Linux Kernel Module Programming Guide

1999 Ori Pomerantz

Version 1.1.0, 26 April 1999.

This book is about writing Linux Kernel Modules. It is, hopefully, useful for programmers who know C and want to learn how to write kernel modules. It is written as an ‘How-To’ instruction manual, with examples of all of the important techniques.

Although this book touches on many points of kernel design, it is not supposed to fulfill that need — there are other books on this subject, both in print and in the Linux documentation project.

You may freely copy and redistribute this book under certain conditions. Please see the copyright and distribution statement.

Names of all products herein are used for identification purposes only and are trademarks and/or registered trademarks of their respective owners. I make no claim of ownership or corporate association with the products or companies that own them.

Copyright © 1999 Ori Pomerantz

Ori Pomerantz
Apt. #1032
2355 N Hwy 360
Grand Prairie
TX 75050
USA
E-mail: mpg@simple-tech.com

The *Linux Kernel Module Programming Guide* is a free book; you may reproduce and/or modify it under the terms of version 2 (or, at your option, any later version) of the GNU General Public License as published by the Free Software Foundation. Version 2 is enclosed with this document at Appendix E.

This book is distributed in the hope it will be useful, but **without any warranty**; without even the implied warranty of merchantability or fitness for a particular purpose.

The author encourages wide distribution of this book for personal or commercial use, provided the above copyright notice remains intact and the method adheres to the provisions of the GNU General Public License (see Appendix E). In summary, you may copy and distribute this book free of charge or for a profit. No explicit permission is required from the author for reproduction of this book in any medium, physical or electronic.

Note, derivative works and translations of this document *must* be placed under the GNU General Public License, and the original copyright notice must remain intact. If you have contributed new material to this book, you must make the source code (e.g., LaTeX source) available for your revisions. Please make revisions and updates available directly to the document maintainer, Ori Pomerantz. This will allow for the merging of updates and provide consistent revisions to the Linux community.

If you plan to publish and distribute this book commercially, donations, royalties, and/or printed copies are greatly appreciated by the author and the Linux Documentation Project. Contributing in this way shows your support for free software and the Linux Documentation Project. If you have questions or comments, please contact the address above.

Contents

Chapter 0

Introduction

So, you want to write a kernel module. You know C, you've written a number of normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.

Well, welcome to the club. I once had a wild pointer wipe an important directory under DOS (thankfully, now it stands for the **D**ead **O**perating **S**ystem), and I don't see why living under Linux should be any safer.

Warning: I wrote this and checked the program under versions 2.0.35 and 2.2.3 of the kernel running on a Pentium. For the most part, it should work on other CPUs and on other versions of the kernel, as long as they are 2.0.x or 2.2.x, but I can't promise anything. One exception is chapter 11, which should not work on any architecture except for x86.

0.1 Who Should Read This

This document is for people who want to write kernel modules. Although I will touch on how things are done in the kernel in several places, that is not my purpose. There are enough good sources which do a better job than I could have done.

This document is also for people who know how to write kernel modules, but have not yet adapted to version 2.2 of the kernel. If you are such a person, I suggest you look at appendix A to see all the differences I encountered while updating the examples. The list is nowhere near comprehensive, but I think it covers most of the basic functionality and will be enough to get you started.

The kernel is a great piece of programming, and I believe that programmers should read

at least some kernel source files and understand them. Having said that, I also believe in the value of playing with the system first and asking questions later. When I learn a new programming language, I don't start with reading the library code, but by writing a small 'hello, world' program. I don't see why playing with the kernel should be any different.

0.2 Note on the Style

I like to put as many jokes as possible into my documentation. I'm writing this because I enjoy it, and I assume most of you are reading this for the same reason. If you just want to get to the point, ignore all the normal text and read the source code. I promise to put all the important details in remarks.

0.3 Changes

0.3.1 New in version 1.0.1

1. **Changes section**, 0.3.
2. **How to find the minor device number**, 2.
3. **Fixed the explanation of the difference between character and device files**, 2
4. **Makefiles for Kernel Modules**, 1.1.
5. **Symmetrical Multiprocessing**, 12.
6. **A 'Bad Ideas' Chapter**, 13.

