**Linux Device Drivers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module** | **Linux Device Drivers** | | | |
| **Sl.No** | **Topics / Subtopics** | **Session No:** | **Lab:** | **Page No:** |
| **1** | **INTRODUCTION TO DEVICE DRIVERS** | 1 | y | 4 |
|  | Introduction to Linux |  |  | 4 |
|  | Roll of Device Drivers |  |  | 6 |
|  | Types of Drivers |  |  | 6 |
|  | Kernel versioning |  |  | 6 |
|  | Writing the kernel module |  |  |  |
|  | Compiling and Loading modules |  |  | 11 |
|  | Important commands |  |  | 12 |
|  | module parameters | 2 |  | 15 |
|  | Debugging by Printing |  |  | 17 |
|  | Using the /proc Filesystem |  |  | 19 |
|  | sysfs |  |  | 20 |
| **2** | **CHAR DRIVERS** | 3 | y | 21 |
|  | What is a character device |  |  | 21 |
|  | Device Numbers |  |  | 21 |
|  | How to create a device file |  |  | 21 |
|  | registration and unregistration functions |  |  | 22 |
|  | Char Device Registration |  |  | 26 |
|  | Struct cdev |  |  | 26 |
|  | container\_of macro |  |  | 27 |
|  | open and release | 4 |  | 27 |
|  | read and write |  |  | 28 |
|  | The Current Process |  |  | 31 |
| **3** | **ADVANCED CHAR DRIVER OPERATIONS** | 5 | y | 32 |
|  | Seeking a Device |  |  | 32 |
|  | The IOCTL interface |  |  | 32 |
|  | The Implementation of the ioctl Commands |  |  | 34 |
|  | Blocking I/O |  |  | 35 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl.No** | **Topics / Subtopics** | **Session No:** | **Page No:** | **Lab** |
| **4** | **LINUX KERNEL BUILD & STATIC LINKING** | 6 | y | 2 |
|  | Compile Linux Kernel from Source to Build Custom Kernel |  |  | 37 |
|  | How to add a static module to the kernel |  |  | 37 |
| **5** | **FLOW OF TIME** | 7 | y | 40 |
|  | Time intervals in the kernel |  |  | 40 |
|  | Delaying Execution |  |  | 40 |
|  | Current time |  |  | 42 |
|  | Tasklets |  |  | 44 |
|  | Workqueues |  |  | 44 |
|  | Kernel Timers |  |  | 46 |
| **6** | **HARDWARE MANAGEMENT** | 8 | y | 48 |
|  | Using I/O Ports |  |  | 48 |
|  | Using I/O Memory | 9 |  | 49 |
| **7** | **INTERRUPT HANDLING** | 10 | y | 52 |
|  | Interrupts and Exceptions |  |  | 52 |
|  | Installing an Interrupt Handler |  |  | 52 |
|  | Writing a BH Bottom Half |  |  | 56 |
|  |  |  |  |  |

|  |
| --- |
|  |

# CHAPTER I

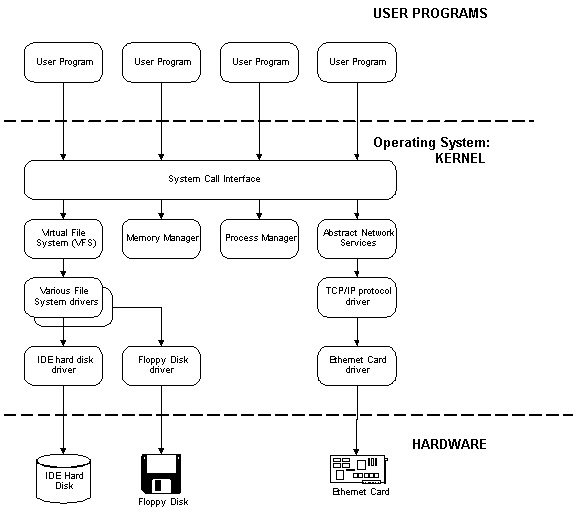
## INTRODUCTION TO DEVICE DRIVERS

### Introduction to Linux:

Linus B. Torvalds wrote the first Linux kernel in 1991 and made its source code freely available on internet. Linux kernel is coded in GNU C. Linux gained in popularity because it has always been distributed as free software unlike windows. Since the source code is readily available, users can freely change the kernel to suit their needs. However, it is important to understand how the Linux kernel has evolved and how it currently works. Linux supports nearly twenty hardware platforms including popular x86, AMD, SUN, SPARC, and Motorola.

### What is a Device Driver?

Device driver is a software which gives a device its behavior. Device driver is a piece of software which gives intelligence to hardware to make work as it should.

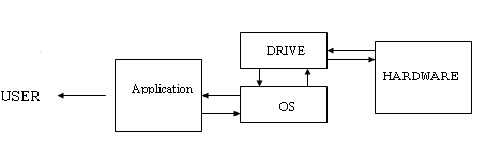


A device driver, often called a driver for short, is a computer program that enables another program, typically, an operating system (OS) (e.g., Windows, Linux, FreeBSD) to interact with a hardware device. A driver is essentially a set of instructions that provides the operating system with the information on how to control and communicate with a particular piece of hardware. In layman's terms, a driver is an important, vital piece to a program application; the main ingredients of the system

Device drivers serve several different purposes. In their purest form, they are the link between software and hardware.

For example, applications generally do not care where on a storage device a file resides; instead, they submit generic calls to the operating system to open, access, and close a file, and it is the task of the operating system and the device drivers to cooperatively locate the file on the storage device and read from or write to the correct physical locations of the device.

The fig depicts the control flow of a user request to the hardware device graphically. The advantage of this architecture is that no change needs to be made to the application if the hardware is changed.



DRIVERR

**Fig:1.1**

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don’t force particular policies on the user, since different users have different needs.

Device drivers take on a special role in the Linux kernel. They make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.

This programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

A device driver is a “C’ program that controls a device. The device can be a physical device (such as a disk) or a virtual device (such as a RAM disk). However, unlike a C program, you do not link a device driver into an executable program as it does not have a main( ) function - meaning that a device driver does not have a single entry point.

Each driver is different; as a driver writer, you will need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers.

When writing drivers, a programmer should pay particular attention access the hardware, but don’t force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about *how* to use the hardware to the applications.

### Why to write device drivers in linux?

There are a number of reasons to be interested in the writing of Linux device drivers.The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future.

Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works and why things aren’t always as fast as you expect or don’t do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn’t need special hardware to be tested.

### Device Drivers can be classified into

**Statically linked driver:** whose object code is linked with the kernel. The code of such device driver is physically contained in the kernel and therefore loaded in memory when the system boots.

**Dynamically linked driver**: whose object code is NOT linked with the kernel. The code of such device driver is NOT contained in the kernel, and the device driver is loaded and unloaded as and when required

### Advantages of Dynamic Loading:

Linux device drivers can be integrated into the kernel in two different ways : either by compiling them into the kernel so that they are always available, or by compiling them into an object format that the kernel can load whenever access to a specific device is required. Kernel code that can be automatically loaded into the kernel is referred to as a loadable kernel module. When configuring the Linux kernel, each kernel configuration editor displays a description of available kernel configuration variables and enables you to specify whether each should be deactivated, compiled into the kernel, or compiled as a loadable kernel module.

Compiling device drivers into the kernel has the advantage that they are always instantly available, but each increases the size of the kernel that you are running. Compiling device drivers as loadable kernel modules implies some slight overhead when you search for and initially load the module in order to access the associated device, plus some small runtime overhead, but these are negligiblecompared to thesavings in size and associated memory requirements. Writing device drivers as loadable kernel modules also provides significant advantages during development. As you develop and debug your device driver, you can dynamically unload the previous version and load the new version each time you want to test the new version. If your device driver is compiled into the kernel, you have to recompile the kernel and reboot each time that you want to test a set of iterative changes. Similarly, developing and deploying device drivers as loadable kernel modules simplifies maintaining them in the field, since the device driver can be updated as a separate system component without requiring a kernel update.

Configuring your kernel for support for loadable modules is done in the Loadable module support section of your kernel configuration editor. The Automatic kernel module loading option determines whether the kernel will automatically try to locate and load modules as they are needed by new devices or subsystems.

The Module versioning support option (marked as experimental in the current 2.6 kernel source) adds extra versioning information to compiled modules at build-time. This information is designed to help increase module portability to kernels other than the one that they were compiled against. The Module unloading option is new to 2.6 kernel support for loadable kernel modules. You must enable this option if you want your kernel to be able to unload modules when they are no longer needed. This is especially important in resource-constrained and power-sensitive environments such as embedded systems. If you activate this option, you can also activate the Forced module unloading option, which enables you to forcibly unload modules regardless of whether the kernel believes that they are in use.

**Linux Kernel Versions:**

1. **Stable kernels** : these are production level releases suitable for widespread

deployment

**Ex: 3.10.18**

3 is Major version no.

10 is minor version no.

Changed only when there drastic changes to kernel i.e from 3.9 to 3.10

18 is first release

These are releases with bugfixes.

1. **Development kernels:** these undergo rapid change where anything goes, often

drastic changes to the kernel are made

**Ex: 3.9.18**

### What Is Kernel?

Kernel is core part of operating system which is always running.

It includes:

* Process control subsystem
* File subsystem
* Memory management subsystem
* IPC
* Scheduler
* Device Drivers

Device drivers are part of the kernel which deal with yhe hardware. All the requests from

user are ultimately handeled by device drivers.

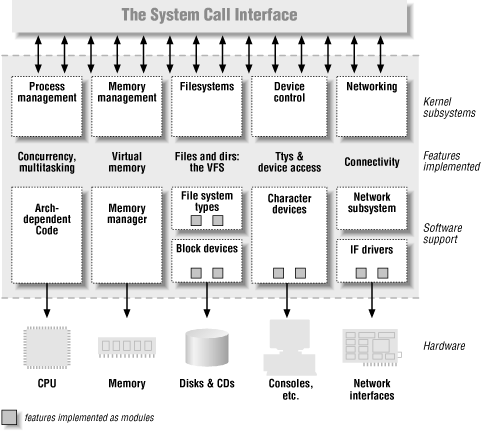
### Features of linux kernel:

1. Does not have access to C library.
2. Kernel stack is fixed
3. Kernel cannot easily use floating point
4. No memory protection in kernel space.

