Modules

Table des matières

cf The Linux Kernel Module Programming Guide.
Plus de doc dans les sources su noyau (ensuite) :

```
# cd /usr/src/linux
# make help
# make htmldocs
```

1 Hello world

1.1 The simplest kernel module

- code

```
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>/* Needed by all modules */
#include <linux/kernel.h>/* Needed for KERN_INFO */

int init_module(void)
{
printk(KERN_INFO "Hello world 1.\n");

/*
 * A non 0 return means init_module failed; module can't be loaded.
 */
return 0;
}
```

```
void cleanup_module(void)
{
printk(KERN_INFO "Goodbye world 1.\n");
}
```

- linux headers

```
# apt-get install linux-headers-$(uname -r)
# apt-get source linux-image-2.6-486         # (facultatif)
```

- compilation cf */usr/src/linux-src/Documentation/kbuild/makefile.txt* :

```
$ echo 'obj-m := hello-1.o' > Kbuild
$ make -C /lib/modules/‘uname -r‘/build M=$PWD modules
```

- test

```
# modinfo hello-1.ko
# insmod hello-1.ko
$ lsmod | grep hello
$ cat /proc/modules | grep^hello
# rmmod hello-1.ko
$ dmesg | tail -n 5
```

- installation **Marche pas !**

```
# make -C /lib/modules/‘uname -r‘/build M=$PWD modules_install
# modprobe hello-1 (marche pas)
```

1.2 module_init() and module_exit() macros

- code

```
/*
 *  hello-3.c - Demonstrating the module_init() and module_exit() macros.
 *  This is preferred over using init_module() and cleanup_module().
 */
#include <linux/module.h>/* Needed by all modules */
#include <linux/kernel.h>/* Needed for KERN_INFO */
#include <linux/init.h>/* Needed for the macros */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
printk(KERN_INFO "Hello, world 2\n");
return 0;
}

static void __exit hello_3_exit(void)
{
printk(KERN_INFO "Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

- compilation

```
$ echo 'obj-m += hello-3.o' >> Kbuild
$ make -C /lib/modules/‘uname -r‘/build M=$PWD modules
```

1.3 module documentation

- code

```
/*
 *  hello-4.c - Demonstrates module documentation.
 */
#include <linux/module.h>/* Needed by all modules */
#include <linux/kernel.h>/* Needed for KERN_INFO */
#include <linux/init.h>/* Needed for the macros */
#define DRIVER_AUTHOR "Bibi <bibi@bibalse.org>"
#define DRIVER_DESC   "A megaone driver"

static int __init init_hello_4(void)
{
printk(KERN_INFO "Hello, world 4\n");
return 0;
}

static void __exit cleanup_hello_4(void)
{
printk(KERN_INFO "Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
 *  You can use strings, like this:
 */

/*
 * Get rid of taint message by declaring code as GPL.
 */
MODULE_LICENSE("GPL");

/*
 * Or with defines, like this:
 */
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */

/*
 *  This module uses /dev/testdevice.  The MODULE_SUPPORTED_DEVICE macro might
 *  be used in the future to help automatic configuration of modules, but is
 *  currently unused other than for documentation purposes.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

- test

```
# modinfo hello-4.ko
```

1.4 command line argument passing

- code

```
/*
 *  hello-5.c - Demonstrates command line argument passing to a module.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
static int myintArray[2] = { -1, -1 };
static int arr_argc = 0;

/*
 * module_param(foo, int, 0000)
 * The first param is the parameters name
 * The second param is it's data type
 * The final argument is the permissions bits,
 * for exposing parameters in sysfs (if non-zero) at a later stage.
 */

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

/*
 * module_param_array(name, type, num, perm);
 * The first param is the parameter's (in this case the array's) name
 * The second param is the data type of the elements of the array
 * The third argument is a pointer to the variable that will store the number
 * of elements of the array initialized by the user at module loading time
 * The fourth argument is the permission bits
 */
module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");

static int __init hello_5_init(void)
```

```
{
int i;
printk(KERN_INFO "Hello, world 5\n=============\n");
printk(KERN_INFO "myshort is a short integer: %hd\n", myshort);
printk(KERN_INFO "myint is an integer: %d\n", myint);
printk(KERN_INFO "mylong is a long integer: %ld\n", mylong);
printk(KERN_INFO "mystring is a string: %s\n", mystring);
for (i = 0; i < (sizeof myintArray / sizeof (int)); i++)
{
printk(KERN_INFO "myintArray[%d] = %d\n", i, myintArray[i]);
}
printk(KERN_INFO "got %d arguments for myintArray.\n", arr_argc);
return 0;
}

static void __exit hello_5_exit(void)
{
printk(KERN_INFO "Goodbye, world 5\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

- test