0.3.2 New in version 1.1.0

1. **Support for version 2.2 of the kernel**, all over the place.
2. **Multi kernel version source files**, 2.1.
3. **Changes between 2.0 and 2.2**, A.
4. **Kernel Modules in Multiple Source Files**, 1.2.
5. **Suggestion not to let modules which mess with system calls be rmmod'ed**, 7.

0.4 Acknowledgements

I'd like to thank Yoav Weiss for many helpful ideas and discussions, as well as for finding mistakes within this document before its publication. Of course, any remaining mistakes are purely my fault.

The TEX skeleton for this book was shamelessly stolen from the 'Linux Installation and Getting Started' guide, where the TEX work was done by Matt Welsh.

My gratitude to Linus Torvalds, Richard Stallman and all the other people who made it possible for me to run a high quality operating system on my computer and get the source code goes without saying (yeah, right — then why did I say it?).

0.4.1 For version 1.0.1

I couldn't list everybody who e-mailed me here, and if I've left you out I apologize in advance. The following people were specially helpful:

- **Frodo Looijaard from the Netherlands** For a host of useful suggestions, and information about the 2.1.x kernels.
- **Stephen Judd from New Zealand** Spelling corrections.
- **Magnus Ahltorp from Sweden** Correcting a mistake of mine about the difference between character and block devices.

0.4.2 For version 1.1.0

- **Emmanuel Papirakis from Quebec, Canada** For porting all of the examples to version 2.2 of the kernel.
- **Frodo Looijaard from the Netherlands** For telling me how to create a multiple file kernel module (1.2).

Of course, any remaining mistakes are my own, and if you think they make the book unusable you're welcome to apply for a full refund of the money you paid me for it.

Chapter 1

Hello, world

When the first caveman programmer chiseled the first program on the walls of the first cave computer, it was a program to paint the string ‘Hello, world’ in Antelope pictures. Roman programming textbooks began with the ‘Salut, Mundi’ program. I don’t know what happens to people who break with this tradition, and I think it’s safer not to find out.

A kernel module has to have at least two functions: `init_module` which is called when the module is inserted into the kernel, and `cleanup_module` which is called just before it is removed. Typically, `init_module` either registers a handler for something with the kernel, or it replaces one of the kernel function with its own code (usually code to do something and then call the original function). The `cleanup_module` function is supposed to undo whatever `init_module` did, so the module can be unloaded safely.

hello.c

```
/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We’re doing kernel work */
```

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


/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}


/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
```

1.1 Makefiles for Kernel Modules

A kernel module is not an independant executable, but an object file which will be linked into the kernel in runtime. As a result, they should be compiled with the `-c` flag. Also, all kernel modules have to be compiled with certain symbols defined.

- `__KERNEL__` — This tells the header files that this code will be run in kernel mode, not as part of a user process.
- `MODULE` — This tells the header files to give the appropriate definitions for a kernel module.
- `LINUX` — Technically speaking, this is not necessary. However, if you ever want to write a serious kernel module which will compile on more than one operating system, you'll be happy you did. This will allow you to do conditional compilation on the parts which are OS dependant.

There are other symbols which have to be included, or not, depending on the flags the kernel was compiled with. If you're not sure how the kernel was compiled, look it up in `/usr/include/linux/config.h`

- `__SMP__` — Symmetrical MultiProcessing. This has to be defined if the kernel was compiled to support symmetrical multiprocessing (even if it's running just on one CPU). If you use Symmetrical MultiProcessing, there are other things you need to do (see chapter 12).
- `CONFIG_MODVERSIONS` — If CONFIG_MODVERSIONS was enabled, you need to have it defined when compiling the kernel module and and to include `/usr/include/linux/modversions.h`. This can also be done by the code itself.

Makefile

```
# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c hello.c
echo insmod hello.o to turn it on
echo rmmod hello to turn if off
echo
echo X and kernel programming do not mix.
```

