

# **Linux Driver Writing**

*Student Guide*

**Revision 1.0**

# **Linux Driver Writing**

## **Student Guide**

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.

Copyright ©2001 Object Innovations, Inc. All rights reserved.

Object Innovations, Inc.  
420 Boston Turnpike  
Shrewsbury, MA 01545  
(508) 845-1195  
[www.ObjectInnovations.com](http://www.ObjectInnovations.com)

Printed in the United States of America.

# Table of Contents

Chapter 1	Introduction to Linux Driver Development
Chapter 2	Device Drivers
Chapter 3	Linux Kernel Facilities
Chapter 4	Modules
Chapter 5	Character Devices
Chapter 6	Hardware Aspects
Chapter 7	Block Drivers
Chapter 8	Network
Chapter 9	Network Devices
Chapter 10	SCSI Subsystem
Chapter 11	Device Driver Debugging
Appendix A	Learning Resources
Appendix B	Data Structures
Appendix C	Labs
Appendix D	Solutions



# **Chapter 5**

## **Character Devices**

# Character Devices

## Objectives

---

*After completing this unit you will be able to:*

- **Understand what are the characteristics of a character device driver.**
- **Identify the components of a character device driver.**
- **Implement the methods needed in order to use a character device.**
- **Identify which methods must be implemented in a character device driver, and which ones do not absolutely need to be implemented.**

# Character Devices

---

- **Character devices are hardware components that read or write one character at a time, in a sequential order.**
  - They are accessed one bytes at a time or in arbitrary-sized blocks of data, without any automatic buffering mechanism.
  - Tape devices, serial and parallel ports are good examples of character devices.
  - Linux character drivers are located in the drivers/char directory of the Kernel tree.

## Major and Minor Numbers

---

- **Each character or block device is accessed through a file in the filesystem.**
  - This file is usually located in the /dev directory, which contains all device special files.
  - These files are represented with a “c” in the output of the “ls -l” command, or with a “b” for a block device.
  - The output of the “ls -l” command also gives the major and minor numbers of the device. These two fields are separated by a comma, as in the example bellow.

```
[/dev]$ ls null ttyS0 zero hda1 -l
brw-rw---- 1 root disk 3, 1 May 5 1998 hda1
crw-rw-rw- 1 root root 1, 3 May 5 1998 null
crw----- 1 root uucp 4, 64 Feb 15 11:15 ttyS0
crw-rw-rw- 1 root root 1, 5 May 5 1998 zero
```

- **The major number is an 8-bit number representing the device type.**
  - Due to its length (8 bits), this number cannot exceed 255.
  - Devices using the same driver are usually represented by the same major number on the system.
- **A minor number usually identifies a specific device among other devices sharing the same driver.**
  - This number is also stored on 8 bits, so the system is limited to 255 devices per major number.

## Major and Minor Numbers (Cont'd)

---

- **Most devices are associated with a fixed major number.**
  - Major number should not be associated with two different devices, since this could lead to a conflict.
  - Major numbers, along with their associated devices, are identified in Documentation/devices.txt.
- **Assigning a major number to a device is accomplished in the Kernel by invoking the `devfs_register_chrdev()` function when the driver is being initialized.**
  - This function will register the major number and associate it with the device name as well as with a `file_operations` structure, which are both given as the arguments.
  - This function adds the registered device to a table named `chrdevs`.
  - This table contains the maximum number of major numbers, i.e. 255. The major number of a specific device serves as the index of the table element.
  - Each element of the table contains the name of the device as well as a `file_operations` structure.
  - The name of the device will appear in `/proc/devices` upon successful completion of the `devfs_register_chrdev()` function.