```
# insmod hello-5.ko mystring="bebop" myint=255 myintArray=-1
# insmod hello-5.ko mystring="supercalifragilisticexpialidocious" \
myint=256 myintArray=-1,-1
# insmod hello-5.ko mylong=hello
```

1.5 multi filed modules

- start.c

```
/*
 *  start.c - Illustration of multi filed modules
 */

#include <linux/kernel.h>/* We're doing kernel work */
#include <linux/module.h>/* Specifically, a module */

int init_module(void)
{
printk(KERN_INFO "Hello, world - this is the kernel speaking\n");
return 0;
}
```

- stop.c

```
/*
 *  stop.c - Illustration of multi filed modules
 */

#include <linux/kernel.h>/* We're doing kernel work */
```

```
#include <linux/module.h>/* Specifically, a module  */

void cleanup_module()
{
printk(KERN_INFO "Short is the life of a kernel module\n");
}
```

- compilation

```
$ cat >> Kbuild <<EOF
obj-m += startstop.o
startstop-objs := start.o stop.o
EOF
$ make -C /lib/modules/‘uname -r‘/build M=$PWD modules
```

2 Character device drivers

- chardev.c

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you’ve read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>/* for put_user */

/*
 *  Prototypes - this would normally go in a .h file
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80 /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major; /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?
 * Used to prevent multiple access to device */
static char msg[BUF_LEN]; /* The msg the device will give when asked */
static char *msg_Ptr;
```

```
static struct file_operations fops = {
.read = device_read,
.write = device_write,
.open = device_open,
.release = device_release
};

/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

if (Major < 0) {
  printk(KERN_ALERT "Registering char device failed with %d\n", Major);
  return Major;
}

printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
printk(KERN_INFO "the driver, create a dev file with\n");
printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
printk(KERN_INFO "the device file.\n");
printk(KERN_INFO "Remove the device file and module when done.\n");

return SUCCESS;
}

/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
/*
 * Unregister the device
 */
unregister_chrdev(Major, DEVICE_NAME);
}

/*
 * Methods
 */

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
static int counter = 0;

if (Device_Open)
return -EBUSY;
```

```
Device_Open++;
sprintf(msg, "I already told you %d times Hello world!\n", counter++);
msg_Ptr = msg;
try_module_get(THIS_MODULE);

return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
Device_Open--; /* We're now ready for our next caller */

/*
 * Decrement the usage count, or else once you opened the file, you'll
 * never get get rid of the module.
 */
module_put(THIS_MODULE);

return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,/* see include/linux/fs.h   */
   char *buffer,/* buffer to fill with data */
   size_t length,/* length of the buffer     */
   loff_t * offset)
{
/*
 * Number of bytes actually written to the buffer
 */
int bytes_read = 0;

/*
 * If we're at the end of the message,
 * return 0 signifying end of file
 */
if (*msg_Ptr == 0)
return 0;

/*
 * Actually put the data into the buffer
 */
while (length && *msg_Ptr) {

/*
 * The buffer is in the user data segment, not the kernel
 * segment so "*" assignment won't work.  We have to use
```

```
 * put_user which copies data from the kernel data segment to
 * the user data segment.
 */
put_user(*(msg_Ptr++), buffer++);

length--;
bytes_read++;
}

/*
 * Most read functions return the number of bytes put into the buffer
 */
return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
return -EINVAL;
}
```

