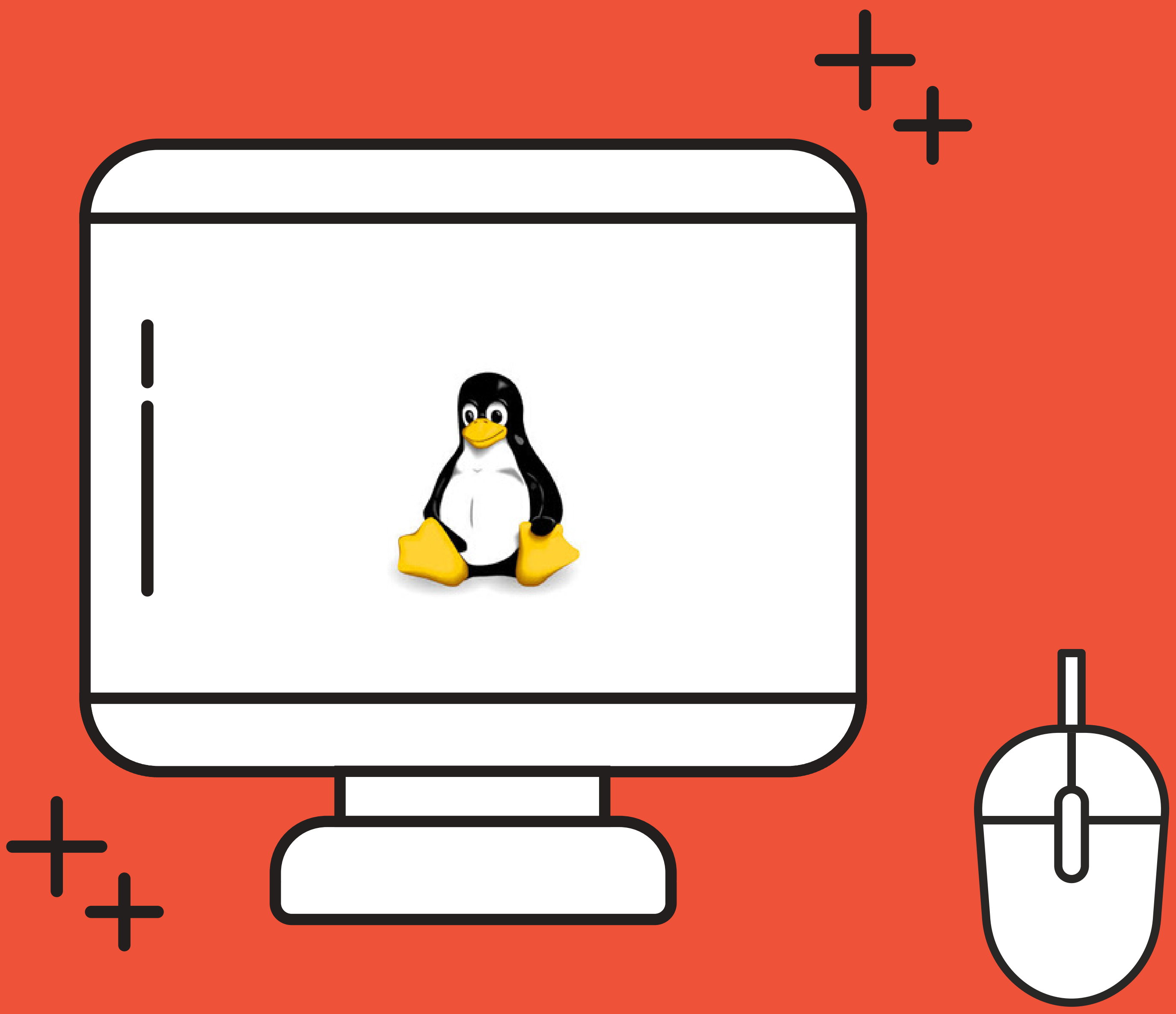


Linux Device Drivers for Dummies

A book on the basics of Device Drivers, for everyone.

SHUKRITHI RATHNA

ACKNOWLEDGEMENTS

I take this opportunity to extend my sincere gratitude to Prof. Sankaran Vaidyanthan for mentoring and guiding me through the entire course of Device Drivers. I would also like to thank my friends and family who have greatly helped me in writing this book.