### Splitting the Kernel

In a UNIX system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource.

The *kernel* is the big chunk of executable code in charge of handling all such requests. Though the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split, as shown in Figure 1-1, into the following parts:



**Figure 1-2. A split view of the kernel**

### Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel.

In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

### Memory management

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc*/*free* pair to much more exotic functionalities.

### Filesystems

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, diskettes may be formatted with either the Linux-standard ext2 filesystem or with the commonly used FAT filesystem.

### Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape streamer. This aspect of the kernel's functions is our primary interest in this book.

### Networking

Networking must be managed by the operating system because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

### Classes of Devices and Modules

The UNIX way of looking at devices distinguishes between three device types. Each module usually implements one of these types, and thus is classifiable as a *char* module, a block module, or a network module. The classes are the following:

* Character devices
* Block devices
* Network devices
* Pseudo device drivers

### 

### Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

### Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device -- it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node and the difference between them is transparent to the user.

### Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data packets.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program

### Pseudo device driver

Not all device drivers control physical hardware. Such device drivers are called “Pseudo device drivers” or just drivers. Like block & character device drivers, pseudo device drivers make use of device drivers interfaces. Unlike block & character device drivers, pseudo device drivers do not operate on the bus. One example of pseudo device drivers is the pseudo terminal or pty terminal driver, which simulates a terminal device. The pty terminal driver is a character device driver typically used for remote logins.

### When is a device driver called ?

A device driver is called during:

* Autoconfiguration

The kernel calls a device driver at autoconfiguration time to determine what devices are available & to initialize them.

* I/O operations

The kernel calls a device driver to perform I/O operations on the device. These operations include opening the device to perform reads & writes & closing the device.

* Interrupt handling

The kernel calls a device driver to handle interrupts from devices capable of generating them.

* The kernel calls a device driver to handle special requests through ioctl calls
* Reinitialisation

The kernel calls the device driver to reinitialize the driver, the device, or both when the bus is reset.

Some of these requests, such as input or output, result directly or indirectly from corresponding system calls in a user program. Other requests, such as the calls at autoconfiguration time, do not result from system calls but from activities that occur at boot time.

### Security Issues:

A kernel module can do anything, A module is just as powerful as a superuser shell, any security check in the system is enforced by the kernel code, if the kernel code has security holes, then the system has holes**,** The system call *create\_module* checks if the invoking process is authorized to load a module into the kernel, thus, when running an official kernel, only the superuser can load a module into the kernel.

### Writing the kernel module

### What is a module?

Module is a program that is dynamically linked to the kernel and runs in the kernel space.

The below code shows a simple “hello world” module

# include <linux/module.h>

#include <linux/init.h>

int my\_init(void) /\*ENTRY POINT FOR THE MODULE \*/

{

printk("<1>Hello, world\n");

return 0;

}

void my\_cleanup(void) ) /\*EXIT POINT FOR THE MODULE

{

printk("<1>Goodbye world\n");

}

module\_init(my\_init); /\*macro specifying for your entry point

module\_exit(my\_cleanup); ); /\*macro specifying for your exit point

### How to compile a module:

Once you have everything set up, creating a makefile for your module is straightforward.

for the “hello world” example shown earlier in this chapter, a single line will suffice: EXAMPLE: Create a simple makefile by

root# vi Makefile

> obj-m := hello.o #inside the makefile type this and save your file

Execute the command given below on the shell to generate your object file with filename.ko extension

root# make -C /lib/modules/2.6<version>/build M=`pwd` modules

The assignment above (which takes advantage of the extended syntax provided by GNU *make*) states that there is one module to be built from the object file *hello.o*. The resulting module is named *hello.ko* after being built from the object file.

If, instead, you have a module called *module.ko* that is generated from two source files (called, say, *file1.c* and *file2.c*), the correct incantation would be:

obj-m := module.o

module-objs := file1.o file2.o

For a makefile like those shown above to work, it must be invoked within the context of the larger kernel build system. If your kernel source tree is located in, say, your *~/kernel-version directory*, the *make* command required to build your module (typed in the directory containing the module source and makefile) would be:

make -C /lib/modules/<version>/build M=`pwd` modules

This command starts by changing its directory to the one provided with the – C option (that is, your kernel source directory). There it finds the kernel’s top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable, which we’ve set to *module.o* in our examples. Typing the previous *make* command can get tiresome after a while, so the kernel developers have developed a sort of makefile idiom, which makes life easier for those building modules outside of the kernel tree. Write your makefile as follows:

### Writing a complete makefile to compile a module:

**#vi Makefile**

obj-m := hello.o

KERNELDIR? = /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

The printk function is defined in the Linux kernel and behaves similarly to the standard C library function printf. As memtioned earlier kernel does not have access to c library and kernel needs its own printing. The module can call printk because, after insmod has loaded it, the module is linked to the kernel and kernel libraries which is needed by printk. The string <1> is the priority of the message. ( printk discussed later in debugging).

### How to insert module in kernel space:

You can insert the module in kernel space by calling insmod (loading the module) and rmmod (unloading the module). Note that only the superuser can load and unload a module. This mechanism of inserting a module in the running kernel is called “dynamic loading”.

After compiling the module as memtioned earlier insert the module in kernel space.

root# **insmod hello.ko**

### Removing the module from kernel space :

root# **rmmod hello**

### Listing the modules:

root# **lsmod**

The lsmod program produces a list of the modules currently loaded in the kernel.

Some other information, such as any other modules making use of a specific module,

is also provided. lsmod works by reading the /proc/modules virtual file. Information

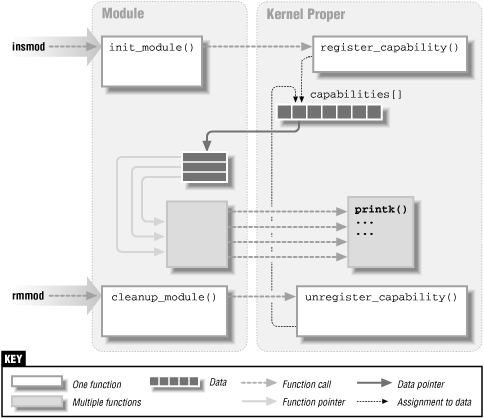
on currently loaded modules can also be found in the sysfs virtual filesystem

under /sys/module.

### Kernel Modules versus Applications

* An application program performs a single task from beginning to end, a module registers itself in order to serve future requests, and its "main" function terminates immediately.
* In other words, the task of the function *init\_module* (the module's entry point) is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do."
* The second entry point of a module, *cleanup\_module*, gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." The ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.
* An application program uses the library function, the linking stage resolves external references using the appropriate library of functions. *printf* is one of those callable functions and is defined in *libc*.
* A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The *printk* function used in *hello.c* earlier, for example, is the version of *printf* defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

Figure 2-1 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.



**Figure 2-1. Linking a module to the kernel**

Because no library is linked to modules, source files should *never* include the usual header files. Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in *include/linux* and *include/asm* inside the kernel sources (usually found in */usr/src/linux*). Older distributions (based on *libc* version 5 or earlier) used to carry symbolic links from */usr/include/linux*and */usr/include/asm* to the actual kernel sources, so your *libc* include tree could refer to the headers of the actual kernel source you had installed. These symbolic links made it convenient for user-space applications to include kernel header files, which they occasionally need to do.

### Name Space Pollution:

Namespace pollution is what happens when there are many functions and global variables whose names aren't meaningful enough to be easily distinguished.

Developers can't afford to fall into such an error when writing kernel code because even the smallest module will be linked to the whole kernel. The best approach for preventing namespace pollution is to declare all your symbols as static and to use a prefix that is unique within the kernel for the symbols you leave global. Also note that you, as a module writer, can control the external visibility of your symbols.

Most versions of *insmod* (but not all of them) export all non-static symbols if they find no specific instruction in the module; that's why it's wise to declare as static all the symbols you are not willing to export.

### The kernel symbol table:

The table contains the addresses of global kernel items functions and variables that are needed to implement modularized drivers. The public symbol table can be read in text form from the file.

**Command to see kernel symbols:**

root# cat */proc/ ksyms or root# ksyms*

### The Usage Count

The system keeps a usage count for every module in order to determine whether the module can be safely removed. The system needs this information because a module can't be unloaded if it is busy: you can't remove a filesystem type while. the filesystem is mounted, and you can't drop a char device while a process is using it, or you'll experience some sort of segmentation fault or kernel panic when wild pointers get dereferenced. In modern kernels, the system can automatically track the usage count for you.

### Passing Parameters to modules:

Several parameters that a driver needs to know can change from system to system. These parameter values can be assigned at load time by *insmod* or *modprobe(*will be discussed shortly*)*; the latter can also read parameter assignment from its configuration file (*/etc/modprobe.*  *conf*). The commands accept the specification of several types of values on the command line. As a way of demonstrating this capability, imagine a much-needed enhancement to the “hello world” module (called *hellop*) shown at the beginning of this chapter. We add two parameters: an integer value called howmany and a character string called whom. Such a module could then be loaded with a command line such as:

insmod hellop howmany=10 whom="Mom"

insmod hellop howmany=10 // when loaded with fewer parameters the, the module works with the values provided in the code.

Upon being loaded that way, *hellop* would say “Hello, Mom” 10 times. However, before *insmod* can

change module parameters, the module must make them available. Parameters are declared with the module\_param macro, which is defined in *moduleparam.h*. module \_ param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. So *hellop* would declare its parameters and make them available to *insmod* as follows:

static char \*whom = "world";

static int howmany = 1;

module\_param (howmany, int, S\_IRUGO);

module\_param (whom, charp, S\_IRUGO);

Numerous types are supported for module parameters:

bool, charp, int , long, short, uint, ulong, ushort.