- Kbuild

```
obj-m := chardev.o'
```

- Makefile

```
all:
        make -C /lib/modules/`uname -r`/build M=$(shell pwd) modules

clean:
        @rm -f *{.mod.c,.cmd*,.o,.ko}
        @rm -f .*o.cmd .*.o.d
        @rm -f modules.order Module.symvers
```

- Tests

```
# insmod chardev.ko
# tail -f /var/log/syslog
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] I was assigned major number 251. To talk to
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] the driver, create a dev file with
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] 'mknod /dev/chardev c 251 0'.
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] Try various minor numbers. Try to cat and e
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] the device file.
Mar 17 17:10:39 lpnhews5234 kernel: [112590.348002] Remove the device file and module when done

# mknod /dev/chardev c 251
# cat /dev/chardev
I already told you 4 times Hello world!
```

```
# echo "coucou" > /dev/chardev
bash: echo: write error: Argument invalide

# rmmod chardev.ko
# rm /dev/chardev
```

3 The */proc* File System

3.1 create a "file" in */proc*

- procfs1.c

```
/*
 *  procfs1.c -  create a "file" in /proc
 *
 */

#include <linux/module.h>/* Specifically, a module */
#include <linux/kernel.h>/* We're doing kernel work */
#include <linux/proc_fs.h>/* Necessary because we use the proc fs */

#define procfs_name "helloworld"

/**
 * This structure hold information about the /proc file
 *
 */
struct proc_dir_entry *Our_Proc_File;

/* Put data into the proc fs file.
 *
 * Arguments
 * =========
 * 1. The buffer where the data is to be inserted, if
 *    you decide to use it.
 * 2. A pointer to a pointer to characters. This is
 *    useful if you don't want to use the buffer
 *    allocated by the kernel.
 * 3. The current position in the file
 * 4. The size of the buffer in the first argument.
 * 5. Write a "1" here to indicate EOF.
 * 6. A pointer to data (useful in case one common
 *    read for multiple /proc/... entries)
 *
 * Usage and Return Value
 * ======================
 * A return value of zero means you have no further
 * information at this time (end of file). A negative
 * return value is an error condition.
 *
 * For More Information
 * ====================
 * The way I discovered what to do with this function
 * wasn't by reading documentation, but by reading the
```

```
 * code which used it. I just looked to see what uses
 * the get_info field of proc_dir_entry struct (I used a
 * combination of find and grep, if you're interested),
 * and I saw that  it is used in <kernel source
 * directory>/fs/proc/array.c.
 *
 * If something is unknown about the kernel, this is
 * usually the way to go. In Linux we have the great
 * advantage of having the kernel source code for
 * free - use it.
 */
int
procfile_read(char *buffer,
      char **buffer_location,
      off_t offset, int buffer_length, int *eof, void *data)
{
int ret;

printk(KERN_INFO "procfile_read (/proc/%s) called\n", procfs_name);

/*
 * We give all of our information in one go, so if the
 * user asks us if we have more information the
 * answer should always be no.
 *
 * This is important because the standard read
 * function from the library would continue to issue
 * the read system call until the kernel replies
 * that it has no more information, or until its
 * buffer is filled.
 */
if (offset > 0) {
/* we have finished to read, return 0 */
ret  = 0;
} else {
/* fill the buffer, return the buffer size */
ret = sprintf(buffer, "HelloWorld!\n");
}

return ret;
}

int init_module()
{
Our_Proc_File = create_proc_entry(procfs_name, 0644, NULL);

if (Our_Proc_File == NULL) {
remove_proc_entry(procfs_name, &proc_root);
printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
       procfs_name);
return -ENOMEM;
}

Our_Proc_File->read_proc = procfile_read;
```

```
Our_Proc_File->owner   = THIS_MODULE;
Our_Proc_File->mode   = S_IFREG | S_IRUGO;
Our_Proc_File->uid   = 0;
Our_Proc_File->gid   = 0;
Our_Proc_File->size   = 37;

printk(KERN_INFO "/proc/%s created\n", procfs_name);
return 0; /* everything is ok */
}

void cleanup_module()
{
remove_proc_entry(procfs_name, &proc_root);
printk(KERN_INFO "/proc/%s removed\n", procfs_name);
}
```