PREFACE

In this book, we will explore the basics of device drivers in terms of what they are, the role they play in computer systems and why it is necessary for anyone using a computer to know about them. Towards the end of the book, we will also look at how to write your own very basic device driver and how to implement them. The main aim of this book is to stay away from technical terms wherever possible, and present the subject in a manner that is easily understandable by everyone.

CONTENTS

Part I

INTRODUCTION

1

INTRODUCTION TO DEVICE DRIVERS

More often than not, you would have come across the term 'Device Driver' when you were trying to troubleshoot some problem you had with any of your peripheral devices such as an external mouse, monitor, printer or modem. Device Drivers work quietly in the background and keep all your devices running smoothly and effectively

In Linux kernel device drivers are, essentially, a shared library of privileged, memory resident, low level hardware handling routines. It is Linux's device drivers that handle the peculiarities of the devices they are managing.

This approach allows the system to assume control of any connected hardware part without learning its details. With the common, understandable interface, it also becomes possible to build a two-way communication between the system (or the kernel) and the hardware.

1.1 WHAT IS A DEVICE DRIVER?

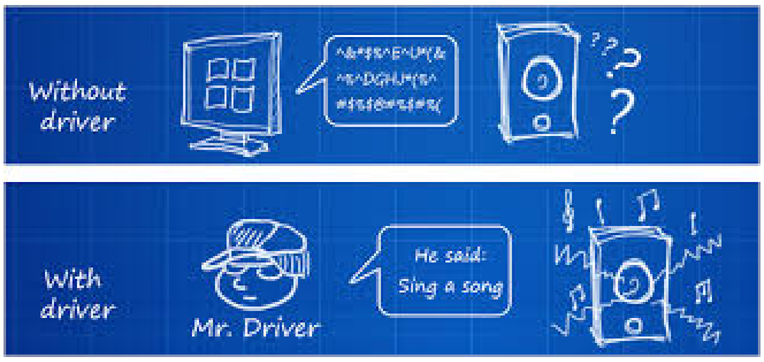
In technical terms, a device driver is is a kernel module that is responsible for managing the low-level I/O operations of a hardware device. The kernel is the part of the operating system that directly interacts with the physical structure of the system i.e, the underlying hardware.

But now let's try to use simple words to define what a device driver is so that we really understand them Basically, a device driver is a computer program that enables user to operate and control a particular type of device that is connected to the computer. It is the software that handles or manages a hardware controller. A device driver allows your computer's operating system to communicate with the hardware device that the driver is written for.

1.2 WHY DO WE NEED DEVICE DRIVERS?

Device drivers are essential for a computer to work properly. These programs may be compact, but they provide the all-important means for a computer to interact with hardware, for everything from mouse, keyboard and display (user input/output) to working with networks, storage and graphics.

A device driver is like a black box that means, if any user-application wants to interact with the hardware, it must go through the corresponding device driver only and not directly as it might cause any damage to the hardware.



1.3 TYPES OF DEVICE DRIVERS

Almost every device has its driver – starting from BIOS and up to various virtual machines developed by Microsoft. All device drivers can be divided into two main categories:

- kernel device drivers
- user device drivers.

Device drivers can also be classified by how they handle input and output operations. By this classification, there are three types of Device Drivers:

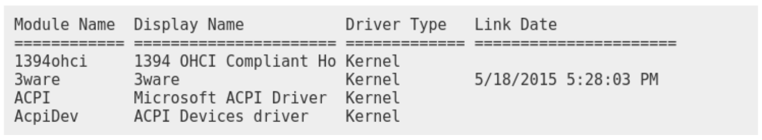
- Block Device Drivers - These are used to manage devices with physically addressable storage media, such as disks. In such devices, input or output data is handled in the form of asynchronous (discontinuous) chunk, instead of one constant stream of date

- Character Device Drivers - These are used for devices that perform input and output on a continuous flow of bytes.

More details about the different types of device drivers and their purpose are given in the second chapter.