```
echo Do the insmod and rmmod from outside X.
```

So, now the only thing left is to `su` to root (you didn't compile this as root, did you? Living on the edge[1]...), and then `insmod hello` and `rmmod hello` to your heart's content. While you do it, notice your new kernel module in `/proc/modules`.

By the way, the reason why the Makefile recommends against doing `insmod` from X is because when the kernel has a message to print with `printk`, it sends it to the console. When you don't use X, it just goes to the virtual terminal you're using (the one you chose with Alt-F<n>) and you see it. When you do use X, on the other hand, there are two possibilities. Either you have a console open with `xterm -C`, in which case the output will be sent there, or you don't, in which case the output will go to virtual terminal 7 — the one 'covered' by X.

If your kernel becomes unstable you're likelier to get the debug messages without X. Outside of X, `printk` goes directly from the kernel to the console. In X, on the other hand, `printk`'s go to a user mode process (`xterm -C`). When that process receives CPU time, it is supposed to send it to the X server process. Then, when the X server receives the CPU, it is supposed to display it — but an unstable kernel usually means that the system is about to crash or reboot, so you don't want to delay the error messages, which might explain to you what went wrong, for longer than you have to.

1.2 Multiple File Kernel Modules

Sometimes it makes sense to divide a kernel module between several source files. In this case, you need to do the following:

1. In all the source files but one, add the line `#define __NO_VERSION__`. This is important because `module.h` normally includes the definition of `kernel_version`, a global variable with the kernel version the module is compiled for. If you need `version.h`, you need to include it yourself, because `module.h` won't do it for you with `__NO_VERSION__`.

2. Compile all the source files as usual.

3. Combine all the object files into a single one. Under x86, do it with `ld -m elf_i386 -r -o <name`

[1] The reason I prefer not to compile as root is that the least done as root the safer the box is. I work in computer security, so I'm paranoid

of module>.o <1st source file>.o <2nd source file>.o.

Here's an example of such a kernel module.

start.c

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

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



/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
```

```
}
```

stop.c

```
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version. This
 * file includes just the stop routine.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */

#define __NO_VERSION__      /* This isn't "the" file
                             * of the kernel module */
#include <linux/module.h>   /* Specifically, a module */

#include <linux/version.h>   /* Not included by
                              * module.h because
                              * of the __NO_VERSION__ */



/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

```
/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
```

Makefile

```
# Makefile for a multifile kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: start.o stop.o
ld -m elf_i386 -r -o hello.o start.o stop.o

start.o: start.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c start.c

stop.o: stop.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c stop.c
```

Chapter 2

Character Device Files

So, now we're bold kernel programmers and we know how to write kernel modules to do nothing. We feel proud of ourselves and we hold our heads up high. But somehow we get the feeling that something is missing. Catatonic modules are not much fun.

There are two major ways for a kernel module to talk to processes. One is through device files (like the files in the `/dev` directory), the other is to use the proc file system. Since one of the major reasons to write something in the kernel is to support some kind of hardware device, we'll begin with device files.

The original purpose of device files is to allow processes to communicate with device drivers in the kernel, and through them with physical devices (modems, terminals, etc.). The way this is implemented is the following.

Each device driver, which is responsible for some type of hardware, is assigned its own major number. The list of drivers and their major numbers is available in `/proc/devices`. Each physical device managed by a device driver is assigned a minor number. The `/dev` directory is supposed to include a special file, called a device file, for each of those devices, whether or not it's really installed on the system.

For example, if you do `ls -l /dev/hd[ab]*`, you'll see all of the IDE hard disk partitions which might be connected to a machine. Notice that all of them use the same major number, 3, but the minor number changes from one to the other *Disclaimer: This assumes you're using a PC architecture. I don't know about devices on Linux running on other architectures*.

When the system was installed, all of those device files were created by the `mknod` command. There's no technical reason why they have to be in the `/dev` directory, it's just

a useful convention. When creating a device file for testing purposes, as with the exercise here, it would probably make more sense to place it in the directory where you compile the kernel module.

Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose by which order to respond to them. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of `ls -l`. If it's 'b' then it's a block device, and if it's 'c' then it's a character device.

This module is divided into two separate parts: The module part which registers the device and the device driver part. The `init_module` function calls `module_register_chrdev` to add the device driver to the kernel's character device driver table. It also returns the major number to be used for the driver. The `cleanup_module` function deregisters the device.

This (registering something and unregistering it) is the general functionality of those two functions. Things in the kernel don't run on their own initiative, like processes, but are called, by processes via system calls, or by hardware devices via interrupts, or by other parts of the kernel (simply by calling specific functions). As a result, when you add code to the kernel, you're supposed to register it as the handler for a certain type of event and when you remove it, you're supposed to unregister it.

The device driver proper is composed of the four device_<action> functions, which are called when somebody tries to do something with a device file which has our major number. The way the kernel knows to call them is via the `file_operations` structure, `Fops`, which was given when the device was registered, which includes pointers to those four functions.

Another point we need to remember here is that we can't allow the kernel module to be `rmmod`ed whenever root feels like it. The reason is that if the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of

another function within the kernel. The results of this would be impossible to predict, but they can't be positive.

Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With `cleanup_module` that is impossible because it's a void function. Once `cleanup_module` is called, the module is dead. However, there is a use counter which counts how many other kernel modules are using this kernel module, called the reference count (that's the last number of the line in `/proc/modules`). If this number isn't zero, `rmmod` will fail. The module's reference count is available in the variable `mod_use_count_`. Since there are macros defined for handling this variable (`MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT`), we prefer to use them, rather than `mod_use_count_` directly, so we'll be safe if the implementation changes in the future.

chardev.c