- tests

```
# insmod procfs1.ko
# cat /proc/helloworld
HelloWorld!
# rmmod procfs1
```

3.2 read and write a */proc* file

- procfs2.c

```
/**
 *  procfs2.c -  read and write a /proc file
 *
 */

#include <linux/module.h>/* Specifically, a module */
#include <linux/kernel.h>/* We're doing kernel work */
#include <linux/proc_fs.h>/* Necessary because we use the proc fs */
#include <asm/uaccess.h>/* for copy_from_user */

#define PROCFS_MAX_SIZE 1024
#define PROCFS_NAME  "buffer1k"

/**
 * This structure hold information about the /proc file
 *
 */
static struct proc_dir_entry *Our_Proc_File;

/**
 * The buffer used to store character for this module
 *
 */
static char procfs_buffer[PROCFS_MAX_SIZE];

/**
 * The size of the buffer
```

```
 *
 */
static unsigned long procfs_buffer_size = 0;

/**
 * This function is called then the /proc file is read
 *
 */
int
procfile_read(char *buffer,
      char **buffer_location,
      off_t offset, int buffer_length, int *eof, void *data)
{
int ret;

printk(KERN_INFO "procfile_read (/proc/%s) called\n", PROCFS_NAME);

if (offset > 0) {
/* we have finished to read, return 0 */
ret  = 0;
} else {
/* fill the buffer, return the buffer size */
memcpy(buffer, procfs_buffer, procfs_buffer_size);
ret = procfs_buffer_size;
}

return ret;
}

/**
 * This function is called with the /proc file is written
 *
 */
int procfile_write(struct file *file, const char *buffer, unsigned long count,
   void *data)
{
/* get buffer size */
procfs_buffer_size = count;
if (procfs_buffer_size > PROCFS_MAX_SIZE ) {
procfs_buffer_size = PROCFS_MAX_SIZE;
}

/* write data to the buffer */
if ( copy_from_user(procfs_buffer, buffer, procfs_buffer_size) ) {
return -EFAULT;
}

return procfs_buffer_size;
}

/**
 *This function is called when the module is loaded
 *
 */
```

```
int init_module()
{
/* create the /proc file */
Our_Proc_File = create_proc_entry(PROCFS_NAME, 0644, NULL);

if (Our_Proc_File == NULL) {
printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
PROCFS_NAME);
return -ENOMEM;
}

Our_Proc_File->read_proc  = procfile_read;
Our_Proc_File->write_proc = procfile_write;
Our_Proc_File->owner    = THIS_MODULE;
Our_Proc_File->mode    = S_IFREG | S_IRUGO;
Our_Proc_File->uid    = 0;
Our_Proc_File->gid    = 0;
Our_Proc_File->size    = 37;

printk(KERN_INFO "/proc/%s created\n", PROCFS_NAME);
return 0; /* everything is ok */
}

/**
 *This function is called when the module is unloaded
 *
 */
void cleanup_module()
{
remove_proc_entry(PROCFS_NAME, Our_Proc_File->parent);
printk(KERN_INFO "/proc/%s removed\n", PROCFS_NAME);
}
```

- test

```
# insmod procfs2.ko
# cat > /proc/buffer1k
plouif plouf toto
en slip :)
# cat /proc/buffer1k
en slip :)
# rmmod procfs2
```

3.3 manage /proc file with standard filesystem

Introduction des droits d'accèes.

3.4 manage /proc file with seq_file

Les séquences permettent de décomposer l'action sour forme de boucle for.

