UnixWorld Online: Tutorial Article No. 010 Listing

Listing 1. A user program that employs system calls.

```
#include "stdio.h"
#include "sys/types.h"
#include "unistd.h"
#include "fcntl.h"

void
main() {

    int fd;        /* file descriptor for the device */
    char buff[16]; /* string of 16 bytes */

    if ((fd = open("/dev/xxx", O_RDWR)) < 0) {
       fprintf(stderr, "Can't open: "); perror("/dev/xxx"); exit(1);
    } else {
       printf("/dev/xxx opened with fd = %d\n", fd);
    }

    printf("size of buff is %d\n", sizeof(buff));
    printf("read returns %d\n", read(fd, buff, sizeof(buff)));

    buff[15] = 0;    /* null terminate the string */

    /* display the data read into the buffer */
    printf("buff is:\n%s\n", buff);
    close(fd);
}
```

Listing 2. The `file_operations` structure and device initialization.

```
struct file_operations xxx_fops = {
    NULL,          /* lseek()   */
    xxx_read,      /* read()    */
    xxx_write,     /* write()   */
    NULL,          /* readdir() */
    NULL,          /* select()  */
    xxx_ioctl,     /* ioctl()   */
    NULL,          /* mmap()    */
    xxx_open,      /* open()    */
    xxx_close      /* close()   */
    };

long
xxx_init(long kmem_start) {
    printk("Sample Device Driver Initialization\n");

    if (register_chrdev(22, "xxx", &xxx_fops))
        printk("error--cannot register to major device 22!\n");
    /* detect hardware and initialize it */
```

```
    return kmem_start;
}
```

Listing 3. The `xxx_write()` routine from a character device driver.

```
/* system include files */
#include "linux/kernel.h"
#include "linux/sched.h"
#include "linux/tty.h"
#include "linux/signal.h"
#include "linux/errno.h"
#include "asm/io.h"
#include "asm/segment.h"
#include "asm/system.h"
#include "asm/irq.h"

static int
xxx_write(struct inode *inode, struct file *file, char *buffer, int count) {

    unsigned int minor = MINOR(inode->i_rdev); /* minor number of device */
    int          offset = 0;
    char         ret;

    if (count > 4095)
        return(-ENOMEM);
    if (count <= 0)
        return(-EINVAL);

    while (count > 0) {
        ret = xxx_write_byte(minor);
        if (ret < 0) {
            xxx_handle_error(WRITE, ret, minor);
            continue;
        }
        buff++ = ret; offset++;
    }
    return offset; /* return number of bytes written */
    /* xxx_write_byte() and xxx_handle_error() are functions
       defined elsewhere in xxx_drv.c */
}
```

Listing 4. The `xxx_write()` routine for an interrupt-driven character-device driver.

```
/* register definitions */
#define XXX_BUFFER_SIZE 1024
#define XXX_INTERRUPT_TIMEOUT 100

static int
xxx_write(struct inode *inode, struct file *file, char *buffer, int count) {

    unsigned int minor = MINOR(inode->i_rdev);     /* minor number of device */
    unsigned long copy_size, bytes_written, total_bytes_written = 0;

    struct xxx_struct *xxx = &xxx_table[minor];
```

```
    do {
        copy_size = (count <= XXX_BUFFER_SIZE ? count : XXX_BUFFER_SIZE);
        memcpy_fromfs(xxx->xxx_buffer, buf, copy_size);   /* move the data */

        while (copy_size) {     /* while there is data in the buffer */

            /* initiate interrupts */
            if (error_occurred) {     /* handle error condition */ }
            current_timeout = jiffies + XXX_INTERRUPT_TIMEOUT;

            / * set timeout in case interrupt is missed */
            interruptible_sleep_on(&xxx_wait_queue);

            /* defined in , xxx_wait_queue is a FIFO queue of
            processes that are sleeping; once initialized (to NULL) the kernel
            handles the queue and to access it, then supply a pointer */

            bytes_written = xxx->bytes_xfered;
            xxx->bytes_written = 0;
            if (current->signal & ~current->blocked) {
                if (total_bytes_written + bytes_written)
                    return(total_bytes_written + bytes_written);
                else    return(-EINTR); /* system call interrupted, try again */
                }
        }
        total_bytes_written += bytes_written;
        buf += bytes_written;
        count -= bytes_written;
    } while (count > 0);

    return(total_bytes_written); /* bytes written on a call */
}

static void xxx_interrupt(int irq) {

    struct xxx_struct *xxx = &xxx_table[xxx_irq[irq]];

    /* execute interrupt actions, check flag in xxx_table to see
       if reading or writing */

    /* increment xxx->bytes_xfered by however many characters transferred */
    if (buffer too full/empty)
        wake_up_interruptible(&xxx->xxx_wait_queue);
}
```