A Boolean (true or false) value (the associated variable should be of type int). A char pointer value . Basic integer values of various lengths. The versions starting with u are for unsigned values. Memory is allocated for user-provided strings, and the pointer is set accordingly.

Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

module\_param\_array (name,type,num,perm);

Where name is the name of your array (and of the parameter), type is the type of the array elements, num is an integer variable, and perm is the usual permissions value. If the array parameter is set at load time, num is set to the number of values supplied.

The module loader refuses to accept more values than will fit in the array. If you really need a type that does not appear in the list above, there are hooks in the module code that allow you to define them; see *moduleparam.h* for details on how to do that. All module parameters should be given a default value; *insmod* changes the value only if explicitly told to by the user. The module can check for explicit parameters by testing parameters against their default values. The final *module\_param* field is a permission value; you should use the definitions found in *<linux/stat.h>*. This value controls who can access the representation of the module parameter in sysfs. If perm is set to 0, there is no sysfs entry at all; otherwise, it appears under */sys/module*\* with the given set of permissions. Use S\_IRUGO for a parameter that can be read by the world but cannot be changed; S\_IRUGO|S\_IWUSR allows root to change the parameter. Note that if a parameter is changed by sysfs, the value of that parameter as seen by your module changes, but your module is not notified in any other way. You should probably not make module parameters writable, unless you are prepared to detect the change and react accordingly

### Macros for Documentation:

**MODULE\_AUTHOR (name)**

Puts the author's name into the object file

**MODULE\_DESCRIPTION (desc)**

Puts a description of the module into the object file

**MODULE\_SUPPORTED\_DEVICE (dev)**

Places an entry describing what device is supported by this module. Comments in the kernel source suggest that this parameter may eventually be used to help with automated module loading, but no such use is made at this time.

### Macros for to manage the visibility of your symbols

**EXPORT\_SYMBOL(name);**

**EXPORT\_SYMBOL\_GPL(name);**

Either of the above macros makes the given symbol available outside the module. The \_GPL version makes the symbol available to GPL-licensed modules only.

### Other Macros:

**MODULE\_LICENSE("GPL");**

The specific licenses recognized by the kernel are “GPL” (for any version of the GNU General Public License), “GPL v2” (for GPL version two only), “GPL and additional rights,” “Dual BSD/GPL,” “Dual MPL/GPL,” and “Proprietary.” Unless your module is explicitly marked as being under a free license recognized by the kernel, it is assumed to be proprietary, and the kernel is “tainted” when the module is loaded.

### More commands

### modinfo :

Display contents of .modinfo section in an LKM object file.

### modprobe –l (-r : to remove the module) :

Insert or remove an LKM or set of LKMs intelligently. For example, if you must load A before loading B, Modprobe will automatically load A when you tell it to load B.

modprobe calculates all of the module dependencies and then load the module along with the dependencies, while insmod does not care about dependencies, insmod only loads the module in question. usually runs when system is booted or when there is new module installed or when we call *depmod -a* from shell.

**depmod :**

this tool’s function is to calculate module dependencies for all modules located in /lib/modules/`uname -r`/ and then keep the dependencies information in /lib/modules/`uname -r`/modules.dep.

**Process to update the module dependencies in-order to use modprobe:**

1. Copy/create a link of your module in /lib/modules/`uname -r`

**Example:**

# ln -s absolute\_path\_of\_module.ko /lib/modules/`uname -r`

1. refresh the modules.dep using the command

#depmod –a

1. Then call the modprobe, without .ko

# modprobe <module-name>

**Debugging by Printing**

The most common debugging technique is monitoring, which in applications programming is done by calling printf at suitable points. When you are debugging kernel code, you can accomplish the same goal with printk.

One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro.

The loglevel macro expands to a string, which is concatenated to the message text at compile time; that’s why there is no comma between the priority and the format string in the following examples. Here are two examples of printk commands, a debug message and a critical message:

printk(KERN\_DEBUG "Here I am: %s:%i\n", \_\_FILE\_\_, \_\_LINE\_\_);

printk(KERN\_CRIT "I'm trashed; giving up on %p\n", ptr);

There are eight possible loglevel strings, defined in the header <linux/kernel.h>; we list them in order of decreasing severity:

|  |  |
| --- | --- |
| **Priority (high to low)** | **Description** |
| KERN\_EMERG <0> | Used for emergency messages, usually those that precede a crash. |
| KERN\_ALERT <1> | A situation requiring immediate action. |
| KERN\_CRIT <2> | Critical conditions, often related to serious hardware or software failures. |
| KERN\_ERR <3> | Used to report error conditions; device drivers often use KERN\_ERR to report hardware difficulties. |
| KERN\_WARNING <4> | Warnings about problematic situations that do not, in themselves, create serious problems with the system. |
| KERN\_NOTICE <5> | Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level. |
| KERN\_INFO <6> | Informational messages. Many drivers print information about the hardware  they find at startup time at this level. |
| KERN\_DEBUG <7> | Used for debugging messages. |

A printkstatement with no specified priority defaults to DEFAULT\_MESSAGE\_LOGLEVEL,

specified in *kernel/printk.c* as an integer. In the 2.6.10 kernel, DEFAULT\_MESSAGE\_LOGLEVEL

is KERN\_WARNING, but that has been known to change in the past.

Based on the loglevel, the kernel may print the message to console. If the priority is less than the

integer variable console\_loglevel, the message is delivered to the console one line at a time (nothing is sent unless a trailing newline is provided). If both klogd and syslogd are running on the system, kernel messages are appended to /var/log/messages (or otherwise treated depending on your syslogd configuration), independent of console\_loglevel. If klogd is not running, the message won’t reach user space unless you read /proc/kmsg (which is often most easily done with the dmesg command). When using klogd, you should remember that it doesn’t save consecutive identical lines; it only saves the first such line and, at a later time, the number of repetitions it

received.

It is possible to read and modify the console loglevel using the text file */proc/sys/*

*kernel/printk*.

The file hosts four integer values:

1. the current loglevel,
2. the default level for messages that lack an explicit loglevel,
3. the minimum allowed loglevel, and the
4. boot-time default loglevel.

Writing a single value to this file changes the current loglevel to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

# echo 8 > /proc/sys/kernel/printk

**How Messages Get Logged**

The *printk* function writes messages into a circular buffer that is \_\_LOG\_BUF\_LEN bytes long: a value from 4 KB to 1 MB chosen while configuring the kernel. The function then wakes any process that is waiting for messages, that is, any process that is sleeping in the syslog system call or that is reading */proc/kmsg*. These two interfaces to the logging engine are almost equivalent, but note that reading from */proc/kmsg* consumes the data from the log buffer, whereas the syslog system call can optionally return log data while leaving it for other processes as well. In general, reading the /proc file is easier and is the default behavior for *klogd*. The dmesg command can be used to look at the content of the buffer without flushing it; actually, the command returns to stdout the whole content of the buffer, whether or not it has already been read. If you happen to read the kernel messages by hand, after stopping *klogd*, you’ll find that the /proc file looks like a FIFO, in that the reader blocks, waiting for more data.

If the circular buffer fills up, printk wraps around and starts adding new data to the

beginning of the buffer, overwriting the oldest data.

**proc Filesystem**

The /proc is a virtual file system. It's sometimes referred to as a process information pseudo-file system.

contains user-accessible objects that pertain to the runtime state of the kernel.

"Pseudo" is used because the proc file system exists only as a reflection of the in-memory kernel data structures it displays. This is why most files and directories within /proc are 0 bytes in size.

Directory listing of /proc reveals two main file groups.

Each numerically named directory within /proc corresponds to the process ID (PID) of a process currently executing on the system. For EX:

dr-xr-xr-x 3 noorg noorg 0 Apr 16 23:24 19636

Here Directory 19636 corresponds to PID 19636, a current bash shell session. These per-process directories contain both subdirectories and regular files that further elaborate on the runtime attributes of a given process.

Check man 5 proc for various process attributes.

The second file group within /proc is the non-numerically named directories and regular files that describe some aspect of kernel operation.

Proc files are either read-only or read-write.

Try the following

cat /proc/modules

cat /proc/interrupts

cat /proc/devices

**sysfs**

sysfs is a virtual file system provided by Linux.

sysfs exports information about devices and drivers from the kernel device model to user space, and is also used for configuration.

sysfs is an in-memory filesystem that was originally based on ramfs.

ramfs was written around the time Linux 2.4.0 was being stabilized.

sysfs was originally called ddfs (Device Driver Filesystem) and was initially created to debug the new driver model as it was being written.

The driverfs was later renamed to sysfs.

End of Chapter I

# 

# CHAPTER 2

## CHAR DRIVERS

### What is a character device?

Character devices do data transfer character by character example: keyboard, mouse, and parallel port.

### What is a character driver?

A character driver drives a character device

A char driver is in charge of implementing the behavior of character device.

### How does linux identify device and their drivers?

Linux identifies devices and their drivers with the help of Device Numbers

* Major number
* Minor number

### Major and Minor Numbers

The **major number identifies the driver associated with the device.** The kernel uses the major number at open time to dispatch execution to the appropriate driver. The minor number is used only by the driver; other parts of the kernel don't use it, and merely pass it along to the driver. It is common for a driver to control several devices; the minor number provides a way for the driver to differentiate among them.

### The Internal Representation of Device Numbers

Within the kernel, the dev\_t type (defined in <linux/types.h>) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev\_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <linux/kdev\_t.h>. To obtain the major or minor parts of a dev\_t, use:

MAJOR(dev\_t dev);

MINOR(dev\_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev\_t,

use:

MKDEV(int major, int minor);

### How to create a device file?

The command to create a device node on a filesystem is mknod; superuser privileges are required for this operation. The command takes three arguments in addition to the name of the file being created. For example, the command

mknod mydev c 254 0

Creates a char device (c) whose major number is 254 and whose minor number is 0. Minor numbers should be in the range 0 to 255 because, for historical reasons, they are sometimes stored in a single byte. There are sound reasons to extend the range of available minor numbers, but for the time being, the eight-bit limit is still in force.