1.4 Examples of device drivers

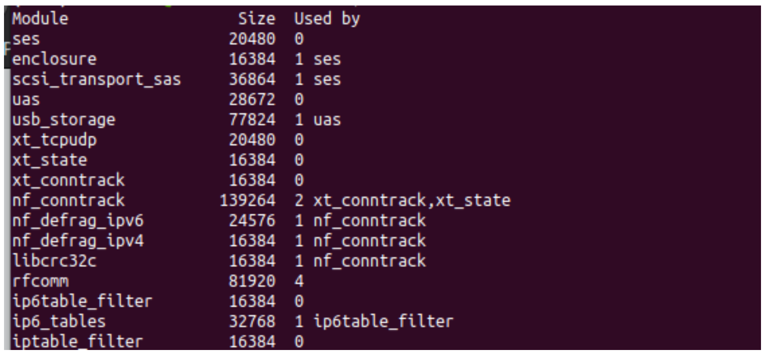
The 'driverquery' command in Windows will produce a list of all device drivers installed. A sample output is shown below



```
Module Name  Display Name           Driver Type   Link Date
============ ====================== ============= ======================
1394ohci     1394 OHCI Compliant Ho Kernel
3ware        3ware                  Kernel        5/18/2015 5:28:03 PM
ACPI         Microsoft ACPI Driver  Kernel
AcpiDev      ACPI Devices driver    Kernel
```

Figure 1: Device Drivers on Windows

Similarly, the 'lsmod' command on Linux will give you a similar output.



```
Module                  Size  Used by
ses                    20480  0
enclosure              16384  1 ses
scsi_transport_sas     36864  1 ses
uas                    28672  0
usb_storage            77824  1 uas
xt_tcpudp              20480  0
xt_state               16384  0
xt_conntrack           16384  0
nf_conntrack          139264  2 xt_conntrack,xt_state
nf_defrag_ipv6         24576  1 nf_conntrack
nf_defrag_ipv4         16384  1 nf_conntrack
libcrc32c              16384  1 nf_conntrack
rfcomm                 81920  4
ip6table_filter        16384  0
ip6_tables             32768  1 ip6table_filter
iptable_filter         16384  0
```

Figure 2: Device Drivers on Linux

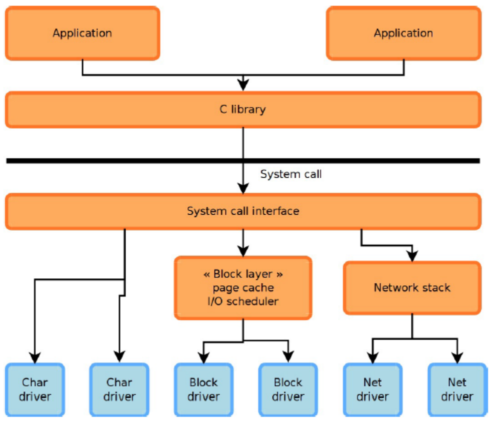
1.5 Applications

Modern day hardware and operating systems are very diverse in nature and because of this drivers can operate in many different environments. Generally, drivers interface with:
Printers
Video adapters
Network and Sound cards
Hard disk, CD-ROM, and floppy disk buses
Image scanners and Digital cameras

2

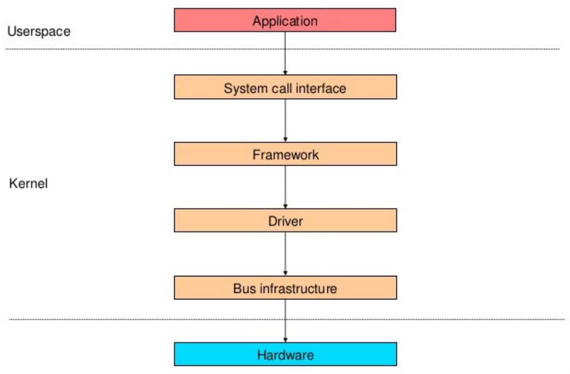
TYPES OF DEVICE DRIVERS

In this section, we will look at the different types of Device Drivers in more detail



2.1 KERNEL DEVICE DRIVERS

Drivers from this category are loaded with the operating system. They function as a part of your OS after they are loaded in the memory. However, the system loads not the full driver but a pointer. This method allows invoking any drive when it is needed. Kernel device drivers include drivers for BIOS, processor, motherboard, and other similar hardware.



However, there is one disadvantage with kernel device drivers: when one driver is invoked, it goes straight into the random-access memory (RAM) and cannot be transferred to the virtual memory. This means that you might experience a lag or drop in performance on your computer when multiple device drivers are loaded and running.