Listing 5. A Bus Mouse Device Driver

Notes: The file mouse.c contains the code to call mouse-dependent routines, which can be found, for example, in file `busmouse.c`. The main points to extract from review of this driver are an understanding of basic driver structure.

Line 23 (of file `mouse.c`) declares the head of the doubly linked `mouse_list` structure. Line 26 is the start of the generic routine to open all mouse devices. Line 30 retrieves the minor number of the device. Line 39 calls the mouse-specific open routine. Line 81 calls the mouse-specific initialization routine. Line 88 registers the device and it s entry points with the VFS.

```
  1  /* Filename:    linux/drivers/char/mouse.c
  2   * Description: generic mouse open routine
```

```
 * the independent mouse drivers are independent
 */

#ifdef MODULE
#include
#include
#else
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif

#include
#include
#include
#include
#include
#include
#include

/* head entry for the doubly linked mouse list */
static struct mouse mouse_list = {0, "head", NULL, &mouse_list, &mouse_list};

/* include declarations of the various mouse initialization routines */
extern int bus_mouse_init(void);

/* generic routine to open all mouse devices */
static int mouse_open(struct inode * inode, struct file * file) {
    int minor = MINOR(inode->i_rdev);
    struct mouse *c = mouse_list.next;
    file->f_op = NULL;
    while (c != &mouse_list) {
        if (c->minor == minor) {
            file->f_op = c->fops; break;
        }
        c = c->next;
    }
    if (file->f_op == NULL) return(-ENODEV); /* no device found */

    /* call the mouse-dependent open routine */
    return file->f_op->open(inode, file);
}

static struct file_operations mouse_fops = {
    NULL,   /* seek    */
    NULL,   /* read    */
    NULL,   /* write   */
    NULL,   /* readdir */
    NULL,   /* select  */
    NULL,   /* ioctl   */
    NULL,   /* mmap    */
    mouse_open,
    NULL    /* release */
};

int mouse_register(struct mouse * mouse)
{
    if (mouse->next || mouse->prev)
        return(-EBUSY);
    MOD_INC_USE_COUNT;
    mouse->next = &mouse_list;
    mouse->prev = mouse_list.prev;
```

```
    mouse->prev->next = mouse;
    mouse->next->prev = mouse;
    return 0;
}

int mouse_deregister(struct mouse * mouse)
{
    if (!mouse->next || !mouse->prev)
        return (-EINVAL);
    MOD_DEC_USE_COUNT;
    mouse->prev->next = mouse->next; /* removing mouse from list */
    mouse->next->prev = mouse->prev;
    mouse->next = NULL;
    mouse->prev = NULL;
    return 0;
}

int mouse_init(void)
{

#ifdef CONFIG_BUSMOUSE
bus_mouse_init();  /* called to initialize the proper mouse device */
#endif

    if (register_chrdev(MOUSE_MAJOR, "mouse", &mouse_fops)) {
        printk("unable to get major number %d for mouse devices\n",
            MOUSE_MAJOR);
        return(-EIO);
    }
    return 0;
}

void cleanup_module(void)
{
    if (MOD_IN_USE) { /* mouse is being used */
        printk("mouse: in use, remove delayed \n");
        return;
    }
    unregister_chrdev(MOUSE_MAJOR, "mouse");
}
```