### Two ways of Allocation of a major no. for a driver:

1. Statically
2. Dynamically

**Static Allocation**: Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel

source tree. So, as a driver writer, you have a choice: you can simply pick a number that appears to

be unused. ( may lead to conflicts and hence trouble) to avoid this use dynamic allocation

**Dynamic allocation**: use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. Using *alloc\_chrdev\_region.*

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is *register\_chrdev\_region*, which is declared in *<linux/fs.h>*:

**Static request for device numbers**

int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name);

Arguments to register\_chrdev\_region and return value:

* first is the beginning device number of the range you would like to allocate
* count is the total number of contiguous device numbers you are requesting. name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.
* return value from register\_chrdev\_regionwill be 0 if the allocation was successfully erformed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

**Dynamicly request for numbers**

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name);

Arguments to register\_chrdev\_region and return value:

* dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
* firstminor should be the requested first minor number to use; it is usually 0.
* count is the total number of contiguous device numbers you are requesting.
* name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.

### Freeing the device

When a module is unloaded from the system, the major number must be released. This is accomplished with the following function, which you call from the module's cleanup function:

void unregister\_chrdev\_region(dev\_t first, unsigned int count);

Arguments to unregister\_chrdev\_region and return value:

* dev is the device no.
* count is the count of the minor numbers.

Example:

#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/types.h>

#include <linux/kdev\_t.h>

#include <linux/fs.h>

static dev\_t first; // Global variable for the first device number

static int \_\_init myinit (void) {

    printk(KERN\_INFO "sample driver registered");

    if (alloc\_chrdev\_region(&first, 0, 3, "sample") < 0)

    {

        return -1;

    }

    printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n",MAJOR(first), MINOR(first));

    return 0;

}

static void \_\_exit myexit(void)

{

    unregister\_chrdev\_region(first, 3);

    printk(KERN\_INFO "sample driver unregistered\n");

}

module\_init(myinit);

module\_exit(myexit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Type your name");

MODULE\_DESCRIPTION("Our First sample character Driver");

### Important kernel data structures

Some of the datastructures required to implement fundamental driver operations are file\_operations, file,and inode.

### File Operations *<linux/fs.h>*:

struct file\_operatios contains pointers to functions performed on our device through our driver. Open and check the fs.h it is a vary vast structure, discussing few important function pointers

**int (\*open) (struct inode \*, struct file \*);**

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method.

If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

**int (\*release) (struct inode \*, struct file \*);**

This operation is invoked when the file structure is being released. Like *open*, *release* can be missing.

[Note that *release* isn't invoked every time a process calls *close*. Whenever a file structure is shared (for example, after a *fork* or a *dup*), *release* won't be invoked until all copies are closed. If you need to flush pending data when any copy is closed, you should implement the *flush* method.]

**ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);**

Used to retrieve data from the device. A null pointer in this position causes the *read* system call to fail with -EINVAL ("Invalid argument").

A non-negative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

**ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);**Sends data to the device. If missing, -EINVAL is returned to the program calling the *write* system call. The return value, if non-negative, represents the number of bytes successfully written.

**loff\_t (\*llseek) (struct file \*, loff\_t, int)**

The *llseek* method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The loff\_t is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If the function is not specified for the driver, a seek relative to end-of-file fails, while other seeks succeed by modifying the position counter in the file structure

**int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);**

The *ioctl* system call offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few *ioctl* commands are recognized by the kernel without referring to the fops table. If the device doesn't offer an *ioctl* entry point, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device"). If the device method returns a non-negative value, the same value is passed back to the calling program to indicate successful completion.

### The file Structure <linux/fs.h>:

Note that a file has nothing to do with the FILEs of user-space programs. A FILE is defined in the C library and never appears in kernel code. A struct file, on the other hand, is a kernel structure that never appears in user programs.

The file structure represents an open file. (It is not specific to device drivers; every open file in the system has an associated struct file in kernel space.) It is created by the kernel on open and is passed to any function that operates on the file, until the last close.

After all instances of the file are closed, the kernel releases the data structure. An open file is different from a disk file, represented by struct inode.

In the kernel sources, a pointer to struct file is usually called either file or filp ("file pointer”) . We'll consistently call the pointer filp to prevent ambiguities with the structure itself. Thus, file refers to the structure and filp to a pointer to the structure.

The most important fields of struct file are shown here. As in the previous section, the list can be skipped on a first reading. In the next section though, when we face some real C code, we'll discuss some of the fields, so they are here for you to refer to.

**mode\_t f\_mode;**

The file mode identifies the file as either readable or writable (or both), by means of the bits FMODE\_READ and FMODE\_WRITE. You might want to check this field for read/ write permission in your ioctl function, but you don't need to check permissions for read and write because the kernel checks before invoking your method. An attempt to write without permission, for example, is rejected without the driver even knowing about it.

**loff\_t f\_pos;**

The current reading or writing position. L off\_t is a 64-bit value (long long in gcc terminology). The driver can read this value if it needs to know the current position in the file, but should never change it (read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f\_pos directly).

**unsigned int f\_flags;**

These are the file flags, such as O\_RDONLY, O\_NONBLOCK, and O\_SYNC. A driver needs to check the flag for nonblocking operation, while the other flags are seldom used. In particular, read/write permission should be checked using f\_mode instead of f\_flags. All the flags are defined in the header <linux/fcntl.h>.

**struct file\_operations \*f\_op:**

The operations associated with the file. The kernel assigns the pointer as part of its implementation of open, and then reads it when it needs to dispatch any operations. The value in filp->f\_op is never saved for later reference; this means that you can change the file operations associated with your file whenever you want, and the new methods will be effective immediately after you return to the caller.

For example, the code for open associated with major number 1 (/dev/null, /dev/zero, and so on) substitutes the operations in filp->f\_op depending on the minor number being opened.

This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of "method overriding" in object-oriented programming.

**void \*private\_data;**

The open system call sets this pointer to NULL before calling the openmethod for the driver. The driver is free to make its own use of the field or to ignore it.

The driver can use the field to point to allocated data, but then must free memory in the release method before the file structure is destroyed by the kernel. Private\_data is a useful resource for preserving state information across system calls and is used by most of our sample modules.

**struct dentry \*f\_dentry;**

The directory entry (dentry) structure associated with the file. Dentries are an optimization introduced in the 2.1 development series. Device driver writers normally need not concern themselves with dentry structures, other than to access the inode structure as filp->f\_dentry->d\_inode.

The real structure has a few more fields, but they aren't useful to device drivers. We can safely ignore those fields because drivers never fill file structures; they only access structures created elsewhere.

### Char Device Registration

The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device’s operations, you must allocate and register one or more of these structures. To do so, your code should include <linux/cdev.h>, where the structure and its associated helper functions are defined.

There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

struct cdev \*my\_cdev = cdev\_alloc( );

my\_cdev->ops = &my\_fops;

however, that you will want to embed the cdev structure within a device-specific structure of your own; In that case, you should initialize the structure that you have already allocated with:

void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops);

Either way, there is one other struct cdev field that you need to initialize. Like the file\_operations structure, struct cdev has an owner field that should be set to THIS\_MODULE.

Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

Here, dev is the cdev structure, num is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.

There are a couple of important things to keep in mind when using cdev\_add. The first is that this call can fail. If it returns a negative error code, your device has not been added to the system. It almost always succeeds, however, and that brings up the other point: as soon as cdev\_add returns, your device is “live” and its operations can be called by the kernel. You should not call cdev\_add until your driver is completely ready to handle operations on the device.

To remove a char device from the system, call:

void cdev\_del(struct cdev \*dev);

you should not access the cdev structure after passing it to cdev\_del.

### Example with cdev:

struct scull\_dev {

data \*data; /\* Pointer to first quantum set \*/

int current,max\_size

struct cdev cdev; /\* Char device structure \*/

};

The struct cdev interfaces our device to the kernel. Thisistructure must be initialized and added to the system as described above;

static void my\_init( )

{

int err, devno = MKDEV(scull\_major, scull\_minor);

cdev\_init(&dev->cdev, &scull\_fops);

dev->cdev.owner = THIS\_MODULE;

err = cdev\_add (&dev->cdev, devno, 1);

/\* Fail gracefully if need be \*/

if (err)

printk(KERN\_NOTICE "Error %d adding scull%d", err, index);

}

Since the cdev structure is embedded within struct scull\_dev, cdev\_init must be called to perform the initialization of that structure.

### container\_of macro

container\_of(pointer, type, field);

A convenience macro that may be used to obtain a pointer to a structure from a

pointer to some other structure contained within it.

### The open method

In most drivers, open should perform the following tasks:

* Check for device-specific errors (such as device-not-ready or similar hardware problems)
* Initialize the device, if it is being opened for the first time
* Identify the minor number and update the f\_op pointer, if necessary
* Allocate and fill any data structure to be put in filp->private\_data

In char, most of the preceding tasks depend on the minor number of the device being opened. Therefore, the first thing to do is identify which device is involved. We can do that by looking at inode->i\_rdev.

We've already talked about how the kernel doesn't use the minor number of the device, so the driver is free to use it at will. In practice, different minor numbers are used to access different devices or to open the same device in a different way. For example, */dev/st0* (minor number 0) and /dev/st1 (minor 1) refer to different SCSI tape drives, whereas /dev/nst0 (minor 128) is the same physical device as */dev/st0*, but it acts differently (it doesn't rewind the tape when it is closed). All of the tape device files have different minor numbers, so that the driver can tell them apart.

A driver never actually knows the name of the device being opened, just the device number -- and users can play on this indifference to names by aliasing new names to a single device for their own convenience. If you create two special files with the same major/minor pair, the devices are one and the same, and there is no way to differentiate between them. The same effect can be obtained using a symbolic or hard link, and the preferred way to implement aliasing is creating a symbolic link.