- start
- next...
- stop

4 Talking to Device Files (writes and IOCTLs)

- chardev.c

```
/*
 *  chardev.c - Create an input/output character device
 */

#include <linux/kernel.h>/* We're doing kernel work */
#include <linux/module.h>/* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h>/* for get_user and put_user */

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

/*
 * Is the device open right now? Used to prevent
 * concurent access into the same device
 */
static int Device_Open = 0;

/*
 * The message the device will give when asked
 */
static char Message[BUF_LEN];

/*
 * How far did the process reading the message get?
 * Useful if the message is larger than the size of the
 * buffer we get to fill in device_read.
 */
static char *Message_Ptr;

/*
 * This is called whenever a process attempts to open the device file
 */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
printk(KERN_INFO "device_open(%p)\n", file);
#endif

/*
 * We don't want to talk to two processes at the same time
 */
if (Device_Open)
return -EBUSY;

Device_Open++;
/*
 * Initialize the message
```

```
 */
Message_Ptr = Message;
try_module_get(THIS_MODULE);
return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
printk(KERN_INFO "device_release(%p,%p)\n", inode, file);
#endif

/*
 * We're now ready for our next caller
 */
Device_Open--;

module_put(THIS_MODULE);
return SUCCESS;
}

/*
 * This function is called whenever a process which has already opened the
 * device file attempts to read from it.
 */
static ssize_t device_read(struct file *file,/* see include/linux/fs.h   */
   char __user * buffer,/* buffer to be
 * filled with data */
   size_t length,/* length of the buffer     */
   loff_t * offset)
{
/*
 * Number of bytes actually written to the buffer
 */
int bytes_read = 0;

#ifdef DEBUG
printk(KERN_INFO "device_read(%p,%p,%d)\n", file, buffer, length);
#endif

/*
 * If we're at the end of the message, return 0
 * (which signifies end of file)
 */
if (*Message_Ptr == 0)
return 0;

/*
 * Actually put the data into the buffer
 */
while (length && *Message_Ptr) {

/*
 * Because the buffer is in the user data segment,
```

```
 * not the kernel data segment, assignment wouldn't
 * work. Instead, we have to use put_user which
 * copies data from the kernel data segment to the
 * user data segment.
 */
put_user(*(Message_Ptr++), buffer++);
length--;
bytes_read++;
}

#ifdef DEBUG
printk(KERN_INFO "Read %d bytes, %d left\n", bytes_read, length);
#endif

/*
 * Read functions are supposed to return the number
 * of bytes actually inserted into the buffer
 */
return bytes_read;
}

/*
 * This function is called when somebody tries to
 * write into our device file.
 */
static ssize_t
device_write(struct file *file,
     const char __user * buffer, size_t length, loff_t * offset)
{
int i;

#ifdef DEBUG
printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, length);
#endif

for (i = 0; i < length && i < BUF_LEN; i++)
get_user(Message[i], buffer + i);

Message_Ptr = Message;

/*
 * Again, return the number of input characters used
 */
return i;
}

/*
 * This function is called whenever a process tries to do an ioctl on our
 * device file. We get two extra parameters (additional to the inode and file
 * structures, which all device functions get): the number of the ioctl called
 * and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to the
 * calling process), the ioctl call returns the output of this function.
```

```
 *
 */
int device_ioctl(struct inode *inode,/* see include/linux/fs.h */
 struct file *file,/* ditto */
 unsigned int ioctl_num,/* number and param for ioctl */
 unsigned long ioctl_param)
{
int i;
char *temp;
char ch;

/*
 * Switch according to the ioctl called
 */
switch (ioctl_num) {
case IOCTL_SET_MSG:
/*
 * Receive a pointer to a message (in user space) and set that
 * to be the device's message.  Get the parameter given to
 * ioctl by the process.
 */
temp = (char *)ioctl_param;

/*
 * Find the length of the message
 */
get_user(ch, temp);
for (i = 0; ch && i < BUF_LEN; i++, temp++)
get_user(ch, temp);

device_write(file, (char *)ioctl_param, i, 0);
break;

case IOCTL_GET_MSG:
/*
 * Give the current message to the calling process -
 * the parameter we got is a pointer, fill it.
 */
i = device_read(file, (char *)ioctl_param, 99, 0);

/*
 * Put a zero at the end of the buffer, so it will be
 * properly terminated
 */
put_user('\0', (char *)ioctl_param + i);
break;

case IOCTL_GET_NTH_BYTE:
/*
 * This ioctl is both input (ioctl_param) and
 * output (the return value of this function)
 */
return Message[ioctl_param];
break;
```

```
}

return SUCCESS;
}

/* Module Declarations */

/*
 * This structure will hold the functions to be called
 * when a process does something to the device we
 * created. Since a pointer to this structure is kept in
 * the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions.
 */
struct file_operations Fops = {
.read = device_read,
.write = device_write,
.ioctl = device_ioctl,
.open = device_open,
.release = device_release,/* a.k.a. close */
};

/*
 * Initialize the module - Register the character device
 */
int init_module()
{
int ret_val;
/*
 * Register the character device (atleast try)
 */
ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

/*
 * Negative values signify an error
 */
if (ret_val < 0) {
printk(KERN_ALERT "%s failed with %d\n",
       "Sorry, registering the character device ", ret_val);
return ret_val;
}

printk(KERN_INFO "%s The major device number is %d.\n",
       "Registeration is a success", MAJOR_NUM);
printk(KERN_INFO "If you want to talk to the device driver,\n");
printk(KERN_INFO "you'll have to create a device file. \n");
printk(KERN_INFO "We suggest you use:\n");
printk(KERN_INFO "mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
printk(KERN_INFO "The device file name is important, because\n");
printk(KERN_INFO "the ioctl program assumes that's the\n");
printk(KERN_INFO "file you'll use.\n");

return 0;
}
```

```
/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
int ret;

/*
 * Unregister the device
 */
ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

/*
 * If there's an error, report it
 */
if (ret < 0)
printk(KERN_ALERT "Error: unregister_chrdev: %d\n", ret);
}
```