2.2 USER DEVICE DRIVERS

These types of drivers are triggered by users when you use the computer. They are like support software for the devices you connect to the computer. User device drivers can be installed on your disk so that they will be less hungry for the system resources. However, it is better to keep gaming devices like advanced mics and keyboards in the RAM

2.3 BLOCK AND CHARACTER DEVICE DRIVERS

These drivers are needed for proper writing/reading of the data on your computer. Writing and reading devices such as hard disk drives, USB flash drives, CD-ROMs, and so on are supported by these drivers. The type of the driver – block driver or character driver – depends on how it is used.

Character drivers are used in serial buses. At one given time they can only write one character (a byte). Character drivers step into play when a device is attached to a serial port. For example, an external mouse is a serial device that uses a serial port for connection. It also requires a device driver to work.

Block drivers are capable of writing/reading more than one character. Usually, they tend to form blocks of information and retrieve as much data as possible within block limits. The first device on block drivers that comes to mind is going to be your hard disk. Another example – your CD-ROM, but the kernel has to confirm device connection each time CD-ROM is used by any program.

2.4 GENERIC AND OEM DRIVERS

These are two other types of drivers that we haven't mentioned before. Generic device drivers come with the operating software. They can be used for one type of device regardless of the manufacturer.

These generic drivers often need to be replaced by more advanced drivers in some cases. These drivers are written by hardware manufacturers. They are called OEM device drivers and must be downloaded and installed manually after the installation of the operating system. Popular examples of such drivers are drivers for video cards by NVIDIA and AMD.

Another type of device driver that is worth mentioning are virtual device drivers. These are for the virtual devices that are installed on the computer. Virtual device drivers are that software to support virtual hardware operation.

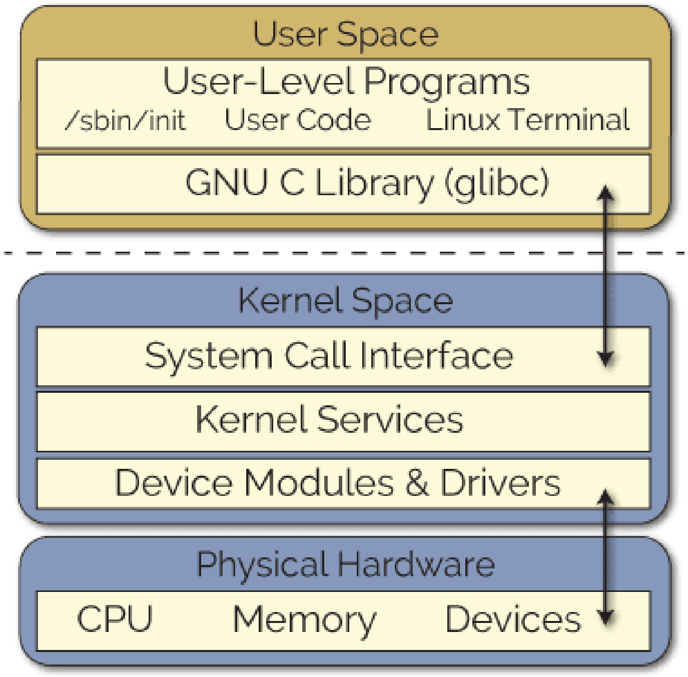
3

HOW DO DEVICE DRIVERS WORK?

3.1 COMMUNICATING WITH THE COMPUTER

When a user-application wants to interact with the hardware device, it reaches out to the device driver. the driver then checks for various conditions like the permissions and privileges of the user. After all the relevant conditions are met, the corresponding device driver responds and completes the task requested by the user-application (such as read/write to the device etc.). Finally, when the driver has completed its interaction with the hardware device, it responds back to the user-application with the information requested.

Generally a driver communicates with the device through the computer bus which is used to connect the device with the computer. Instead of accessing a device directly, an operating system loads the device drivers and calls the specific functions in the driver software in order to execute specific tasks on the device. Each driver contains the device specific codes required to carry out the actions on the device.



The most important thing about a device driver is that it runs in kernel space, with the same permissions as the kernel, and therefore can access hardware directly. Applications are (usually) not permitted to do that.

3.2 WORKING

Let's take an example of a printer, when it is connected to the computer and the specific device driver is installed, a device object is created on the computer which is designed to control the specific peripheral device it is connecting to. The driver represents the peripheral device and consists of a physical structure of modes that make up the process of allowing your operating system to control the peripheral device.

This device object represents the printer and its physical structure modes that allow your computer's operating system to control its functions. When we choose an operation (like Control + P to print a document) on the printer then this command goes to the device driver through the kernel of the operating system. Resultantly a calling program invokes a routine in the device driver and the driver issues corresponding commands to the microcontrollers within the printer. Further these microcontrollers control the components of the printer like motors etc. to start printing the document.