## Major and Minor Numbers (Cont'd)

---

- **The `devfs_register_chrdev()` function may also register major number dynamically.**
  - This is useful if the driver is not specified in the Documentation/devices.txt file, and thus not bound to a fixed major number.
  - Dynamically assigning major number is typically used for custom devices that are not widespread enough to be assigned a fixed major number in the Linux Kernel.
  - Specifying 0 as the major number in the first argument of `devfs_register_chrdev()` will take the first unused major number in the `chrdevs` table for the new device major number.
  - The advantage of this method is that the new driver is assured to have a new, unused, major number.
  - Problems could however occur in the case that a module is loaded in the system afterwards and expects to assign itself a fixed major number. If the number has already been assigned by `devfs_register_chrdev()`, the module registration will fail.
- **Similarly, the `devfs_unregister_chrdev()` function is used when a character is unloaded from the Kernel.**

## Registration with DevFS

---

- **devfs\_register\_chrdev() and devfs\_unregister\_chrdev are functions provided for Kernels that do not use the new DevFS filesystem.**
  - The DevFS filesystem allows the Kernel to dynamically created files in the /dev directory.
  - The **register\_chrdev()** and **unregister\_chrdev()** functions are still available in 2.4, but they were typically used in 2.2 (or older) Kernels.
  - **devfs\_register\_chrdev()** and **devfs\_unregister\_chrdev()** acts merely as wrappers for the old functions. This is necessary in case the DevFS filesystem is not available.
- **The DevFS filesystem does not necessitate the devfs\_register\_chrdev() and devfs\_unregister\_chrdev functions.**
  - With DevFS, every files in the /dev directory could be created with **devfs\_register()**.
  - This new feature in the 2.4 Kernel avoids having to manually create files in /dev with the mknod command.
- **The Kernel source makes an heavy use of the devfs\_register() function in almost every driver.**
  - If the Kernel was not compiled with DevFS support, these functions will return NULL and will be useless.
  - Drivers may support both the old and new ways of registering devices in the Kernel.

## Registration with DevFS (Cont'd)

---

- The `devfs_register()` function is defined in the following way:

```
devfs_handle_t devfs_register (devfs_handle_t dir,
    const char *name,
    unsigned int namelen,
    unsigned int flags,
    unsigned int major,
    unsigned int minor,
    umode_t mode,
    uid_t uid,
    gid_t gid,
    void *ops,
    void *info);
```

- Device drivers may support both the static and dynamic allocation of `/dev` entries. The following is an example of how this could be done:

```
/* The old static major number allocation */
devfs_register_chrdev(DRUMS_MAJOR, "drums",
&drums_fops);

/* Create a directory that olds new device files */
drums_dir = devfs_mk_dir(NULL, "drums", 0, NULL);

for (i=0; i<DRUMS_NR_DEV; i++) {
    drums_devs[i]=devfs_register(drums_dir,
        drums_strings[i], DRUMS_NAME_LEN,
        DEVFS_FL_NONE, DRUMS_MAJOR, i,
        S_IFCHR | S_IRUGO, 0, 0,
        &drums_fops, NULL);
}
```

## Miscellaneous Devices

---

- **Spending a new major number for every character device driver loaded in the Kernel was not a careful way to save major numbers for other more important device drivers.**
  - A major number was thus assigned for similar character devices.
  - This major number 10 was assigned for this type of devices.
  - The drivers/char/misc.c file manages the registration of new miscellaneous devices in the system.
  - It allows many unrelated devices to work with the same major number.
  - The Kernel differentiates these devices by looking at their minor number.
  - Miscellaneous character device drivers are registered in the Kernel by using the `misc_register()` function instead of `register_chrdev()`.
  - For example, several mouse drivers and other character devices are registered in the Kernel as miscellaneous devices. The list of these devices may be found in `Documentation/devices.txt` in the field number 10.

# Opening Character Devices

---

- **A character device driver is first accessed by executing its `open()` function.**
  - This function is executed in the `sys_open()` system call.
  - The `open()` function will just not be called if it is not implemented by a specific device driver. This behavior may be observed in `dentry_open()`.
- **In the case that a character needs to implement its `open()` function, it will have to provide the following functionality:**
  - Increment the usage count, so that the driver may not be removed from the Kernel (in the case of a module).
  - Check for hardware-specific problems associated with this particular device.
  - Initialize the hardware, if it is needed.
  - Identify the minor number of the device that was open and update the `f_op` pointer if necessary. This is needed for device sharing the same major number but having different device drivers (i.e., miscellaneous devices).
  - Allocate the memory needed for the various data structures used in the device driver and initialize these structures.