- chardev.h

```
/*
 *  chardev.h - the header file with the ioctl definitions.
 *
 *  The declarations here have to be in a header file, because
 *  they need to be known both to the kernel module
 *  (in chardev.c) and the process calling ioctl (ioctl.c)
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
 * The major device number. We can't rely on dynamic
 * registration any more, because ioctls need to know
 * it.
 */
#define MAJOR_NUM 100

/*
 * Set the message of the device driver
 */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/*
 * _IOR means that we're creating an ioctl command
 * number for passing information from a user process
 * to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device
 * number we're using.
 *
```

```
 * The second argument is the number of the command
 * (there could be several with different meanings).
 *
 * The third argument is the type we want to get from
 * the process to the kernel.
 */

/*
 * Get the message of the device driver
 */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
 * This IOCTL is used for output, to get the message
 * of the device driver. However, we still need the
 * buffer to place the message in to be input,
 * as it is allocated by the process.
 */

/*
 * Get the n'th byte of the message
 */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
 * The IOCTL is used for both input and output. It
 * receives from the user a number, n, and returns
 * Message[n].
 */

/*
 * The name of the device file
 */
#define DEVICE_FILE_NAME "char_dev"

#endif
```

- ioctl.c

```
/*
 *  ioctl.c - the process to use ioctl's to control the kernel module
 *
 *  Until now we could have used cat for input and output.  But now
 *  we need to do ioctl's, which require writing our own process.
 */

/*
 * device specifics, such as ioctl numbers and the
 * major device file.
 */
#include "chardev.h"

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>/* open */
#include <unistd.h>/* exit */
```

```
#include <sys/ioctl.h>/* ioctl */

/*
 * Functions for the ioctl calls
 */

ioctl_set_msg(int file_desc, char *message)
{
int ret_val;

ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

if (ret_val < 0) {
printf("ioctl_set_msg failed:%d\n", ret_val);
exit(-1);
}
}

ioctl_get_msg(int file_desc)
{
int ret_val;
char message[100];

/*
 * Warning - this is dangerous because we don't tell
 * the kernel how far it's allowed to write, so it
 * might overflow the buffer. In a real production
 * program, we would have used two ioctls - one to tell
 * the kernel the buffer length and another to give
 * it the buffer to fill
 */
ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

if (ret_val < 0) {
printf("ioctl_get_msg failed:%d\n", ret_val);
exit(-1);
}

printf("get_msg message:%s\n", message);
}

ioctl_get_nth_byte(int file_desc)
{
int i;
char c;

printf("get_nth_byte message:");

i = 0;
do {
c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

if (c < 0) {
printf
```

```
    ("ioctl_get_nth_byte failed at the %d'th byte:\n",
     i);
exit(-1);
}

putchar(c);
} while (c != 0);
putchar('\n');
}

/*
 * Main - Call the ioctl functions
 */
main()
{
int file_desc, ret_val;
char *msg = "Message passed by ioctl\n";

file_desc = open(DEVICE_FILE_NAME, 0);
if (file_desc < 0) {
printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
exit(-1);
}

ioctl_get_nth_byte(file_desc);
ioctl_get_msg(file_desc);
ioctl_set_msg(file_desc, msg);

close(file_desc);
}
```