The char driver uses the minor number like this: the most significant nibble (upper four bits) identifies the type (personality) of the device, and the least significant nibble (lower four bits) lets you distinguish between individual devices if the type supports more than one device instance. Thus, char0 is different from charpipe0 in the top nibble, while char0 and char1 differ in the bottom nibble.[ [Bit splitting is a typical way to use minor numbers.] The IDE driver, for example, uses the top two bits for the disk number, and the bottom six bits for the partition number. This is similar to the way opening a regular file for writing truncates it to zero length. The operation does nothing if the device is opened for reading.

### The release Method

The role of the release method is the reverse of open. Sometimes you'll find that the method implementation is called device\_close instead of device\_release. Either way, the device method should perform the following tasks:

* Deallocate anything that open allocated in filp->private\_data return 0;
* Shut down the device on last close

The basic form of char has no hardware to shut down, so the code required is minimal:

### read and write

The read *and* write methods perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar and it's worth introducing them at the same time:

ssize\_t read (struct file \*filp, char \*buff, size\_t count, loff\_t \*offp);

ssize\_t write (struct file \*filp, const char \*buff, size\_t count, loff\_t \*offp);

For both methods, filp is the file pointer and count is the size of the requested data transfer. The buff argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed.

Finally, offp is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type;" its use is discussed later.

As far as data transfer is concerned, the main issue associated with the two device methods is the need to transfer data between the kernel address space and the user address space.

The operation cannot be carried out through pointers in the usual way, or through memcpy. User-space addresses cannot be used directly in kernel space, for a number of reasons.

One big difference between kernel-space addresses and user-space addresses is that memory in user-space can be swapped out. When the kernel accesses a user-space pointer, the associated page may not be present in memory, and a page fault is generated.

Cross-space copies are performed in Linux by special functions, defined in <asm/uaccess.h>. Such a copy is either performed by a generic (memcpy-like) function or by functions optimized for a specific data size (char, short, int, long);

The code for read and write in char needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of every read and write implementation:

unsigned long copy\_to\_user (void \*to, const void \*from, unsigned long count);

unsigned long copy\_from\_user (void \*to, const void \*from, unsigned long count);

Although these functions behave like normal memcpy functions, a little extra care must be used when accessing user space from kernel code.

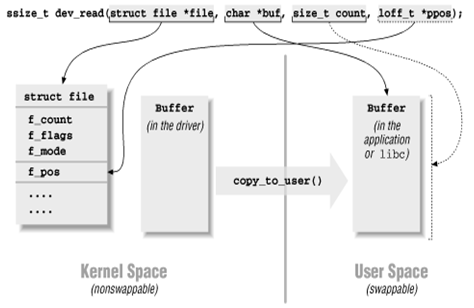
The role of the two functions is not limited to copying data to and from user-space they also check whether the user space pointer is valid. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied.

In both cases, the return value is the amount of memory still to be copied. The char code looks for this error return, and returns -EFAULT to the user if it's not 0.

### The read Method

The return value for read is interpreted by the calling application program as follows:

* If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case.
* If the value is positive, but smaller than count, only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program will retry the read. For instance, if you read using the fread function, the library function reissues the system call till completion of the requested data transfer.
* If the value is 0, end-of-file was reached.
* A negative value means there was an error. The value specifies what the error was, according to <linux/errno.h>. These errors look like -EINTR (interrupted system call) or -EFAULT (bad address).



**fig :3-1**

ssize\_t dev\_read( struct file \*filp, char \*ubuf, size\_t size, loff\_t \*off) {

printk(" in pcd\_read\n");

if(copy\_to\_user( ubuf, kbuf, size)) /\*to copy from kernel space to user space\*/

{

printk(" read Error\n");

return -EFAULT;

}

//\*off=\*off+size;

printk(" %s ->kbuf %s -> ubuf\n", kbuf, ubuf);

return size;

}

Read return value:

The return value for *read* is interpreted by the calling application program as follows:

1. If the value equals the count argument passed to the *read* system call, the requested number of bytes has been transferred. This is the optimal case.
2. If the value is positive, but smaller than count, only part of the data has been transferred..
3. If the value is 0, end-of-file was reached.
4. A negative value means there was an error. The value specifies what the error was, according to <linux/errno.h>. These errors look like -EINTR (interrupted system call) or -EFAULT (bad address).

### The write Method

write, like read, can transfer less data than was requested, according to the following rules for the return value:

* If the value equals count, the requested number of bytes has been transferred.
* If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
* If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to write. We'll examine the exact meaning of this case in "Blocking I/O" in Chapter 5, "Enhanced Char Driver Operations", where blocking write is introduced.
* A negative value means an error occurred; like for read, valid error values are those defined in <linux/errno.h>.

The char code for write deals with a single quantum at a time, like the read method does:

ssize\_t dev\_write(struct file \*filp, const char \*ubuf, size\_t size, loff\_t \*off)

{

printk(" in pcd\_write\n");

if(copy\_from\_user( kbuf, ubuf, size)) /\* to copy from user space to kernel space\*/

{

printk(" Write Err\n");

return -EFAULT;

}

printk(" %s -> kbuf %s -> ubuf\n", kbuf, ubuf);

return size;

}

Write return value: *write*, like *read*, can transfer less data than was requested, according to the following rules for the return value:

1. If the value equals count, the requested number of bytes has been transferred.
2. If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
3. If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to *write*.
4. A negative value means an error occurred; like for *read*, valid error values are those defined in <linux/errno.h>.

### The Current Process:

Most actions performed by the kernel are done on behalf of a specific process. Kernel code can refer to the current process by accessing the global item current, defined in <asm/current.h>, which yields a pointer to struct task\_struct, defined by <linux/sched.h>.

The current pointer refers to the process that is currently executing. During the execution of a system call, such as open or read, the current process is the one that invoked the call. Kernel code can use process-specific information by using current, if it needs to do so.

**For example**

The following statement prints the process ID and the command name of the current process by accessing certain fields in struct task\_struct:

printk(KERN\_INFO "The process is \"%s\" (pid %i)\n",

current->comm, current->pid);

The command name stored in current->comm is the base name of the program file that is being executed by the current process.

End of Chapter 2

# CHAPTER 3

### Advanced Char Driver Operations

### Seeking a Device, The llseek Implementation

The llseek method implements the lseek and llseek system calls. We have already stated that if the llseekmethod is missing from the device's operations, the default implementation in the kernel performs seeks from the beginning of the file and from the current position by modifying filp->f\_pos, the current reading/writing position within the file. Please note that for the lseek system call to work correctly, the read and write methods must cooperate by updating the offset item they receive as argument (the argument is usually a pointer to filp->f\_pos). You may need to provide your own llseek method if the seek operation corresponds to a physical operation on the device or if seeking from end-of-file, which is not implemented by the default method, makes sense. A simple example can be seen in the char driver:

loff\_t char\_llseek(struct file \*filp, loff\_t off, int whence)

{

char\_dev \*dev = filp->private\_data;

loff\_t newpos;

switch(whence) {

case 0: /\* SEEK\_SET \*/

newpos = off;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + off;

break;

case 2: /\* SEEK\_END \*/

newpos = dev->size + off;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if (newpos<0) return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

### The IOCTL interface

###### The ioctl interface typically performs all device-related operations other than read or writes. A device driver’s ioctl interface is called as a result of an ioctl system call. **ioctl** enables **the user to set the terminal baud rate, rewind tape on a tape drive, perform network operations such as setting network address and changing various settings.**

The ioctl function call in user space corresponds to the following prototype

**int ioctl (int fd, int cmd, char \*argp);**

The ioctl driver method, on the other hand, receives its arguments according to this declaration:

int (\*ioctl) (struct inode \*inode, struct file \*filp,

unsigned int cmd, unsigned long arg);

The inode and filp pointers are the values corresponding to the file descriptor fd passed on by the application and are the same parameters passed to the open method. The cmd argument is passed from the user unchanged, and the optional arg argument is passed in the form of an unsigned long, regardless of whether it was given by the user as an integer or a pointer. If the invoking program doesn't pass a third argument, the arg value received by the driver operation has no meaningful value.

Every device has its own ioctl commands with four bit fields

namely type, number, direction and size.

• The type field specifies the magic number, for our device we choose ‘k’ as a magic number. The full list of magic number can be seen in *Documentation/ioctlnumber.* *txt* file.

• The number field identifies the sequential number to distinguish ioctls from each other.

• The direction field denotes the direction of data transfer and the direction can be recognized by the following macros:

* + **IO** an ioctl with no parameters
  + **\_IOW** with write parameters (like **copy\_from\_user**), here the kernel read data from user space
  + **\_IOR** with read parameters (like **copy\_to\_user**), here the kernel would actually write data to user space
  + \_**IOWR** with both read and write parameters.
  + The macros are defined in ***<linux/ioctl.h>*** file. **ioctl**

Though the ioctl system call is most often used to act on devices, a few commands are recognized by the kernel. Note that these commands, when applied to your device, are decoded before your own file operations are called. Thus, if you choose the same number for one of your ioctl commands, you won't ever see any request for that command, and the application will get something unexpected because of the conflict between the ioctl numbers.

The predefined commands are divided into three groups:

* Those that can be issued on any file (regular, device, FIFO, or socket)
* Those that are issued only on regular files
* Those specific to the filesystem type

int access\_ok(int type, const void \*addr, unsigned long size);

The first argument should be either VERIFY\_READ or VERIFY\_WRITE, depending on whether the action to be performed is reading the user-space memory area or writing it. The addr argument holds a user-space address, and size is a byte count. If ioctl, for instance, needs to read an integer value from user space, size is sizeof(int). If you need to both read and write at the given address, use VERIFY\_WRITE, since it is a superset of VERIFY\_READ.

put\_user(datum, ptr)

\_\_put\_user(datum, ptr)

These macros write the datum to user space; they are relatively fast, and should be called instead of copy\_to\_userwhenever single values are being transferred. The size of the data transfer depends on the type of the ptr argument and is determined at compile time using a special gcc pseudo-function that isn't worth showing here.

get\_user(local, ptr)

\_\_get\_user(local, ptr)

These macros are used to retrieve a single datum from user space. They behave like put\_user and \_\_put\_user, but transfer data in the opposite direction. The value retrieved is stored in the local variable local; the return value indicates whether the operation succeeded or not.