```
/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * Create a character device (read only)
 */

/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */
#include <linux/fs.h>       /* The character device
                             * definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does
```

```
                                * next to nothing at
                                * at present, but may
                                * help for compatibility
                                * with future versions
                                * of Linux */


/* In 2.2.3 /usr/include/linux/version.h includes
 * a macro for this, but 2.0.35 doesn't - so I add
 * it here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


/* Conditional compilation. LINUX_VERSION_CODE is
 * the code (as per KERNEL_VERSION) of this version. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for put_user */
#endif



#define SUCCESS 0


/* Device Declarations ************************** */

/* The name for our device, as it will appear
 * in /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message from the device */
#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 function is called whenever a process
 * attempts to open the device file */
static int device_open(struct inode *inode,
       struct file *file)
{
  static int counter = 0;

#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif

  /* This is how you get the minor device number in
   * case you have more than one physical device using
   * the driver. */
  printk("Device: %d.%d\n",
 inode->i_rdev >> 8, inode->i_rdev & 0xFF);

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

  /* If this was a process, we would have had to be
```

```
 * more careful here.
 *
 * In the case of processes, the danger would be
 * that one process might have check Device_Open
 * and then be replaced by the schedualer by another
 * process which runs this function. Then, when the
 * first process was back on the CPU, it would assume
 * the device is still not open.
 *
 * However, Linux guarantees that a process won't be
 * replaced while it is running in kernel context.
 *
 * In the case of SMP, one CPU might increment
 * Device_Open while another CPU is here, right after
 * the check. However, in version 2.0 of the
 * kernel this is not a problem because there's a lock
 * to guarantee only one CPU will be kernel module at
 * the same time. This is bad in  terms of
 * performance, so version 2.2 changed it.
 * Unfortunately, I don't have access to an SMP box
 * to check how it works with SMP.
 */

Device_Open++;

/* Initialize the message. */
sprintf(Message,
  "If I told you once, I told you %d times - %s",
  counter++,
  "Hello, world\n");
/* The only reason we're allowed to do this sprintf
 * is because the maximum length of the message
 * (assuming 32 bit integers - up to 10 digits
 * with the minus sign) is less than BUF_LEN, which
 * is 80. BE CAREFUL NOT TO OVERFLOW BUFFERS,
 * ESPECIALLY IN THE KERNEL!!!
```

```
   */

  Message_Ptr = Message;

  /* Make sure that the module isn't removed while
   * the file is open by incrementing the usage count
   * (the number of opened references to the module, if
   * it's not zero rmmod will fail)
   */
  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* This function is called when a process closes the
 * device file. It doesn't have a return value in
 * version 2.0.x because it can't fail (you must ALWAYS
 * be able to close a device). In version 2.2.x it is
 * allowed to fail - but we won't let it.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
  struct file *file)
#else
static void device_release(struct inode *inode,
   struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