## Closing Character Devices

---

- **A character device driver is closed when a User space application no longer needs it.**
  - This function is executed in the `sys_close()` system call.
  - The `release()` function will not be called if it is not implemented by a specific device driver. This behavior may be observed in `fput()`.
- **The `release()` function is in charge of the following steps:**
  - It decrements the usage count. This is necessary in order for the Kernel to be able to remove the module.
  - Removes any unnecessary data from memory. This is particularly true for data placed in the `private_data` field of the file structure associated with the device.
  - Shut down the physical device if needed. This includes any operation that must be executed in order to leave the hardware in a sane state, and disabling interrupts.
- **The `release()` function associated with a particular driver will not be invoked if the `open()` function was not called.**
  - Again, this may be observed in the `fput()` function.

# Reading Character Devices

---

- **read() is called to read data from the device. The form of the read function is as follows:**

```
static ssize_t device_read(struct file * file,  
                           char * buffer, size_t count, loff_t *ppos)
```

- **Note that the arguments of the read() method have changed in the 2.2 and subsequent Kernels.**
  - The new method passes only a file structure, from which we can find the disk inode associated with the device file.
- **The read() method implemented by a device driver should copy the specified number of bytes into the buffer and return the actual number of bytes read (or an error code).**
  - The read() function may therefore read less data than was requested. In this case the returned value will be less than size passed in parameter.
  - A negative return value means that there was an error.
  - Note that the buffer field passed to the device drivers refers to the memory space of the User space process that invoked the read() system call.
  - The copy\_to\_user() function must thus be used in order to
  - return the data in the proper memory segment.

## Reading Character Devices (Cont'd)

---

- The following is an example of the structure used in a simple read() method. It was taken from the PC Watchdog driver.

```
#define TEMP_MINOR          131

static ssize_t pcwd_read(struct file *file, char *buf, size_t count,
                        loff_t *ppos)
{
    unsigned short c;
    unsigned char cp;

    /* Can't seek on this device, so return an error */
    /* if the offset is not the same as current reading position */
    if (ppos != &file->f_pos)
        return -ESPIPE;

    /* Find the minor number associated with the device file*/
    switch(MINOR(file->f_dentry->d_inode->i_rdev))
    {
        /* Go ahead if this is the device we want to handle */
        case TEMP_MINOR:
            /*
             * Convert metric to Fahrenheit, since this was
             * the decided 'standard' for the return value.
             */
            c = inb(current_readport);
            cp = (c * 9 / 5) + 32;

            /* Copy the result in User space */
            if(copy_to_user(buf, &cp, 1))
                return -EFAULT;

            /*Only one byte can be read, thus return 1 */
            return 1;

        /* Return error if wrong minor number */
        default:
            return -EINVAL;
    }
}
```

# Writing Character Devices

---

- **The write() method implemented by a driver is invoked to write data to the device. It is defined as follows:**

```
static ssize_t device_write(struct file *file,
    const char *buf, size_t count, loff_t *ppos)
```

- **write() should copy the specified number of bytes from the User space buffer into the device.**
  - The number of bytes specified by the value of count should be written to the device.
  - Similarly to the read() method, the number returned by the write() function should match the value of count. If it's not the case, the data was partially written or, in the case of a negative value, an error occurred.
- **The following is an example of the implementation of a write() function. It does not address hardware issues, which we will reserve for the next chapters.**

```
static ssize_t pcwd_write(struct file *file, const char *buf, size_t
len, loff_t *ppos)
{
    /* We can't seek on this device */
    if (ppos != &file->f_pos)
        return -ESPIPE;

    if (len)
    {
        pcwd_send_heartbeat();
        return 1;
    }
    return 0;
}
```

## Character Devices - ioctl's

---

- **The `ioctl()` function processes `ioctl` calls sent by processes running in User space. It expects four arguments:**

```
int dev_ioctl(struct inode *inode, struct file
              *file, unsigned int cmd, unsigned long arg)
```

