv_poll,
213.     .fasync   = char_drv_fasync,
214.     .release = char_drv_release,
215. };
216.
217. /* 模块加载时会调用的函数 */
218. static int __init char_drv_init(void)
219. {
220.     /* 用于接受返回值 */
221.     u32 ret = 0;
222.
223.     /* 初始化原子变量 */
224.     atomic_set(&alinx_char.key_sts, 0);
225.
226.     /* 获取设备节点 */
227.     alinx_char.nd = of_find_node_by_path("/alinxkey");
228.     if(alinx_char.nd == NULL)
229.     {
230.         printk("alinx_char node not find\r\n");
231.         return -EINVAL;
232.     }
233.     else
234.     {
235.         printk("alinx_char node find\r\n");
236.     }
237.
238.     /* 获取节点中 gpio 标号 */
239.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinxkey-gpios", 0);
240.     if(alinx_char.alinx_key_gpio < 0)
241.     {
242.         printk("can not get alinxkey-gpios");
```

```
243.         return -EINVAL;
244.     }
245.     printk("alinxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
246.
247.     /* 申请 gpio 标号对应的引脚 */
248.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinxkey");
249.     if(ret != 0)
250.     {
251.         printk("can not request gpio\r\n");
252.         return -EINVAL;
253.     }
254.
255.     /* 把这个 io 设置为输入 */
256.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
257.     if(ret < 0)
258.     {
259.         printk("can not set gpio\r\n");
260.         return -EINVAL;
261.     }
262.
263.     /* 获取中断号 */
264.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
265.     /* 申请中断 */
266.     ret = request_irq(alinx_char.irq,
267.                       key_handler,
268.                       IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
269.                       "alinxkey",
270.                       &alinx_char);
271.     if(ret < 0)
272.     {
273.         printk("irq %d request failed\r\n", alinx_char.irq);
274.         return -EFAULT;
275.     }
276.
277.     alinx_char.timer.function = timer_function;
278.     init_timer(&alinx_char.timer);
279.
280.     init_waitqueue_head(&alinx_char.wait_q_h);
281.
282.     /* 注册设备号 */
283.     alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
284.
285.     /* 初始化字符设备结构体 */
286.     cdev_init(&alinx_char.cdev, &ax_char_fops);
```

```
287.
288.     /* 注册字符设备 */
289.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
290.
291.     /* 创建类 */
292.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
293.     if(IS_ERR(alinx_char.class))
294.     {
295.         return PTR_ERR(alinx_char.class);
296.     }
297.
298.     /* 创建设备节点 */
299.     alinx_char.device = device_create(alinx_char.class, NULL,
300.                                       alinx_char.devid, NULL,
301.                                       DEVICE_NAME);
302.     if (IS_ERR(alinx_char.device))
303.     {
304.         return PTR_ERR(alinx_char.device);
305.     }
306.
307.     return 0;
308. }
309.
310. /* 卸载模块 */
311. static void __exit char_drv_exit(void)
312. {
313.     /* 释放 gpio */
314.     gpio_free(alinx_char.alinx_key_gpio);
315.
316.     /* 释放中断 */
317.     free_irq(alinx_char.irq, NULL);
318.
319.     /* 删除定时器 */
320.     del_timer_sync(&alinx_char.timer);
321.
322.     /* 注销字符设备 */
323.     cdev_del(&alinx_char.cdev);
324.
325.     /* 注销设备号 */
326.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
327.
328.     /* 删除设备节点 */
329.     device_destroy(alinx_char.class, alinx_char.devid);
330.
```

```
331.     /* 删除类 */
332.     class_destroy(alinx_char.class);
333. 
334.     printk("timer_led_dev_exit_ok\n");
335. }
336. 
337. /* 标记加载、卸载函数 */
338. module_init(char_drv_init);
339. module_exit(char_drv_exit);
340. 
341. /* 驱动描述信息 */
342. MODULE_AUTHOR("Alinx");
343. MODULE_ALIAS("alinx char");
344. MODULE_DESCRIPTION("FASYNC LED driver");
345. MODULE_VERSION("v1.0");
346. MODULE_LICENSE("GPL");
```

21 行添加头文件 linux/fcntl.h。

49 行添加 fasync_struct 结构体到设备结构体中。

82 行在定时器的处理函数中，确认按键按下后，先判断 fasync_struct 结构体有没有初始化过，初始化过说明调用过 fasync 函数，也就是应用程序是能了异步通知，所以就发送对应的信号。

204~208 行实现 fasync 函数，里面就是简单的调用了 fasync_helper 来初始化 fasync_struct 结构体。

204 行在 release 函数中调用 fasync 函数，释放 fasync_struct 结构体。

213 行把我们实现的 fasync 函数添加到 file_operations 结构体中。

另外，相比较上一章，这章里面使用了私有数据来代替设备结构体变量的全局变量，具体的用法说明，可以回顾一下第一章的实验代码里的解释。

注意 270 行中断的服务函数输入参数设置和 64 行定时器的服务函数输入参数设置。

## 11.2.4 测试代码

新建 QT 工程名为”ax_fasync_test”，新建 main.c，输入下列代码：