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

```
  /* Decrement the usage count, otherwise once you
   * opened the file you'll never get rid of the module.
   */
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}


/* This function is called whenever a process which
 * have already opened the device file attempts to
 * read from it. */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* The buffer to fill with data */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,   /* The buffer to fill with
     * the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

  /* 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 ("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 - unsupported in this example. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
    const char *buffer,    /* The buffer */
    size_t length,   /* The length of the buffer */
```

```
    loff_t *offset)  /* Our offset in the file */
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  return -EINVAL;
}



/* Module Declarations **************************** */

/* The major device number for the device. This is
 * global (well, static, which in this context is global
 * within this file) because it has to be accessible
 * both for registration and for release. */
static int Major;

/* 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 = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  NULL,   /* ioctl */
```

```
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,   /* flush */
#endif
  device_release  /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
  /* Register the character device (atleast try) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);

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

  printk ("%s The major device number is %d.\n",
          "Registeration is a success.",
          Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
          "and see what happens.\n");

  return 0;
```

```
}


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

  /* Unregister the device */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

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

2.1 Multiple Kernel Versions Source Files

The system calls, which are the major interface the kernel shows to the processes, generally stay the same across versions. A new system call may be added, but usually the old ones will behave exactly like they used to. This is necessary for backward compatibility — a new kernel version is **not** supposed to break regular processes. In most cases, the device files will also remain the same. On the other hand, the internal interfaces within the kernel can and do change between versions.

The Linux kernel versions are divided between the stable versions (n.<even number>.m) and the development versions (n.<odd number>.m). The development versions include all the cool new ideas, including those which will be considered a mistake, or reimplemented, in the next version. As a result, you can't trust the interface to remain the same in those versions (which is why I don't bother to support them in this book, it's too much work and it would become dated too quickly). In the stable versions, on the other hand, we can expect the interface to remain the same regardless of the bug fix version (the m number).

This version of the MPG includes support for both version 2.0.x and version 2.2.x of the Linux kernel. Since there are differences between the two, this requires condi-

tional compilation depending on the kernel version. The way to do this to use the macro `LINUX_VERSION_CODE`. In version a.b.c of the kernel, the value of this macro would be $2^{16}a + 2^8b + c$. To get the value for a specific kernel version, we can use the `KERNEL_VERSION` macro. Since it's not defined in 2.0.35, we define it ourselves if necessary.

Chapter 3

The /proc File System

In Linux there is an additional mechanism for the kernel and kernel modules to send information to processes — the `/proc` file system. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as `/proc/modules` which has the list of modules and `/proc/meminfo` which has memory usage statistics.

The method to use the proc file system is very similar to the one used with device drivers — you create a structure with all the information needed for the `/proc` file, including pointers to any handler functions (in our case there is only one, the one called when somebody attempts to read from the `/proc` file). Then, `init_module` registers the structure with the kernel and `cleanup_module` unregisters it.

The reason we use `proc_register_dynamic`[1] is because we don't want to determine the inode number used for our file in advance, but to allow the kernel to determine it to prevent clashes. Normal file systems are located on a disk, rather than just in memory (which is where `/proc` is), and in that case the inode number is a pointer to a disk location where the file's index-node (inode for short) is located. The inode contains information about the file, for example the file's permissions, together with a pointer to the disk location or locations where the file's data can be found.

Because we don't get called when the file is opened or closed, there's no where for us to put `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` in this module, and if the file is opened and then the module is removed, there's no way to avoid the consequences. In the next chapter we'll see a harder to implement, but more flexible, way of dealing with

[1] In version 2.0, in version 2.2 this is done for us automatically if we set the inode to zero.

/proc files which will allow us to protect against this problem as well.

procfs.c

```
/* procfs.c -  create a "file" in /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* 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. Zero (for future use?).

Usage and Return Value

If you use your own buffer, like I do, put its location in the second argument and return the number of bytes used in the buffer.