Part II

DEVELOPMENT

4 OVERVIEW

4.1 LINUX DEVICE DRIVERS

In a Linux based operating system, the programming interface is maintained in such a way that the device drivers can be written separately from the rest of the kernel and plugged in wherever necessary. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

Linux has a monolithic kernel. For this reason, writing a device driver for Linux requires performing a combined compilation with the kernel. Another way around is to implement your driver as a kernel module, in which case you won't need to recompile the kernel to add another driver. We'll be concerned with this second option: kernel modules.

4.2 KERNEL MODULES

At its base, a module is a specifically designed object file. When working with modules, Linux links them to its kernel by loading them to its address space. The Linux kernel was developed using the C programming language and Assembler. C implements the main part of the kernel, and Assembler implements parts that depend on the architecture.

Unfortunately, these are the only two languages we can use for writing Linux device drivers. We cannot use C++, which is used for the Microsoft Windows operating system kernel, because some parts of the Linux kernel source code – header files, to be specific – may include keywords from C++ (for example, delete or new), while in Assembler we may encounter lexemes such as ' : : '.

4.3 MODULE ERRORS

We run the module code in the kernel context. This requires a developer to be very attentive, as it entails extra responsibilities: if a developer makes a mistake when implementing a user-level application, this will not cause problems outside the user application in most cases; but if a developer makes a mistake when implementing a kernel module, the consequences will be problems at the system level.

The Linux kernel has a nice feature of being resistant to errors in module code. When the kernel encounters non-critical errors (for example, null pointer dereferencing), the oops message (insignificant malfunctions during Linux operation are called oops) is returned, after which the malfunctioning module will be unloaded, allowing the kernel and other modules to work as usual. In addition, a record that precisely describes this error can be found in the kernel log. Continuing work after an oops message is not recommended, as doing so may lead to instability and kernel panic.

4.4 LOADING AND UNLOADING MODULES

To create a simple sample module, we don't need to do much work. Here's some code that demonstrates this:

Listing 4.1: Sample Module

```c
#include <linux/init.h>
#include <linux/module.h>

static int my_init(void)
{
    return  0;
}

static void my_exit(void)
{
    return;
}

module_init(my_init);
module_exit(my_exit);
```

The only two things this module does is load and unload itself. To load a Linux driver, we call the my_init function, and to unload it, we call the my_exit function. The module_init and module_exit macros notify the kernel about driver loading and unloading. The my_init and my_exit functions must have identical signatures, which must be exactly as follows:

int init(void); void exit(void);

4.5 KERNEL VERSION

If the module requires a certain kernel version and must include information on the version, we need to link the linux/module.h header file. Trying to load a module built for another kernel version will lead to the Linux operating system prohibiting its loading. There's a reason for such behavior: updates to the kernel API are released quite often, and when you call a module function whose signature has been changed, you cause damage to the whole stack. The module_init and module_exit macros are declared in the linux/init.h header file.

5

REGISTERING A CHAR DEVICE

The example module in the previous chapter isvery simple. This section will show how to work with logging into the kernel and how to interact with device files. These tools may be simple, but they come in handy for any driver, and to some extent, they make the kernel-mode development process richer.

5.1 DEVICE FILES

Commonly, device files are found in the /dev folder. They facilitate interaction between the user and the kernel code. If the kernel must receive anything, you can just write it to a device file to pass it to the module serving this file; anything that's read from a device file originates from the module serving this file.

Device files van be divided into two groups: character files and block files. Character files are non-buffered, whereas block files are buffered. As their names imply, character files allow you to read and write data character by character, while block files allow you to write only whole blocks of data.

5.2 MAJOR AND MINOR NUMBERS

Linux systems have a way of identifying device files via major device numbers, which identify modules serving device files or a group of devices, and minor device numbers, which identify a specific device among a group of devices that a major device number specifies. In the driver code, we can define these numbers as constants or they can be allocated dynamically. In case a number defined as a constant has already been used, the system will return an error. When a number is allocated dynamically, the function reserves that number to prohibit it from being used by anything else.