Notes: Files named `busmouse.h` and `busmouse.c` contain code that is spscific to a particular busmouse, for example the Logitech bus mouse. Lines 29 -39 (of file busmouse.h) define a structure to keep track of the mouse. Line 37 defines the wait queue structure that the device uses to sleep on.

```
 1  /*
 2   * Filename:    linux/drivers/char/busmouse.h
 3   */
 4
 5  #define    MOUSE_IRQ   5
 6  #define    LOGITECH_BUSMOUSE       0  /* Minor device num for Logitech  */
 7  #define    MICROSOFT_BUSMOUSE      2  /* Minor device num for Microsoft */
 8  #define    MSE_DATA_PORT           0x23c
 9  #define    MSE_SIGNATURE_PORT      0x23d
10  #define    MSE_CONTROL_PORT        0x23e
11  #define    MSE_INTERRUPT_PORT      0x23e
12  #define    MSE_CONFIG_PORT         0x23f
13  #define    MSE_ENABLE_INTERRUPTS   0x00
14  #define    MSE_DISABLE_INTERRUPTS  0x10
15  #define    MSE_READ_X_LOW          0x80
```

```
#define    MSE_READ_X_HIGH         0xa0
#define    MSE_READ_Y_LOW          0xc0
#define    MSE_READ_Y_HIGH         0xe0

/* magic number used to check if the mouse exists */
#define    MSE_CONFIG_BYTE         0x91
#define    MSE_DEFAULT_MODE        0x90
#define    MSE_SIGNATURE_BYTE      0xa5

/* useful Logitech Mouse macros */
#define    MSE_INT_OFF()    outb(MSE_DISABLE_INTERRUPTS, MSE_CONTROL_PORT)
#define    MSE_INT_ON()     outb(MSE_ENABLE_INTERRUPTS,  MSE_CONTROL_PORT)

struct mouse_status {
    unsigned char        buttons;
    unsigned char        latch_buttons;
    int                  dx;
    int                  dy;
    int                  present;
    int                  ready;
    int                  active;
    struct wait_queue    *wait;
    struct fasync_struct *fasyncptr;
};
```

Busmouse.c Listing

Notes: This file contains the actual hardware dependent routines to detect, initialize, and manage a busmouse. At system startup time, mem_init() calls mouse_init(), which in turn calls bus_mouse_init(). Lines 178 to 186 verify that the mouse actually exists, then lines 187 to 198 initializes the mouse's state, but leaves interrupts disabled. This routine is only called once.

Interrupts for the mouse are not enabled until the open_mouse() routine is invoked. This routine is accessed when a user process opens `/dev/mouse`. Line 88 checks to see if the mouse is present, and exits if bus_mouse_init() did not find it during system startup. Line 89 checks to see if the driver is already open, and if so the call returns and the driver exits. However, if the device is present and is not open, lines 90 to 100 register the interrupt handler and enable interrupts on the mouse device.

The interrupt service routine starts on line 29. First, line 34 disables other mouse interrupts. Then the mouse position and mouse button status is obtained from the mouse device. These values are adjusted for mouse randomness and then used to update the driver's notion of the mouse position. Line 52 wakes up any processes sleeping on the wait queue. Lines 56-59 cap the driver's mouse-position values. Line 60 generate SIGIO signals for processes that have requested them. Line 62 enables mouse interrupts before the interrupt service routine returns.

The read_mouse() routine starts on line 110 and continues to line 143. Line 116 returns an error to the user process if a read of less than 3 bytes is attempted. Attempts to read more than 3 bytes result in zero padding. An interesting feature of this routine is that the user process never sleeps on this device. Line 119 of the mouse driver returns EAGAIN to the user if the mouse has no data. Most drivers typically would call sleep_on() until data became available, causing the process to sleep. On line 117, the verify_area() kernel routine is called to ensure that the user process has provided a valid pointer into its address space. It is very important that the driver perform checking of its parameters, as failing to do so can cause security holes or system crashes. Lines 137 to 141 utilize the put_user() kernel function to copy the mouse data into the user- process address space.