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 zero)
{
  int len;  /* The number of bytes actually used */

  /* This is static so it will still be in memory
   * when we leave this function */
  static char my_buffer[80];

  static int count = 1;

  /* 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)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer,
    "For the %d%s time, go away!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
```

```
          (count % 10 == 3) ? "rd" : "th" );
  count++;

  /* Tell the function which called us where the
   * buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}


struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    4, /* Length of the file name */
    "test", /* The file name */
    S_IFREG | S_IRUGO, /* File mode - this is a regular
                        * file which can be read by its
                        * owner, its group, and everybody
                        * else */
    1, /* Number of links (directories where the
        * file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it
            * to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode
           * (linking, removing, etc.) - we don't
           * support any. */
    procfile_read, /* The read function for this file,
                    * the function called when somebody
                    * tries to read something from it. */
    NULL /* We could have here a function to fill the
          * file's inode, to enable us to play with
          * permissions, ownership, etc. */
```

```
  };



/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure , so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root is the root directory for the proc
   * fs (/proc). This is where we want our file to be
   * located.
   */
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

Chapter 4

Using /proc For Input

So far we have two ways to generate output from kernel modules: we can register a device driver and `mknod` a device file, or we can create a `/proc` file. This allows the kernel module to tell us anything it likes. The only problem is that there is no way for us to talk back. The first way we'll send input to kernel modules will be by writing back to the `/proc` file.

Because the proc filesystem was written mainly to allow the kernel to report its situation to processes, there are no special provisions for input. The `proc_dir_entry` struct doesn't include a pointer to an input function, the way it includes a pointer to an output function. Instead, to write into a `/proc` file, we need to use the standard filesystem mechanism.

In Linux there is a standard mechanism for file system registration. Since every file system has to have its own functions to handle inode and file operations[1], there is a special structure to hold pointers to all those functions, `struct inode_operations`, which includes a pointer to `struct file_operations`. In /proc, whenever we register a new file, we're allowed to specify which `struct inode_operations` will be used for access to it. This is the mechanism we use, a `struct inode_operations` which includes a pointer to a `struct file_operations` which includes pointers to our `module_input` and `module_output` functions.

It's important to note that the standard roles of read and write are reversed in the kernel. Read functions are used for output, whereas write functions are used for input. The reason

[1] The difference between the two is that file operations deal with the file itself, and inode operations deal with ways of referencing the file, such as creating links to it.

for that is that read and write refer to the user's point of view — if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

Another interesting point here is the `module_permission` function. This function is called whenever a process tries to do something with the `/proc` file, and it can decide whether to allow access or not. Right now it is only based on the operation and the uid of the current used (as available in `current`, a pointer to a structure which includes information on the currently running process), but it could be based on anything we like, such as what other processes are doing with the same file, the time of day, or the last input we received.

The reason for `put_user` and `get_user` is that Linux memory (under Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one of each of the processes.

The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The `put_user` and `get_user` macros allow you to access that memory.

procfs.c

```
/* procfs.c -  create a "file" in /proc, which allows
 * both input and output. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* The necessary header files */

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

```
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use proc fs */
#include <linux/proc_fs.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif

/* The module's file functions ********************** */


/* Here we keep the last message received, to prove
 * that we can process our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Since we use the file operations struct, we can't
 * use the special proc output provisions - we have to
 * use a standard read function, which is this function */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
static ssize_t module_output(
    struct file *file,   /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
    struct inode *inode, /* The inode read */
    struct file *file,   /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    int len)  /* The length of the buffer */
#endif
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];

  /* We return 0 to indicate end of file, that we have
   * no more information. Otherwise, processes will
   * continue to read from us in an endless loop. */
  if (finished) {
    finished = 0;
    return 0;
  }

  /* We use put_user to copy the string from the kernel's
   * memory segment to the memory segment of the process
   * that called us. get_user, BTW, is
   * used for the reverse. */
  sprintf(message, "Last input:%s", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);
```

