chardev2.c

```c
/*
 * chardev2.c - Create an input/output character device
 */

#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/poll.h>

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

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

/* Is the device open right now? Used to prevent concurrent access into
 * the same device
 */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

/* The message the device will give when asked */
static char message[BUF_LEN + 1];

static struct class *cls;

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

    try_module_get(THIS_MODULE);
    return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
    pr_info("device_release(%p,%p)\n", inode, file);

    module_put(THIS_MODULE);
    return SUCCESS;
```

```c
}

/* 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 */
    size_t length, /* length of the buffer */
    loff_t *offset)
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;
    /* 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.
    */
    const char *message_ptr = message;

    if (!*(message_ptr + *offset)) { /* we are at the end of message */
        *offset = 0; /* reset the offset */
        return 0; /* signify end of file */
    }

    message_ptr += *offset;

    /* 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 would not 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++;
    }

    pr_info("Read %d bytes, %ld left\n", bytes_read, length);

    *offset += bytes_read;

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

/* 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;
    pr_info("device_write(%p,%p,%ld)", file, buffer, length);

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

    /* 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.
 */
static long
device_ioctl(struct file *file, /* ditto */
unsigned int ioctl_num, /* number and param for ioctl */
unsigned long ioctl_param)
{
    int i;
    long ret = SUCCESS;

    /* We don't want to talk to two processes at the same time. */
    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;

    /* 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.
    */
        char __user *tmp = (char __user *)ioctl_param;
        char ch;

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

            device_write(file, (char __user *)ioctl_param, i, NULL);
        break;
    }
    case IOCTL_GET_MSG: {
        loff_t offset = 0;
```

```c
    /* Give the current message to the calling process - the parameter
    * we got is a pointer, fill it.
    */
    i = device_read(file, (char __user *)ioctl_param, 99, &offset);

    /* Put a zero at the end of the buffer, so it will be properly
    * terminated.
    */
    put_user('\0', (char __user *)ioctl_param + i);
    break;
    }
    case IOCTL_GET_NTH_BYTE:
    /* This ioctl is both input (ioctl_param) and output (the return
    * value of this function).
    */
    ret = (long)message[ioctl_param];
    break;
    }

    /* We're now ready for our next caller */
    atomic_set(&already_open, CDEV_NOT_USED);

    return ret;
}

/* 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.
 */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .unlocked_ioctl = device_ioctl,
    .open = device_open,
    .release = device_release, /* a.k.a. close */
};

/* Initialize the module - Register the character device */
static int __init chardev2_init(void)
{
    /* Register the character device (atleast try) */
    int ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);

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

    cls = class_create(THIS_MODULE, DEVICE_FILE_NAME);
    device_create(cls, NULL, MKDEV(MAJOR_NUM, 0), NULL, DEVICE_FILE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_FILE_NAME);

    return 0;
}

/* Cleanup - unregister the appropriate file from /proc */
static void __exit chardev2_exit(void)
{
    device_destroy(cls, MKDEV(MAJOR_NUM, 0));
    class_destroy(cls);

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

module_init(chardev2_init);
module_exit(chardev2_exit);

MODULE_LICENSE("GPL");
```

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 chardev2.c) and the process
 * calling ioctl() (in userspace_ioctl.c).
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/* The major device number. We can not 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 _IOW(MAJOR_NUM, 0, char *)
/* _IOW means that we are 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 are 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"
#define DEVICE_PATH "/dev/char_dev"

#endif
```

Makefile

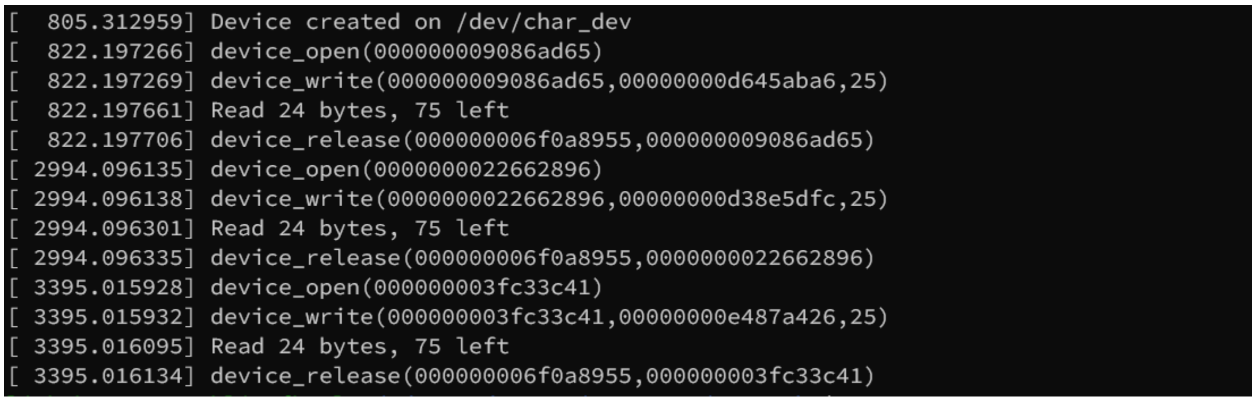
```
obj-m += chardev2.o

PWD := $(CURDIR)

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

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

userspace_ioctl.c

```
/* userspace_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> /* standard I/O */
#include <fcntl.h> /* open */
```

```
#include <unistd.h> /* close */
#include <stdlib.h> /* exit */
#include <sys/ioctl.h> /* ioctl */

/* Functions for the ioctl calls */

int 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);
    }

    return ret_val;
}

int ioctl_get_msg(int file_desc)
{
    int ret_val;
    char message[100] = { 0 };

/* 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);
    }
    printf("get_msg message:%s", message);

    return ret_val;
}

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

    printf("get_nth_byte message:");

    i = 0;
    do {
        c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
        if (c < 0) {
            printf("\nioctl_get_nth_byte failed at the %d'th byte:\n", i);
        return c;
        }

        putchar(c);
    } while (c != 0);

        return 0;
    }

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

    file_desc = open(DEVICE_PATH, O_RDWR);
    if (file_desc < 0) {
        printf("Can't open device file: %s, error:%d\n", DEVICE_PATH,
file_desc);
        exit(EXIT_FAILURE);
    }

    ret_val = ioctl_set_msg(file_desc, msg);
    if (ret_val)
        goto error;
    ret_val = ioctl_get_nth_byte(file_desc);
    if (ret_val)
        goto error;
    ret_val = ioctl_get_msg(file_desc);
    if (ret_val)
        goto error;

    close(file_desc);
    return 0;
error:
    close(file_desc);
    exit(EXIT_FAILURE);
 }
```

dmesg result



```
[  805.312959] Device created on /dev/char_dev
[  822.197266] device_open(000000009086ad65)
[  822.197269] device_write(000000009086ad65,00000000d645aba6,25)
[  822.197661] Read 24 bytes, 75 left
[  822.197706] device_release(000000006f0a8955,000000009086ad65)
[ 2994.096135] device_open(0000000022662896)
[ 2994.096138] device_write(0000000022662896,00000000d38e5dfc,25)
[ 2994.096301] Read 24 bytes, 75 left
[ 2994.096335] device_release(000000006f0a8955,0000000022662896)
[ 3395.015928] device_open(000000003fc33c41)
[ 3395.015932] device_write(000000003fc33c41,00000000e487a426,25)
[ 3395.016095] Read 24 bytes, 75 left
[ 3395.016134] device_release(000000006f0a8955,000000003fc33c41)
```