### The Implementation of the ioctl Commands

The ioctl implementations consist of a switch statement that selects the correct behavior according to the cmd argument.Different commands have different numeric values, which are

usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition. Custom drivers usually declare such symbols in their own header files. User programs must, of course, include that header file as well to have access to those symbols.

The char implementation of ioctl only transfers the configurable parameters of the device and turns out to be as easy as the following:

int myioctl (struct inode \*ind, struct file \*filp, unsigned int cmd, unsigned long arg) {

int ret = 0;

static unsigned long i = 2000;

static int err;

if( \_IOC\_TYPE( cmd ) != MAGIC ) //check whether doing ioctl with correct device

{

printk(" wrong type :cmd \n");

return -ENODEV;

}

if ( \_IOC\_DIR(cmd) & \_IOC\_READ ) {

err = !access\_ok(VERIFY\_WRITE, (void \*)arg, \_IOC\_SIZE (cmd));

} /\* verify process has write access to the memory pointed by arg\*/

else if ( \_IOC\_DIR(cmd) & \_IOC\_WRITE ) {

err = !access\_ok(VERIFY\_READ, (void \*)arg, \_IOC\_SIZE (cmd)) ;

} /\* verify process has Read access to the memory pointed by arg\*/

if (err) {

printk("NO Read permission, access failed\n");

return -EFAULT;

}

switch ( cmd ) {

case IOREAD: /\* case for reading data from kernel space\*/

ret = \_\_put\_user ( i , ( unsigned long \* ) arg ); /\*put value of I into user space memory pointed by arg \*/

printk(" data returned in READ = %d", i );

break;

case IOWRITE: /\*case for write to kernel space\*/

ret = \_\_get\_user ( i, ( unsigned long \* ) arg ); /\*get value of I from user space memory pointed by arg \*/

printk(" value got from user = %ld\n", i);

break;

default:

printk (" no such cmd \n");

return -ENOTTY;

}

return ret;

}

### Blocking I/O

One problem that might arise with read is what to do when there's no data yet, but we're not at end-of-file.

The default answer is "go to sleep waiting for data.'' This section shows how a process is put to sleep, how it is awakened, and how an application can ask if there is data without just blindly issuing a read call and blocking. We then apply the same concepts to write.

### Going to Sleep and Awakening

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. At some future time, when the event being waited for occurs, the process will be woken up and will continue with its job.

wake\_up(wait\_queue\_head\_t \*queue);

This function will wake up all processes that are waiting on this event queue.

wake\_up\_interruptible(wait\_queue\_head\_t \*queue);

*wake\_up\_interruptible* wakes up only the processes that are in interruptible sleeps. Any process that sleeps on the wait queue using a noninterruptible function or macro will continue to sleep.

wake\_up\_sync(wait\_queue\_head\_t \*queue);

wake\_up\_interruptible\_sync(wait\_queue\_head\_t \*queue);

Normally, a *wake\_up* call can cause an immediate reschedule to happen, meaning that other processes might run before *wake\_up* returns. The "synchronous" variants instead make any awakened processes runnable, but do not reschedule the CPU. This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway. Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion.

If your driver is using *interruptible\_sleep\_on*, there is little difference between *wake\_up* and *wake\_up\_interruptible*.

**DECLARE\_WAIT\_QUEUE\_HEAD(wq);**

ssize\_t sleepy\_read (struct file \*filp, char \*buf, size\_t count,

loff\_t \*pos)

{

printk(KERN\_DEBUG "process %i (%s) going to sleep\n",

current->pid, current->comm);

interruptible\_sleep\_on(&wq);

printk(KERN\_DEBUG "awoken %i (%s)\n", current->pid, current->comm);

return 0; /\* EOF \*/

}

ssize\_t sleepy\_write (struct file \*filp, const char \*buf, size\_t count,

loff\_t \*pos)

{

printk(KERN\_DEBUG "process %i (%s) awakening the readers...\n",

current->pid, current->comm);

wake\_up\_interruptible(&wq);

return count; /\* succeed, to avoid retrial \*/

}

### Blocking and Nonblocking Operations

The flag gets its name from "open-nonblock,'' because it can be specified at open time (and originally could only be specified there). If you browse the source code, you'll find some references to an O\_NDELAY flag; this is an alternate name for O\_NONBLOCK, accepted for compatibility with System V code. The flag is cleared by default, because the normal behavior of a process waiting for data is just to sleep. In the case of a blocking operation, which is the default, the following behavior should be implemented in order to adhere to the standard semantics:

* If a process calls read but no data is (yet) available, the process must block. The process is awakened as soon as some data arrives, and that data is returned to the caller, even if there is less than the amount requested in the count argument to the method.
* If a process calls write and there is no space in the buffer, the process must block, and it must be on a different wait queue from the one used for reading. When some data has been written to the hardware device, and space becomes free in the output buffer, the process is awakened and the write call succeeds, although the data may be only partially written if there isn't room in the buffer for the count bytes that were requested.
* The behavior of read and write is different if O\_NONBLOCK is specified. In this case, the calls simply return -EAGAIN if a process calls read when no data is available or if it calls write when there's no space in the buffer. Only the read, write, and open file operations are affected by the nonblocking flag.

End of Chapter 3

# CHAPTER 4

## Linux Kernel Build & static LINKING

**Compile Linux Kernel from Source to Build Custom Kernel**

**1. Download the Latest Stable Kernel**

The first step is to download the latest stable kernel from kernel.org.

# cd /usr/src/ (or to the directory where you have down loaded the kernel )

# wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.3.tar.xz

**2. Untar the Kernel Source**

The second step is to untar the kernel source file for compilation.

# tar -xvJf linux-3.9.3.tar.xz

**3. Configure the Kernel**

The kernel contains nearly 3000 configuration options. To make the kernel used by most people on most hardware, the Linux distro like Ubuntu, Fedora, Debian, RedHat, CentOS, etc, will generally include support for most common hardware. You can take any one of configuration from the distro, and on top of that you can add your own configuration, or you can configure the kernel from scratch, or you can use the default config provided by the kernel.

# cd linux-3.9.3

**How to add a static module to the kernel**

**To build your driver statically into the kernel image:**

* First, you select a directory in drivers directory where you want to put your driver files.
* Assume you want to put your files in drivers/char/. Copy your files into this directory.
* There will be a Kconfig file in the drivers/char/ directory, open it and add an entry like this in the before the endmenu.

config MYDRIVER

bool "This is a driver for something"

default n

help

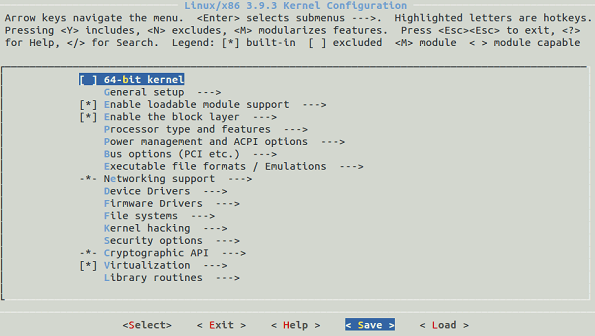
This is a test driver.

* Save the file and open Makefile in the same directory. Goto end of the file and add the following entry.

obj-$(CONFIG\_MYDRIVER) += mydriver.o

* That's it you have added the file to the kernel tree. Now, as usual, executer  make menuconfig ( as shown below) and select MYDRIVER. Compile the kernel ( as shown below).

# make menuconfig



The make menuconfig, will launch a text-based user interface with default configuration options as shown in the figure. You should have installed “libncurses and libncurses-devel” packages for this command to work.(**sudo apt-get install libncurses5-dev).**

After selecting the configuration press esc till you come to the home screen. You will be prompted with an option to save your configrations. Press yes. This creates a .config file in the current directory. You and view this file using the editor. You should not manually edit this file.

**4. Compile the Linux Kernel**

Compile the main kernel:

# make

Compile the kernel modules:

# make modules

Install the kernel modules:

# make modules\_install

At this point, you should see a directory named /lib/modules/your kernel version/ in your system.

**5. Install the New Kernel**

Install the new kernel on the system:

# make install

The make install command will create the following files in the **/boot** directory.

* vmlinuz-kernel\_version – The actual kernel
* System.map- kernel\_version – The symbols exported by the kernel
* initrd.img-kernel\_version– initrd image is temporary root file system used during boot process
* config- kernel\_version – The kernel configuration file

The command “make install” will also update the grub.cfg by default. So we don’t need to manually edit the grub.cfg file.

**6. Boot Linux to the new Kernel**

To use the new kernel that you just compiled, reboot the system.

# reboot

Since, in grub.cfg, the new kernel is added as default boot, the system will boot from the new kernel. Just in case if you have problems with the new kernel, you can select the old kernel from the grub menu during boot and you can use your system as usual.

Once the system is up, use uname command to verify that the new version of Linux kernel is installed.

$ uname -r

# CHAPTER 5

## Flow of Time

Real-world drivers have to deal with issues such as timing, memory management, hardware access etc. We have some kernel resources that are available for drivers for time management.

**Dealing with time involves the following, in order of increasing complexity:**

* Understanding kernel timing
* Knowing the current time
* Delaying operation for a specified amount of time
* Scheduling asynchronous functions to happen after a specified time lapse

### Time intervals in the kernel:

### Jiffies

Jiffies are kernel variables which get initialized to 0 when the system boots, they are basically used to calculate long delays

Every time a timer interrupt occurs, the value of the variable jiffies is incremented. Timer interrupts are generated by the system's timing hardware at regular intervals; this interval is set by thekernel according to the value of HZ, which is an architecture-dependent value defined in **<linux/param.h>** Timer interrupt is the mechanism the kernel uses to keep track of time intervals.and is thus the number of clock ticks since the computer was turned on.