5.3 SPECIFYING THE NAME OF THE DEVICE

The function cited below is used for registering character devices:

Listing 5.1: Register char device

```
int register_chrdev (unsigned int   major,
                     const char *   name,
                     const struct   fops);
                     file_operations *
```

The name and major number of a device are specified to register it, after which the device and the file_operations structure will be linked. If we assign zero to the major parameter, the function will allocate a major device number (i.e. the value it returns) on its own. If the value returned is zero, this signifies success, while a negative number signifies an error. Both device numbers are specified in the 0–255 range.

The device name is passed as a string value of the name parameter (this string can also pass the name of a module if it registers a single device). We then use this string to identify a device in the /sys/devices file.

6 THE FILE_OPERATIONS STRUCTURE

Device file operations such as read, write, and save are processed by the function pointers stored within the file_operations structure. These functions are implemented by the module and the pointers to the module structure identifying this module are also stored within the file_operations structure.

6.1 FILE_OPERATIONS

If the file_operations structure contains some functions that aren't required, you can still use the file without implementing them. A pointer to an unimplemented function can simply be set to be zero. After that, the system will take care of the implementation of the function and make it behave normally.

In this example, we're going to ensure the operation of only a single type of device with our Linux driver. This means the file_operations structure will be global and static. Correspondingly, after it's created, it will be filled statically.

Listing 6.1: File operations

```
static struct file_operations simple_driver_fops =
{
    .owner   = THIS_MODULE,
    .read    = device_file_read,
};
```

The declaration of the THIS_MODULE macro is contained in the linux/module.h header file. The macro will be transformed into the pointer to the module structure of the required module. The pointer to it which is device_file_read is obtained

```
ssize_t device_file_read (struct file *, char *, size_t, loff_t↩
    *);
```

6.2 MORE FUNCTIONS

The file_operations structure allows us to write several functions that will perform and revoke the registration of the device file.

```c
static int device_file_major_number = 0;
static const char device_name[] = "Simple-driver";
static int register_device(void)
{
        int result = 0;
        printk( KERN_NOTICE "Simple-driver: register_device() ↩
            is called." );
        result = register_chrdev( 0, device_name, &↩
            simple_driver_fops );
        if( result < 0 )
        {
            printk( KERN_WARNING "Simple-driver:  can\'t ↩
                register character device with errorcode = %i",↩
                 result );
            return result;
        }
        device_file_major_number = result;
        printk( KERN_NOTICE "Simple-driver: registered ↩
            character device with major number = %i and minor ↩
            numbers 0...255"
             , device_file_major_number );
        return 0;
}
```

The device_file_major_number is a global variable that contains the major device number. When the lifetime of the driver expires, this global variable will revoke the registration of the device file.

6.3 THE PRINTK FUNCTION

The declaration of this function is contained in the linux/kernel.h file, and its task is simple: to log kernel messages. All listed format strings of printk contain the KERN_NOTICE and KERN_WARNING prefixes. NOTICE and WARNING signify the priority level of a message. Levels range from the most insignificant KERN_DEBUG to the critical KERN_EMERG, alerting about kernel instability. This is the only difference between the printk function and the printf library function.

The printk function forms a string, which we write to the circular buffer, where the klog daemon reads it and sends it to the system log. The implementation of the printk function allows it to be called from anywhere in the kernel. The worst case scenario is the overflow of the circular buffer, meaning that the oldest message is not recorded in the log.

6.4 REVERTING REGISTRATION

If a device file is successfully registered, the value of the device_file_major_number will not be zero. This allows us to revoke the registration of the file using the nregister_chrdev function, which we declare in the linux/fs.h file. The major device number is the first parameter of this function, followed by a string containing the device name. The register_chrdev and the unresister_chrdev functions act in analogous ways.

To register a device, the following code is used:

```
void unregister_device(void)
{
    printk( KERN_NOTICE "Simple-driver: unregister_device() is ↩
        called" );
    if(device_file_major_number != 0)
    {
        unregister_chrdev(device_file_major_number, device_name↩
            );
    }
}
```

7

FUNCTION IMPLEMENTATION

7.1 SIGNATURE

The function in this example will read characters from a device. The signature of this function must be appropriate for that from the file_operations structure:

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
```

The first parameter is the pointer to the file structure. This file structure allows us to get necessary information about the file with which we're working, details on private data related to this current file, and so on. The data that has been read is allocated to the user space using the second parameter, which is a buffer. The number of bytes for reading is defined in the third parameter, and we start reading the bytes from a certain offset defined in the fourth parameter. After executing the function, the number of bytes that have been successfully read must be returned, after which the offset must be refreshed.

7.2 IMPLEMENTING THE READ FUNCTION