- A pointer to the inode object corresponding to the device special file which is being accessed.
  - A pointer to the file object corresponding to the device.
  - `ioctl` commands are represented as integers.
  - The command argument, which can be a pointer to a User space memory address.
- **The `ioctl()` function returns the value corresponding to the command, or a negative error number if the command failed.**
  - **The `-EINVAL` error number will be returned if the `ioctl()` function is not implemented by the driver, unless the command is one of the following:**
    - **`FIOCLEX`**, which sets the close-on-exec bit.
    - **`FIONCLEX`**, which clears the close-on-exec bit.
    - **`FIONBIO`** set the `O_NONBLOCK` flag if the argument is non-zero, and clears it otherwise.
    - **`FIOASYNC`** set the `O_SYNC` flag if the argument is non-zero, and clears it otherwise.

## Device Drivers: ioctl (Cont'd)

---

- **The ioctl() function in a device driver often takes the following form (taken from the standard cdrom device driver):**

```
static int cdrom_ioctl(struct inode *ip, struct file *fp, unsigned int
    cmd, unsigned long arg)
{
    kdev_t dev = ip->i_rdev;
    struct cdrom_device_info *cdi = cdrom_find_device(dev);
    struct cdrom_device_ops *cdo = cdi->ops;
    int ret;

    switch (cmd) {

    case CDROMCLOSETRAY: {
        cdinfo(CD_DO_IOCTL, "entering CDROMCLOSETRAY\n");
        if (!CDROM_CAN(CDC_CLOSE_TRAY))
            return -ENOSYS;
        return cdo->tray_move(cdi, 0);
    }

    case CDROMRESET: {
        if (!capable(CAP_SYS_ADMIN))
            return -EACCES;
        cdinfo(CD_DO_IOCTL, "entering CDROM_RESET\n");
        if (!CDROM_CAN(CDC_RESET))
            return -ENOSYS;
        return cdo->reset(cdi);
    }

    case CDROM_GET_CAPABILITY: {
        cdinfo(CD_DO_IOCTL, "entering CDROM_GET_CAPABILITY\n");
        return (cdo->capability & ~cdi->mask);
    }
    }
```

- **From the view of a User space process, the ioctl may be used in the following way:**

```
int cd_close(int cdrom) {
    int ret;
    ret = ioctl(cdrom, CDROMCLOSETRAY); // Send the ioctl command
    if (ret == -1) { // Handle error
        warn("can't close cd drive");
        return 0;
    }
    return 1;
}
```

## Device Drivers: lseek

---

- **When accessing a device in read or write mode, it may be necessary to move the “cursor” at which the read or write operation will begin.**
- **Very few character device drivers actually implement the lseek() method.**
- **The lseek() system call uses the lseek() function of the device on which the system call is invoked. This moves the read or write offset in the file.**
- **The lseek() function takes the following arguments:**
  - A pointer to the inode object corresponding to the device special file which is being accessed.
  - A pointer to the file object corresponding to the device.
  - The offset, which is the number of bytes from the origin of the device.
  - The definition of the origin, which could be the beginning of the device (0), from the current position (1) or from the end of the device (2).
- **The default action (as defined in the VFS) is to modify the f\_pos field in the file object corresponding to the device. This will only be applied if no implementation for lseek() is provided in the device driver.**

# Questions

---

- **What are the most commonly implemented functions in device drivers?**
- **You are in charge of developing the Linux Kernel driver for a new ISA-based data acquisition device. This simple device automatically updates a set of registers containing some information and no registers may be modified. What functions would you typically want to implement for such a device driver?**
- **An application needs a device that will output a given stream of bytes at a user-defined frequency. What would be the basic architecture of such a device (both at the hardware and software levels)?**

# Summary

---

- **Character device drivers usually consist of six well-defined methods. The implementation of these methods constitutes the actual device driver, and some methods may be left unimplemented if they do not make sense for a particular driver.**
- **The `open()` and `release()` methods are used for initializing and closing a device that is needed (or no longer needed) by a User space process.**
- **The `read()` and `write()` methods are used for accessing and modifying the data provided by a device attached to the system.**
- **The `ioctl()` method is used for managing a device, which usually consists in sending commands to the device.**
- **The `lseek()` method may be used to set the position pointer in a device.**