**Knowing the current time:**

Kernel code can always retrieve the current time by looking at the value of jiffies. The act that the value represents only the time since the last boot is not relevant to the river, because its life is limited to the system uptime. Drivers can use the current value of iffies to calculate time intervals across events. Looking at jiffies is almost always sufficient when you need to measure time intervals

### Delaying Execution:

### Long Delays

If you want to delay execution by a multiple of the clock tick or you don't require strict precision (for example, if you want to delay an integer number of seconds), the easiest implementation is the following, also known as busy waiting

unsigned long j = jiffies + jit\_delay \* HZ;

while (jiffies < j)

/\* nothing \*/;

A better solution that allows other processes to run during the time interval is the following, although it can't be used in hard real-time tasks or other time-critical situations.

while (jiffies < j)

schedule();

The variable j in this example and the following ones is the value of jiffies at the expiration of the delay and is always calculated as just shown for busy waitin**int**

time\_before(unsigned long a, unsigned long b)

int time\_after\_eq(unsigned long a, unsigned long b);

int time\_before\_eq(unsigned long a, unsigned long b);

### The first evaluates true when *a*, as a snapshot of jiffies, represents a time after *b*, the second evaluates true when time *a* is before time *b*, and the last two compare for “after or equal” and “before or equal.” The code works by converting the values to signed long, subtracting them, and comparing the result. If you need to know the difference between two instances of jiffies in a safe way, you can use the same trick:

### diff = (long)t2 - (long)t1;.

### You can convert a jiffies difference to milliseconds trivially through:

### msec = diff \* 1000 / HZ;

### Sometimes, however, you need to exchange time representations with user space programs that tend to represent time values with struct timeval and struct timespec. The two structures represent a precise time quantity with two numbers: seconds and microseconds are used in the older and popular struct timeval, and secondsand nanoseconds are used in the newer struct timespec. The kernel export ts four helper functions to convert time values expressed as jiffies to and from those structures:

#include <linux/time.h>

unsigned long timespec\_to\_jiffies(struct timespec \*value);

void jiffies\_to\_timespec(unsigned long jiffies, struct timespec \*value);

unsigned long timeval\_to\_jiffies(struct timeval \*value);

void jiffies\_to\_timeval(unsigned long jiffies, struct timeval \*value);

### Accessing the 64-bit jiffy count is not as straightforward as accessing jiffies. Whileon 64-bit computer architectures the two variables are actually one, access to the value is not atomic for 32-bit processors. This means you might read the wrong value if both halves of the variable get updated while you are reading them. It’s extremely unlikely you’ll ever need to read the 64-bit counter, but in case you do, you’ll be glad to know that the kernel exports a specific helper function that does the proper locking for you:

### #include <linux/jiffies.h>

### u64 get\_jiffies\_64(void);

### In the above prototype, the u64 type is used. This is one of the types defined by*<linux/types.h>* , and represents an unsigned 64-bit type.

### Short Delays

Sometimes a real driver needs to calculate very short delays in order to synchronize with the hardware. In this case, using the jiffies value is definitely not the solution.

The kernel functions udelay and mdelay serve this purpose.[ The u in udelay represents the Greek letter mu and stands for micro.] Their prototypes are

#include <linux/delay.h>

void udelay(unsigned long usecs);

void mdelay(unsigned long msecs);

The functions are compiled inline on most supported architectures. The former uses a software loop to delay execution for the required number of microseconds, and the latter is a loop around udelay, provided for the convenience of the programmer. The udelay function is where the BogoMips value is used: its loop is based on the integer value loops\_per\_second, which in turn is the result of the BogoMips calculation performed at boot time. The udelay call should be called only for short time lapses because the precision of loops\_per\_second is only eight bits, and noticeable errors accumulate when calculating long delays.

Even though the maximum allowable delay is nearly one second (since calculations overflow for longer delays), the suggested maximum value for udelay is 1000 microseconds (one millisecond). The function mdelay helps in cases where the delay must be longer than one millisecond.

It's also important to remember that udelay is a busy-waiting function (and thus mdelay is too); other tasks can't be run during the time lapse. You must therefore be very careful, especially with mdelay, and avoid using it unless there's no other way to meet your goal.

Currently, support for delays longer than a few microseconds and shorter than a timer tick is very inefficient. This is not usually an issue, because delays need to be just long enough to be noticed by humans or by the hardware.

One hundredth of a second is a suitable precision for human-related time intervals, while one millisecond is a long enough delay for hardware activities.

### Current time:

Kernel keeps the current time by reading a clock device .Maintaining a kernel variable with the current time. Current time is accessible to user mode programs via system calls.

*gettimeofday( )* is the usual interface to the current time maintained by the system. It’s quite unlikely that a driver will ever need to know the wall-clock time, expressed in months, days, and hours; the information is usually needed only by user programs such as *cron* and *syslogd*. Dealing with real-world time is usually best left to user space, where the C library offers better support; besides, such code is often too Policy-related to belong in the kernel. There *is* a kernel function that turns a wallclock time into a jiffies value, however:

#include <linux/time.h>

unsigned long mktime (unsigned int year, unsigned int mon,

unsigned int day, unsigned int hour,

unsigned int min, unsigned int sec);

### Processor-Specific Registers

If you need to measure very short time intervals or you need extremely high precision in your figures, you can resort to platform-dependent resources, a choice of precision over portability.

In modern processors, the pressing demand for empirical performance figures is thwarted by the intrinsic unpredictability of instruction timing in most CPU designs due to cache memories, instruction scheduling, and branch prediction. As a response, CPU manufacturers introduced a way to count clock cycles as an easy and reliable way to measure time lapses.

Therefore, most modern processors include a counter register that is steadily incremented once at each clock cycle. Nowadays, this clock counter is the only reliable way to carry out high-resolution timekeeping tasks.

The details differ from platform to platform: the register may or may not be readable from user space, it may or may not be writable, and it may be 64 or 32 bits wide. In the last case, you must be prepared to handle overflows just like we did with the jiffy counter. The register may even not exist for your platform, or it can be implemented in an external device by the hardware designer, if the CPU lacks the feature and you are dealing with a special-purpose computer.

Whether or not the register can be zeroed, we strongly discourage resetting it, even when hardware permits. You might not, after all, be the only user of the counter at any given time; on some platforms supporting SMP, for example, the kernel depends on such a counter to be synchronized across processors.

Since you can always measure differences between values, as long as that difference doesn’t exceed the overflow time, you can get the work done without claiming exclusive ownership of the register by modifying its current value. The most renowned counter register is the TSC (timestamp counter), introduced in x86 processors with the Pentium and present in all CPU designs ever since—including the x86\_64 platform. It is a 64-bit register that counts CPU clock cycles; it can be read from both kernel space and user space.

After including *<asm/msr.h>* (an x86-specific header whose name stands for “machine-specific registers”), you can use one of these macros:

rdtsc(low32,high32);

rdtscl(low32);

rdtscll(var64);

Some of the other platforms offer similar functionality, and kernel headers offer anarchitecture-independent function that you can use instead of rdtsc. It is called get\_cycles, defined in *<asm/timex.h>* (included by *<linux/timex.h>*). Its prototype is:

#include <linux/timex.h>

cycles\_t get\_cycles(void);

This function is defined for every platform, and it always returns 0 on the platforms that have no cycle-counter register. The cycles\_t type is an appropriate unsigned type to hold the value read. It’s important to remember that the three delay functions are busy-waiting; other tasks can’t be run during the time lapse. Thus, they replicate, though on a different scale, the behavior of *jitbusy*.

Thus, these functions should only be used when there is no practical alternative. There is another way of achieving millisecond (and longer) delays that does not involve busy waiting. The file *<linux/delay.h>* declares these functions:

void msleep(unsigned int millisecs);

unsigned long msleep\_interruptible(unsigned int millisecs);

void ssleep(unsigned int seconds)

The first two functions puts the calling process to sleep for the given number of millisecs. A call to *msleep* is uninterruptible; you can be sure that the process sleeps for at least the given number of milliseconds. If your driver is sitting on a wait queue and you want a wakeup to break the sleep, use msleep\_interruptible. The return value from msleep\_interruptibleis normally 0; if, however, the process is awakened early, the return value is the number of milliseconds remaining in the originally requested sleep period. A call to ssleepputs the process into an uninterruptible sleep for the given number of seconds. In general, if you can tolerate longer delays than requested, you should use

schedule\_timeout, msleep, or ssleep.

### Tasklets

Shortly before the release of the 2.4 kernel, the developers added a new mechanism for the deferral of kernel tasks. This mechanism, called tasklets, is now the preferred way to accomplish bottom-half tasks; indeed, bottom halves themselves are now implemented with tasklets.

Each tasklet has associated with it a function that is called when the tasklet is to be executed. The life of some kernel developer was made easier by giving that function a single argument of type unsigned long, which makes life a little more annoying for those who would rather pass it a pointer; casting the long argument to a pointer type is a safe practice on all supported architectures and pretty common in memory management. The tasklet function is of type void; it returns no value.

Software support for tasklets is part of <linux/interrupt.h>, and the tasklet itself must be declared Once a tasklet is scheduled, it is guaranteed to be run once (if enabled) at a safe time. Tasklets may reschedule themselves in much the same manner as task queues. A tasklet need not worry about running against itself on a multiprocessor system, since the kernel takes steps to ensure that any given tasklet is only running in one place. If your driver implements multiple tasklets, however, it should be prepared for the possibility that more than one of them could run simultaneously. In that case, spinlocks must be used to protect critical sections of the code (semaphores, which can sleep, may not be used in tasklets since they run in interrupt time).

### Workqueues

*Workqueues* are, superficially, similar to tasklets; they allow kernel code to request that a function be called at some future time. There are, h however, some significant differences between the two, including:

• Tasklets run in software interrupt context with the result that all tasklet code must be atomic. Instead, workqueue functions run in the context of a special kernel process; as a result, they have more flexibility. In particular, workqueue functions can sleep.

• Tasklets always run on the processor from which they were originally submitted.

Workqueues work in the same way, by default.

• Kernel code can request that the execution of workqueue functions be delayed

for an explicit interval.