- test

```
# insmod chardev.ko
# mknod char_dev c 100 0

$ gcc ioctl.c
$ a.out

# rmmod chardev.ko
```

5 Replacing Printks

5.1 Replacing Printks

- print_string.c

```
/*
 *  print_string.c - Send output to the tty we're running on, regardless if it's
 *  through X11, telnet, etc.  We do this by printing the string to the tty
 *  associated with the current task.
 */
#include <linux/kernel.h>
#include <linux/module.h>
```

```
#include <linux/init.h>
#include <linux/sched.h>/* For current */
#include <linux/tty.h>/* For the tty declarations */
#include <linux/version.h>/* For LINUX_VERSION_CODE */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static void print_string(char *str)
{
struct tty_struct *my_tty;

/*
 * tty struct went into signal struct in 2.6.6
 */
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,5) )
/*
 * The tty for the current task
 */
my_tty = current->tty;
#else
/*
 * The tty for the current task, for 2.6.6+ kernels
 */
my_tty = current->signal->tty;
#endif

/*
 * If my_tty is NULL, the current task has no tty you can print to
 * (ie, if it's a daemon).  If so, there's nothing we can do.
 */
if (my_tty != NULL) {

/*
 * my_tty->driver is a struct which holds the tty's functions,
 * one of which (write) is used to write strings to the tty.
 * It can be used to take a string either from the user's or
 * kernel's memory segment.
 *
 * The function's 1st parameter is the tty to write to,
 * because the same function would normally be used for all
 * tty's of a certain type.  The 2nd parameter controls
 * whether the function receives a string from kernel
 * memory (false, 0) or from user memory (true, non zero).
 * BTW: this param has been removed in Kernels > 2.6.9
 * The (2nd) 3rd parameter is a pointer to a string.
 * The (3rd) 4th parameter is the length of the string.
 *
 * As you will see below, sometimes it's necessary to use
 * preprocessor stuff to create code that works for different
 * kernel versions. The (naive) approach we've taken here
 * does not scale well. The right way to deal with this
 * is described in section 2 of
 * linux/Documentation/SubmittingPatches
```

```
 */
  (((my_tty->driver)->ops)->write) (my_tty,/* The tty itself */
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,9) )
   0,/* Don't take the string
   from user space        */
#endif
   str,/* String                 */
   strlen(str)); /* Length */

/*
 * ttys were originally hardware devices, which (usually)
 * strictly followed the ASCII standard.  In ASCII, to move to
 * a new line you need two characters, a carriage return and a
 * line feed.  On Unix, the ASCII line feed is used for both
 * purposes - so we can't just use \n, because it wouldn't have
 * a carriage return and the next line will start at the
 * column right after the line feed.
 *
 * This is why text files are different between Unix and
 * MS Windows.  In CP/M and derivatives, like MS-DOS and
 * MS Windows, the ASCII standard was strictly adhered to,
 * and therefore a newline requirs both a LF and a CR.
 */

#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,9) )
((my_tty->driver)->write) (my_tty, 0, "\015\012", 2);
#else
(((my_tty->driver)->ops)->write) (my_tty, "\015\012", 2);
#endif
}
}

static int __init print_string_init(void)
{
print_string("The module has been inserted.  Hello world!");
return 0;
}

static void __exit print_string_exit(void)
{
print_string("The module has been removed.  Farewell world!");
}

module_init(print_string_init);
module_exit(print_string_exit);
```