```
  /* Notice, we assume here that the size of the message
   * is below len, or it will be received cut. In a real
   * life situation, if the size of the message is less
   * than len then we'd return len and on the second call
   * start filling the buffer with the len+1'th byte of
   * the message. */
  finished = 1;

  return i;  /* Return the number of bytes "read" */
}


/* This function receives input from the user when the
 * user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,   /* The file itself */
    const char *buf,     /* The buffer with input */
    size_t length,       /* The buffer's length */
    loff_t *offset)      /* offset to file - ignore */
#else
static int module_input(
    struct inode *inode, /* The file's inode */
    struct file *file,   /* The file itself */
    const char *buf,     /* The buffer with the input */
    int length)          /* The buffer's length */
#endif
{
  int i;

  /* Put the input into Message, where module_output
   * will later be able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
  /* In version 2.2 the semantics of get_user changed,
```

```
   * it not longer returns a character, but expects a
   * variable to fill up as its first argument and a
   * user segment pointer to fill it from as the its
   * second.
   *
   * The reason for this change is that the version 2.2
   * get_user can also read an short or an int. The way
   * it knows the type of the variable it should read
   * is by using sizeof, and for that it needs the
   * variable itself.
   */
#else
    Message[i] = get_user(buf+i);
#endif
  Message[i] = '\0';  /* we want a standard, zero
                       * terminated string */

  /* We need to return the number of input characters
   * used */
  return i;
}



/* This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for referece only, and can be overridden here.
```

```
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but
   * only root (uid 0) may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* If it's anything else, access is denied */
  return -EACCES;
}




/* The file is opened - we don't really care about
 * that, but it does mean we need to increment the
 * module's reference count. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;

  return 0;
}


/* The file is closed - again, interesting only because
 * of the reference count. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  MOD_DEC_USE_COUNT;
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}


/* Structures to register as the /proc file, with
 * pointers to all the relevant functions. ********** */



/* File operations for our proc file. This is where we
 * place pointers to all the functions called when
 * somebody tries to do something to our file. NULL
 * means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,    /* Somebody opened the file */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush, added here in version 2.2 */
#endif
    module_close,    /* Somebody closed the file */
    /* etc. etc. etc. (they are all given in
     * /usr/include/linux/fs.h). Since we don't put
     * anything here, the system will keep the default
     * data, which in Unix is zeros (NULLs when taken as
     * pointers). */
  };
```

```
/* Inode operations for our proc file. We need it so
 * we'll have some place to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to
 * an inode (although we don't bother, we just put
 * NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
    module_permission /* check for permissions */
  };


/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
```

```
    7, /* Length of the file name */
    "rw_test", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* File mode - this is a regular file which
     * can be read by its owner, its group, and everybody
     * else. Also, its owner can write to it.
     *
     * Actually, this field is just for reference, it's
     * module_permission that does the actual check. It
     * could use this field, but in our implementation it
     * doesn't, for simplicity. */
    1,  /* Number of links (directories where the
         * file is referenced) */
    0, 0,  /* The uid and gid for the file -
            * we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* A pointer to the inode structure for
     * the file, if we need it. In our case we
     * do, because we need a write function. */
    NULL
    /* The read function for the file. Irrelevant,
     * because we put it in the inode structure above */
  };



/* Module initialization and cleanup ****************** */

/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
```

```
   * inode number automatically if it is zero in the
   * structure , so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

Chapter 5

Talking to Device Files (writes and IOCTLs)

Device files are supposed to represent physical devices. Most physical devices are used for output as well as input, so there has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. In the following example, this is implemented by `device_write`.

This is not always enough. Imagine you had a serial port connected to a modem (even if you have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem, so you don't have to tax your imagination too hard). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to send the rate at which data is sent and received.

The answer in Unix is to use a special function called `ioctl` (short for **i**nput **o**utput **c**on**t**ro**l**). Every device can have its own `ioctl` commands, which can be read `ioctl`'s (to send information from a process to the kernel), write `ioctl`'s (to return information to a process), [1] both or neither. The ioctl function is called with three parameters: the file descriptor of the appropriate device file, the ioctl number, and a parameter, which is of type

[1] Notice that here the roles of read and write are reversed *again*, so in `ioctl`'s read is to send information to the kernel and write is to receive information from the kernel.

long so you can use a cast to use it to pass anything. [2]

The ioctl number encodes the major device number, the type of the ioctl, the command, and the type of the parameter. This ioctl number is usually created by a macro call (`_IO`, `_IOR`, `_IOW` or `_IOWR` — depending on the type) in a header file. This header file should then be `#include`'d both by the programs which will use `ioctl` (so they can generate the appropriate `ioctl`'s) and by the kernel module (so it can understand it). In the example below, the header file is `chardev.h` and the program which uses it is `ioctl.c`.