Writing to the mouse driver is meaningless, so lines 104 to 107 return an error to the user process if the write_mouse() routine was called.

The mouse_select() routine on lines 147 to 169 allows user processes to use the select(2) system call on mouse devices. In contrast to the mouse_read() routine, a user process can sleep on the mouse device becoming ready during this call, although this occurrence is pretty infrequent. The only time the mouse driver is not ready is between the time the first mouse_open() is called and the time the first interrupt actually occurs.

The fasync_mouse() routine on lines 65 to 71 allow user processes to indicate that they wish to receive SIGIO signals when data is available.

The close_mouse() routine on lines 74 to 82 is called when a user process closes the mouse device or the process exits. Line 77 cancels the reception of SIGIO signals for the calling process. Line 78 returns immediately if the driver is still active. If the driver is not active (in use), lines 79 and 80 disable mouse interrupts and unregister the mouse interrupt handler.

```
 1  /*
 2   * Filename:    linux/drivers/char/busmouse.c
 3   * Description:    Logitech Bus Mouse Driver for Linux
 4   */
 5
 6  #define  MOD_INC_USE_COUNT
 7  #define  MOD_DEC_USE_COUNT
 8  #include
 9  #include
10  #include
11  #include
12  #include
13  #include
14  #include
15  #include
16  #include
17  #include
18  #include
19  #include
20
21  static struct mouse_status mouse;
22  static int mouse_irq = MOUSE_IRQ;
23
24  void bmouse_setup(char *str, int *ints)
25  {
26      if (ints[0] > 0) mouse_irq=ints[1];
27  }
28
29  static void mouse_interrupt(int irq, struct pt_regs *regs)
30  {
31      char          dx, dy;
32      unsigned char buttons;
33
34      MSE_INT_OFF();
35      outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
36      dx = (inb(MSE_DATA_PORT) & 0xf);
37      outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
```

```
    dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
    outb(MSE_READ_Y_LOW, MSE_CONTROL_PORT );
    dy = (inb(MSE_DATA_PORT) & 0xf);
    outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
    buttons = inb(MSE_DATA_PORT);
    dy |= (buttons & 0xf) << 4;
    buttons = ((buttons >> 5) & 0x07);

    if (dx != 0 || dy != 0 || buttons != mouse.buttons) {
        add_mouse_randomness((buttons << 16) + (dy << 8) + dx);
        mouse.buttons = buttons;
        mouse.dx += dx;
        mouse.dy -= dy;
        mouse.ready = 1;
        wake_up_interruptible(&mouse.wait);

        /* keep dx/dy reasonable, but still able to track when
           X pages or is busy (long waits between reads) */
        if (mouse.dx < -2048)   mouse.dx = -2048;
        if (mouse.dx >  2048)   mouse.dx =  2048;
        if (mouse.dy < -2048)   mouse.dy = -2048;
        if (mouse.dy >  2048)   mouse.dy =  2048;
        if (mouse.fasyncptr) kill_fasync(mouse.fasyncptr, SIGIO);
    }
    MSE_INT_ON();
}

static int
fasync_mouse(struct inode *inode, struct file *filp, int on)
{
    int retval = fasync_helper(inode, filp, on, &mouse.fasyncptr);
    if (retval < 0)  return retval;
    return 0;
}

/* close access to the mouse */
static void
close_mouse(struct inode *inode, struct file *file)
{
    fasync_mouse(inode, file, 0);
    if (--mouse.active)  return;
    MSE_INT_OFF();
    free_irq(mouse_irq);
    MOD_DEC_USE_COUNT;