```
1.  #include "stdio.h"
2.  #include "unistd.h"
3.  #include "sys/types.h"
4.  #include "sys/stat.h"
5.  #include "fcntl.h"
6.  #include "stdlib.h"
7.  #include "string.h"
8.  #include "poll.h"
9.  #include "sys/select.h"
10. #include "sys/time.h"
```

```c
#include "linux/ioctl.h"
#include "signal.h"

static int fd = 0, fd_l = 0;

static void sigio_signal_func()
{
    int ret = 0;
    static char led_value = 0;
    unsigned int key_value;

    /* 获取按键状态 */
    ret = read(fd, &key_value, sizeof(key_value));
    if(ret < 0)
    {
        printf("read failed\r\n");
    }

    /* 判断按键状态 */
    if(1 == key_value)
    {
        /* 按键被按下，改变吗 led 状态 */
        printf("ps_key1 press\r\n");
        led_value = !led_value;

        fd_l = open("/dev/gpio_leds", O_RDWR);
        if(fd_l < 0)
        {
            printf("file /dev/gpio_leds open failed\r\n");
        }

        ret = write(fd_l, &led_value, sizeof(led_value));
        if(ret < 0)
        {
            printf("write failed\r\n");
        }

        ret = close(fd_l);
        if(ret < 0)
        {
            printf("file /dev/gpio_leds close failed\r\n");
        }
    }
}
```

```
55.
56. int main(int argc, char *argv[])
57. {
58.     int flags = 0;
59.     char *filename;
60.
61.     if(argc != 2)
62.     {
63.         printf("wrong para\n");
64.         return -1;
65.     }
66.
67.     filename = argv[1];
68.     fd = open(filename, O_RDWR);
69.     if(fd < 0)
70.     {
71.         printf("can not open file %s\r\n", filename);
72.         return -1;
73.     }
74.
75.     /* 指定信号 SIGIO，并绑定处理函数 */
76.     signal(SIGIO, sigio_signal_func);
77.     /* 把当前线程指定为将接收信号的进程 */
78.     fcntl(fd, F_SETOWN, getpid());
79.     /* 获取当前线程状态 */
80.     flags = fcntl(fd, F_GETFD);
81.     /* 设置当前线程为 FASYNC 状态 */
82.     fcntl(fd, F_SETFL, flags | FASYNC);
83.
84.     while(1)
85.     {
86.         sleep(2);
87.     }
88.
89.     close(fd);
90.     return 0;
91. }
```

注意 19 行给 led_value 加上 static。
75~82 行就是按照 11.1.2 节中说的步骤操作即可，不重复解释了。

## 11.2.5 运行测试

测试方式和现象和上一章一样，步骤如下：

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concled-drv.ko
insmod ./ax-fasync-drv.ko
cd ./build-ax_fasync_test-ZYNQ-Debug
./ax_fasync_test /dev/fasync_led&
top
```

结果如下图：

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-concled-drv.ko
ax_concled_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt# insmod ./ax-fasync-drv.ko
alinx_char node find
alinxkey-gpio num = 949
root@ax_peta:/mnt# cd ./build-ax_fasync_test-ZYNQ-Debug
root@ax_peta:/mnt/build-ax_fasync_test-ZYNQ-Debug# ./ax_fasync_test /dev/fasync_led&
[1] 1266
root@ax_peta:/mnt/build-ax_fasync_test-ZYNQ-Debug# gpio_test module open
ps_key1 press
gpio_test module open
gpio_test module release
ps_key1 press
gpio_test module open
gpio_test module release
```

```
Mem: 57072K used, 973180K free, 13408K shrd, 1196K buff, 43064K cached
CPU:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.00 0.00 0.00 1/87 1267
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 1242     1 root     S    14352  1.3   1  0.0 /usr/sbin/tcf-agent -d -L- -l0
 1266  1249 root     S    13164  1.2   1  0.0 ./ax_fasync_test /dev/fasync_led
 1249  1247 root     S     2988  0.2   1  0.0 -sh
  756     1 root     S     2972  0.2   1  0.0 /sbin/udevd -d
 1169     1 root     S     2916  0.2   0  0.0 /usr/sbin/inetd
 1228     1 root     S     2788  0.2   0  0.0 /sbin/syslogd -n -O /var/log/messages
 1231     1 root     S     2788  0.2   1  0.0 /sbin/klogd -n
 1267  1249 root     R     2788  0.2   1  0.0 top
 1213     1 root     S     2788  0.2   0  0.0 udhcpc -R -b -p /var/run/udhcpc.eth0.pid -i eth0
 1248     1 root     S     2788  0.2   1  0.0 /sbin/getty 38400 ttyl
 1247     1 root     S     2756  0.2   0  0.0 {start_getty} /bin/sh /bin/start_getty 115200 ttyP
 1220     1 root     S     2364  0.2   0  0.0 /usr/sbin/dropbear -r /etc/dropbear/dropbear_rsa_h
    1     0 root     S     1700  0.1   1  0.0 init
 1254     2 root     SW<      0  0.0   0  0.0 [kworker/0:1H]
```