**Chapter 7. Talking To Device Files**

**7.1. 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 Input Output ConTroL). 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), [10] 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 long so you can use a cast to use it to pass anything. [11] 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 included 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 ioctls in your own kernel modules, it is best to receive an official ioctl assignment, so if you accidentally get somebody else's ioctls, or if they get yours, you'll know something is wrong. For more information, consult the kernel source tree at Documentation/ioctl−number.txt.

**Example 7−1. 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

\*/

The Linux Kernel Module Programming Guide

Chapter 7. Talking To Device Files 44

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);

}

**Example 7−2. 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

**Example 7−3. 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);

}