5.2 Flashing keyboard LEDs

- kbleds.c

```
/*
 *  kbleds.c - Blink keyboard leds until the module is unloaded.
 */
```

```
#include <linux/module.h>
//#include <linux/config.h>
#include <linux/init.h>
#include <linux/vt_kern.h>/* For fg_console */
/* redondant */
//#include <linux/tty.h>
//#include <linux/kd.h>/* For KDSETLED */
//#include <linux/vt.h>         /* For MAX_NR_CONSOLES */
//#include <linux/console_struct.h>/* For vc_cons */

MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");
MODULE_AUTHOR("Daniele Paolo Scarpazza");
MODULE_LICENSE("GPL");

/* bibalse a lula */
#define ALL_LEDS_OFF 0x00
#define LED_ONE_ON   0x02
#define LED_TWO_ON   0x04
#define LED_THREE_ON 0x01
#define RETURNING    0x08
#define OFF_DELAY    HZ*2

struct timer_list my_timer;
struct tty_driver *my_driver;
char kbledstatus = ALL_LEDS_OFF;

#define BLINK_DELAY   HZ/5
#define ALL_LEDS_ON   0x07
#define RESTORE_LEDS  0xFF

/*
 * Function my_timer_func blinks the keyboard LEDs periodically by invoking
 * command KDSETLED of ioctl() on the keyboard driver. To learn more on virtual
 * terminal ioctl operations, please see file:
 *     /usr/src/linux/drivers/char/vt_ioctl.c, function vt_ioctl().
 *
 * The argument to KDSETLED is alternatively set to 7 (thus causing the led
 * mode to be set to LED_SHOW_IOCTL, and all the leds are lit) and to 0xFF
 * (any value above 7 switches back the led mode to LED_SHOW_FLAGS, thus
 * the LEDs reflect the actual keyboard status).  To learn more on this,
 * please see file:
 *     /usr/src/linux/drivers/char/keyboard.c, function setledstate().
 *
 */

static void my_timer_func(unsigned long ptr)
{
int *pstatus = (int *)ptr;

/* if (*pstatus == ALL_LEDS_ON)
*pstatus = RESTORE_LEDS;
else
*pstatus = ALL_LEDS_ON;
*/
```

```
/* hacked by David Hasselhoff (sorry about that) */
if (*pstatus == ALL_LEDS_OFF)
  *pstatus = LED_ONE_ON;

else if (*pstatus == LED_ONE_ON)
  *pstatus = LED_TWO_ON;

else if (*pstatus == LED_TWO_ON)
  *pstatus = LED_THREE_ON;

else if (*pstatus == LED_THREE_ON)
  *pstatus = LED_TWO_ON | RETURNING;

else if (*pstatus == (LED_TWO_ON | RETURNING))
  *pstatus = LED_ONE_ON | RETURNING;

else /* (*pstatus == LED_ONE_ON | RETURNING) */
  *pstatus = ALL_LEDS_OFF;

((my_driver->ops)->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
   (*pstatus & (~RETURNING)));

if (*pstatus == ALL_LEDS_OFF)
  my_timer.expires = jiffies + OFF_DELAY;
else
  my_timer.expires = jiffies + BLINK_DELAY;
add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
int i;

printk(KERN_INFO "kbleds: loading\n");
printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
for (i = 0; i < MAX_NR_CONSOLES; i++) {
if (!vc_cons[i].d)
break;
printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
       MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
       (unsigned long)vc_cons[i].d->vc_tty);
}
printk(KERN_INFO "kbleds: finished scanning consoles\n");

my_driver = vc_cons[fg_console].d->vc_tty->driver;
printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);

/*
 * Set up the LED blink timer the first time
 */
init_timer(&my_timer);
my_timer.function = my_timer_func;
my_timer.data = (unsigned long)&kbledstatus;
```

```
my_timer.expires = jiffies + BLINK_DELAY;
add_timer(&my_timer);

return 0;
}

static void __exit kbleds_cleanup(void)
{
printk(KERN_INFO "kbleds: unloading...\n");
del_timer(&my_timer);
((my_driver->ops)->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
    RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);
```