}

/* open access to the mouse */
static int
open_mouse(struct inode *inode, struct file *file)
{
    if (!mouse.present)  return(-EINVAL);
    if (mouse.active++)  return(0);
    if (request_irq(mouse_irq, mouse_interrupt, 0, "Busmouse")) {
        mouse.active--;
        return(-EBUSY);
    }
    mouse.ready = 0;
    mouse.dx = mouse.dy = 0;
    mouse.buttons = 0x87;
    MOD_INC_USE_COUNT;
    MSE_INT_ON();
```

```
    return 0;
}

/* writes are disallowed */
static int
write_mouse(struct inode *inode, struct file *file, const char *buffer, int cou
{
    return(-EINVAL);
}

/* read mouse data - currently it never blocks */
static int
read_mouse(struct inode *inode, struct file *file, char *buffer, int count)
{
    int           r, dx, dy;
    unsigned char buttons;

    if (count < 3) return(-EINVAL);
    if ((r = verify_area(VERIFY_WRITE, buffer, count)))
        return r;
    if (!mouse.ready)  return(-EAGAIN);

    /* obtain the current mouse params and limit as appropriate
       for the return data format.  Interrupts are disabled while
       obtaining the params, NOT during the puts_fs_byte() calls,
       so paging in put_user() not effect mouse tracking
     */
    MSE_INT_OFF();
    dx = mouse.dx;  dy = mouse.dy;
    if (dx < -127)  dx = -127;
    if (dx >  127)  dx =  127;
    if (dy < -127)  dy = -127;
    if (dy >  127)  dy =  127;
    buttons = mouse.buttons;
    mouse.dx -= dx;
    mouse.dy -= dy;
    mouse.ready = 0;
    MSE_INT_ON();
    put_user(buttons | 0x80, buffer);
    put_user((char)dx, buffer + 1);
    put_user((char)dy, buffer + 2);
    for (r = 3; r < count; r++)
        put_user(0x00, buffer + r);
    return(r);
}

/* select for mouse input */
static int
mouse_select(struct inode *inode, struct file *file, int sel_type, select_table
{
    if (sel_type == SEL_IN) {
        if (mouse.ready)
            return(1);
        select_wait(&mouse.wait, wait);
    }
    return(0);
}

struct file_operations bus_mouse_fops = {
    NULL,          /* mouse_seek */
    read_mouse,
```

```
    write_mouse,
    NULL,          /* mouse_readdir */
    mouse_select,  /* mouse_select */
    NULL,          /* mouse_ioctl */
    NULL,          /* mouse_mmap */
    open_mouse,
    close_mouse,
    NULL,
    fasync_mouse,
};

static struct mouse bus_mouse = {
    LOGITECH_BUSMOUSE, "busmouse", &bus_mouse_fops
};

int bus_mouse_init(void)
{
    int i;
    outb(MSE_CONFIG_BYTE, MSE_CONFIG_PORT);
    outb(MSE_SIGNATURE_BYTE, MSE_SIGNATURE_PORT);

    for (i = 0; i < 100000; i++)  /* busy loop */;

    if (inb(MSE_SIGNATURE_PORT) != MSE_SIGNATURE_BYTE) {
        mouse.present = 0;
        return(-EIO);
    }
    outb(MSE_DEFAULT_MODE, MSE_CONFIG_PORT);
    MSE_INT_OFF();
    mouse.present = 1;
    mouse.active = mouse.ready = 0;
    mouse.buttons = 0x87;
    mouse.dx = mouse.dy = 0;
    mouse.wait = NULL;
    printk("Logitech Bus mouse detected and installed with IRQ %d.\n",
        mouse_irq);
    mouse_register(&bus_mouse);
    return 0;
}

void cleanup_module(void)
{
    if (MOD_IN_USE)
        printk("busmouse: in use - remove delayed\n");
    mouse_deregister(&bus_mouse);
}
```

Edited by Becca Thomas / Online Editor / UnixWorld Online / editor@unixworld.com

Go To Content   Search

Technical review and some commentary kindly provided by Stephen Degler.
Last Modified: Sunday, 21-Jan-96 06:05:09 PST