```

static const char    g_s_Hello_World_string[] = "Hello world ↩
    from kernel mode!\n\0";
static const ssize_t g_s_Hello_World_size = sizeof(↩
    g_s_Hello_World_string);
static ssize_t device_file_read(
                        struct file *file_ptr
                       , char __user *user_buffer
                       , size_t count
                       , loff_t *position)
{
    printk( KERN_NOTICE "Simple-driver: Device file is read at ↩
        offset = %i, read bytes count = %u"
                , (int)*position
                , (unsigned int)count );
```

```c
    /* If position is behind the end of a file we have nothing to read */
    if( *position >= g_s_Hello_World_size )
        return 0;
    /* If a user tries to read more than we have, read only as many bytes as we have */
    if( *position + count > g_s_Hello_World_size )
        count = g_s_Hello_World_size - *position;
    if( copy_to_user(user_buffer, g_s_Hello_World_string + *position, count) != 0 )
        return -EFAULT;
    /* Move reading position */
    *position += count;
    return count;
}
```

8

BUILDING THE KERNEL MODULE

8.1 INITIALIZING THE KERNEL BUILD SYSTEM:

Today, much of the work in building the module is done by the makefile: it starts the kernel build system and provides the kernel with the information about the components required to build the module. A module built from a single source file requires a single string in the makefile. After creating this file, you need only to initiate the kernel build system:

```
obj-m := source_file_name.o
```

The make command initializes the kernel build system:
To build the module:

```
make   C   KERNEL_MODULE_BUILD_SYSTEM_FOLDER M=‘pwd‘ modules
```

To clean up the build folder:

```

make   C   KERNEL_MODULES_BUILD_SYSTEM_FOLDER M=‘pwd‘ clean
```

The module build system is commonly located in /lib/modules/‘uname -r‘/build. To build the first module, execute the following command from the folder where the build system is located:

$$> makemodules_prepare$$

8.2 MAKEFILE

Everything is combined into a single makefile

```
TARGET_MODULE:=simple-module
# If we are running by kernel building system
ifneq ($(KERNELRELEASE),)
    $(TARGET_MODULE)-objs := main.o device_file.o
    obj-m := $(TARGET_MODULE).o
# If we running without kernel build system
```

```
else
    BUILDSYSTEM_DIR:=/lib/modules/$(shell uname -r)/build
    PWD:=$(shell pwd)
all :
# run kernel build system to make module
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules
clean:
# run kernel build system to cleanup in current directory
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) clean
load:
    insmod ./$(TARGET_MODULE).ko
unload:
    rmmod ./$(TARGET_MODULE).ko
endif
```

The load target loads the build module and the unload target deletes it from the kernel.

8.3 LOADING AND USING THE MODULE

The following command executed from the source file folder allows us to load the built module:

$$> makeload$$

After executing this command, the name of the driver is added to the /proc/modules file, while the device that the module registers is added to the /proc/devices file. The added records look like this:

Character devices: 1 mem 4 tty 4 ttyS . . . 250 Simple-driver . . .

The first three records contain the name of the added device and the major device number with which it's associated. The minor number range (0–255) allows the device files to be created in the /dev virtual file system.

$$> \; mknod \;/dev/simple - driver \; c \; 250\,0$$

After the device file is created, the cat command to display the contents and verify that the build was successful

$$\$ > \; cat \;/dev/simple - driver$$

Hello world from kernel mode!

Part III

REFERENCES

9

BIBILIOGRAPHY

Books:

(1) Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

(2) The Linux Kernel Module Programming Guide by Peter Jay Salzman and Ori Pomeranz:

Websites:

(1) $en.wikipedia.org/wiki/Device_driver$

(2) $searchenterprisedesktop.techtarget.com/definition/device-driver$

(3) $www.drivers.com/update/drivers-news/what-device-drivers//$

(4) $static.lwn.net/images/pdf/LDD3/ch01.pdf$

(5) $www.engineersgarage.com/how_to/how-device-drivers-work/$

(6) $docs.oracle.com/cd/E23824_01/html/819-3196/eqbqp.html$

(7) $www.apriorit.com/dev-blog/195-simple-driver-for-linux-os$

(8) $www.vlsifacts.com/device-drivers-role-and-types/$

INDEX