If you want to use `ioctl`'s in your own kernel modules, it is best to receive an official `ioctl` assignment, so if you accidentally get somebody else's `ioctl`'s, or if they get yours, you'll know something is wrong. For more information, consult the kernel source tree at '`Documentation/ioctl-number.txt`'.

chardev.c

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


/* Copyright (C) 1998-99 by Ori Pomerantz */



/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

[2] This isn't exact. You won't be able to pass a structure, for example, through an ioctl — but you will be able to pass a pointer to the structure.

```
/* For character devices */

/* The character device definitions are here */
#include <linux/fs.h>

/* A wrapper which does next to nothing at
 * at present, but may help for compatibility
 * with future versions of Linux */
#include <linux/wrapper.h>


/* Our own ioctl numbers */
#include "chardev.h"


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif



#define SUCCESS 0


/* Device Declarations ******************************** */
```

```
/* The name for our device, as it will appear in
 * /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message for the device */
#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 function is called whenever a process attempts
 * to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p)\n", file);
#endif

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

```
  /* If this was a process, we would have had to be
   * more careful here, because one process might have
   * checked Device_Open right before the other one
   * tried to increment it. However, we're in the
   * kernel, so we're protected against context switches.
   *
   * This is NOT the right attitude to take, because we
   * might be running on an SMP box, but we'll deal with
   * SMP in a later chapter.
   */

  Device_Open++;

  /* Initialize the message */
  Message_Ptr = Message;

  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* This function is called when a process closes the
 * device file. It doesn't have a return value because
 * it cannot fail. Regardless of what else happens, you
 * should always be able to close a device (in 2.0, a 2.2
 * device file could be impossible to close). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                           struct file *file)
#endif
{
#ifdef DEBUG
```

```
  printk ("device_release(%p,%p)\n", inode, file);
#endif

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

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}



/* This function is called whenever a process which
 * has already opened the device file attempts to
 * read from it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
    struct file *file,
    char *buffer, /* The buffer to fill with the data */
    size_t length,     /* The length of the buffer */
    loff_t *offset) /* offset to the file */
#else
static int device_read(
    struct inode *inode,
    struct file *file,
    char *buffer,   /* The buffer to fill with the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;
```

```
#ifdef DEBUG
  printk("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 ("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. */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                            const char *buffer,
                            size_t length,
                            loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

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

  for(i=0; i<length && i<BUF_LEN; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buffer+i);
#else
    Message[i] = get_user(buffer+i);
#endif

  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,
    struct file *file,
    unsigned int ioctl_num,/* The number of the ioctl */
    unsigned long ioctl_param) /* The parameter to it */
{
  int i;
  char *temp;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  char ch;
#endif

  /* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, temp);
      for (i=0; ch && i<BUF_LEN; i++, temp++)
        get_user(ch, temp);
#else
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
```

```
;
#endif

      /* Don't reinvent the wheel - call device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      device_write(file, (char *) ioctl_param, i, 0);
#else
      device_write(inode, file, (char *) ioctl_param, i);
#endif
      break;

    case IOCTL_GET_MSG:
      /* Give the current message to the calling
       * process - the parameter we got is a pointer,
       * fill it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      i = device_read(file, (char *) ioctl_param, 99, 0);
#else
      i = device_read(inode, file, (char *) ioctl_param,
                      99);
#endif
      /* Warning - we assume here the buffer length is
       * 100. If it's less than that we might overflow
       * the buffer, causing the process to core dump.
       *
       * The reason we only allow up to 99 characters is
       * that the NULL which terminates the string also
       * needs room. */

      /* 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 = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,  /* flush */
#endif
  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 = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);

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

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

  return 0;
}


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

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

  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_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.
 */

/* Copyright (C) 1998 by Ori Pomerantz */


/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "chardev.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;
  while (c != 0) {
    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);
  }
  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);
}
```