The key difference between the two is that tasklets execute quickly, **for a short period of time, and in atomic mode**, while workqueue functions may have higher latency **but need not be atomic**.

Each mechanism has situations where it is appropriate.

Workqueues have a type of struct workqueue\_struct, which is defined in *<linux/workqueue.h>*. A workqueue must be explicitly created before use, using one of the

following two functions:

struct workqueue\_struct \*create\_workqueue(const char \*name);

struct workqueue\_struct \*create\_singlethread\_workqueue(const char \*name);

**Each workqueue has one or more dedicated processes (“kernel threads”), which run functions submitted to the queue**.

If you use create\_workqueue, you get a workqueuethat has a dedicated thread for each processor on the system. In many cases, all those threads are simply overkill; if a single worker thread will suffice, create theworkqueue with

create\_singlethread\_workqueue instead.

To submit a task to a workqueue, you need to fill in a work\_struct structure. This can be done at compile time as follows:

DECLARE\_WORK(name, void (\*function)(void \*), void \*data);

Where name is the name of the structure to be declared, function is the function that is to be called from the workqueue, and data is a value to pass to that function. If you need to set up the work\_struct structure at runtime, use the following two macros:

INIT\_WORK(struct work\_struct \*work, void (\*function)(void \*), void \*data);

PREPARE\_WORK(struct work\_struct \*work, void (\*function)(void \*), void \*data);

INIT\_WORKdoes a more thorough job of initializing the structure; you should use it the first time that structure is set up. PREPARE\_WORKdoes almost the same job, but it does not initialize the pointers used to link the work\_struct structure into the workqueue. If there is any possibility that the structure may currently be submitted to a workqueue, and you need to change that structure, use PREPARE\_WORKrather than INIT\_WORK.

There are two functions for submitting work to a workqueue:

int queue\_work(struct workqueue\_struct \*queue, struct work\_struct \*work);

int queue\_delayed\_work(struct workqueue\_struct \*queue, struct work\_struct \*work, unsigned long delay);

Either one adds work to the given queue. If queue\_delayed\_workis used, however, the actual work is not performed until at least delay jiffies have passed. The return value from these functions is 0 if the work was successfully added to the queue; a nonzero result means that this work\_struct structure was already waiting in the queue, and was not added a second time.

At some time in the future, the work function will be called with the given data value. The function will be running in the context of the worker thread, so it can sleep if need be—although you should be aware of how that sleep might affect any other tasks submitted to the same workqueue. What the function cannot do, however, is access user space. Since it is running inside a kernel thread, there simply is no user space to access.

Should you need to cancel a pending workqueue entry, you may call:

int cancel\_delayed\_work(struct work\_struct \*work);

The return value is nonzero if the entry was canceled before it began execution. The kernel guarantees that execution of the given entry will not be initiated after a call to cancel\_delayed\_work. If cancel\_delayed\_workreturns 0, however, the entry may have already been running on a different processor, and might still be running after a call to cancel\_delayed\_work.

To be absolutely sure that the work function is not running anywhere in the system after cancel\_delayed\_workreturns 0, you must follow that call with a call to:

void flush\_workqueue(struct workqueue\_struct \*queue);

After *flush\_workqueue* returns, no work function submitted prior to the call is running anywhere in the system.

When you are done with a workqueue, you can get rid of it with:

void destroy\_workqueue(struct workqueue\_struct \*queue);

### Kernel Timers

The ultimate resources for time keeping in the kernel are the timers. Timers are used to schedule execution of a function (a timer handler) at a particular time in the future. They thus work differently from task queues and tasklets in that you can specify when in the future your function will be called, whereas you can't tell exactly when a queued task will be executed. On the other hand, kernel timers are similar to task queues in that a function registered in a kernel timer is executed only once -- timers aren't cyclic.

You register your function once, and the kernel calls it once when the timer expires. Such a functionality is used often within the kernel proper, but it is sometimes needed by the drivers as well, as in the example of the floppy motor.

The kernel timers are organized in a doubly linked list. This means that you can create as many timers as you want. A timer is characterized by its timeout value (in jiffies) and the function to be called when the timer expires. The timer handler receives an argument, which is stored in the data structure, together with a pointer to the handler itself.

The data structure of a timer looks like the following, which is extracted from <linux/timer.h>):

struct timer\_list {

struct timer\_list \*next; /\* never touch this \*/

struct timer\_list \*prev; /\* never touch this \*/

unsigned long expires; /\* the timeout, in jiffies \*/

unsigned long data; /\* argument to the handler \*/

void (\*function)(unsigned long); /\* handler of the timeout \*/

volatile int running; /\* don't touch \*/

};

The timeout of a timer is a value in jiffies. Thus, timer->function will run when jiffies is equal to or greater than timer->expires. The timeout is an absolute value; it is usually generated by taking the current value of jiffies and adding the amount of the desired delay. These are the functions used to act on timers:

void init\_timer(struct timer\_list \*timer);

This inline function is used to initialize the timer structure. Currently, it zeros the prev and next pointers. Programmers are strongly urged to use this function to initialize a timer and to never explicitly touch the pointers in the structure, in order to be forward compatible.

void add\_timer(struct timer\_list \*timer);

This function inserts a timer into the global list of active timers.

int mod\_timer(struct timer\_list \*timer, unsigned long expires);

Should you need to change the time at which a timer expires, mod\_timer can be used. After the call, the new expires value will be used.

int del\_timer(struct timer\_list \*timer);

If a timer needs to be removed from the list before it expires, del\_timer should be called. When a timer expires, on the other hand, it is automatically removed from the list.

End of Chapter 5

# CHAPTER 6

## Hardware Management

Although working with character drivers is a good introduction to the software interface of a Linux device driver, implementing a real device requires hardware. The driver is the abstraction layer between software concepts and hardware circuitry; as such, it needs to talk with both of them. Up to now, we have examined the internals of software concepts; this chapter completes the picture by showing you how a driver can access **I/O ports and I/O memory** while being portable across Linux platforms.

Most PCI devices map registers into a memory address region. This I/O memory approach is generally preferred because it doesn't require use of special-purpose processor instructions; CPU cores access memory much more efficiently, and the compiler has much more freedom in register allocation and addressing-mode selection when accessing memory.

### Using I/O Ports

I/O ports are the means by which drivers communicate with many devices out there -- at least part of the time. This section covers the various functions available for making use of **I/O ports**.

#include <linux/ioport.h>

int check\_region(unsigned long start, unsigned long len);

struct resource \*request\_region(unsigned long start,

unsigned long len, char \*name);

void release\_region(unsigned long start, unsigned long len);

After a driver has requested the range of I/O ports it needs to use in its activities, it must read and/or write to those ports. To this aim, most hardware differentiates between 8-bit, 16-bit, and 32-bit ports.

A C program, therefore, must call different functions to access different size ports. As suggested in the previous section, computer architectures that support only memory-mapped I/O registers fake port I/O by remapping port addresses to memory addresses, and the kernel hides the details from the driver in order to ease portability. The Linux kernel headers (specifically, the architecture-dependent header <asm/io.h>) define the following inline functions to access I/O ports.

**NOTE:** From now on, when we use unsigned without further type specifications, we are referring to an architecture-dependent definition whose exact nature is not relevant. The functions are almost always portable because the compiler automatically casts the values during assignment -- their being unsigned helps prevent compile-time warnings. No information is lost with such casts as long as the programmer assigns sensible values to avoid overflow.

unsigned inb(unsigned port);

void outb(unsigned char byte, unsigned port);

Read or write byte ports (eight bits wide). The port argument is defined as unsigned long for some platforms and unsigned short for others. The return type of inb is also different across architectures.

unsigned inw(unsigned port);

void outw(unsigned short word, unsigned port);

These functions access 16-bit ports (word wide

unsigned inl(unsigned port);

void outl(unsigned longword, unsigned port);

These functions access 32-bit ports. longword is either declared as unsigned long or unsigned int, according to the platform. Note that no 64-bit port I/O operations are defined. Even on 64-bit architectures, the port address space uses a 32-bit (maximum) data path.

### String Operations

In addition to the single-shot in and out operations, some processors implement special instructions to transfer a sequence of bytes, words, or longs to and from a single I/O port or the same size. These are the so-called string instructions, and they perform the task more quickly than a C-language loop can do. The following macros implement the concept of string I/O by either using a single machine instruction or by executing a tight loop if the target processor has no instruction that performs string I/O.

The prototypes for string functions are the following:

void insb(unsigned port, void \*addr, unsigned long count);

void outsb(unsigned port, void \*addr, unsigned long count);

Read or write count bytes starting at the memory address addr. Data is read from or written to the single port port.

void insw(unsigned port, void \*addr, unsigned long count);

void outsw(unsigned port, void \*addr, unsigned long count);

Read or write 16-bit values to a single 16-bit port.

void insl(unsigned port, void \*addr, unsigned long count);

void outsl(unsigned port, void \*addr, unsigned long count);

Read or write 32-bit values to a single 32-bit port.

### Using I/O Memory

Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory. Both are called I/O memory because the difference between registers and memory is transparent to software.

This is similar to how I/O ports are registered and is accomplished by the following functions:

int check\_mem\_region(unsigned long start, unsigned long len);

void request\_mem\_region(unsigned long start, unsigned long len,char \*name);

void release\_mem\_region(unsigned long start, unsigned long len);

The start argument to pass to the functions is the physical address of the memory region, before any remapping takes place. The functions would normally be used in a manner such as the following:

if (check\_mem\_region(mem\_addr, mem\_size)) {

printk("drivername: memory already in use\n");

return -EBUSY;

}

request\_mem\_region(mem\_addr, mem\_size, "drivername");

[...]

release\_mem\_region(mem\_addr, mem\_size);

### Directly Mapped Memory

Several computer platforms reserve part of their memory address space for I/O locations, and automatically disable memory management for any (virtual) address in that memory range

When you need to access a directly mapped I/O memory area, you still shouldn't dereference your I/O pointers, even though, on some architectures, you may well be able to get away with doing exactly that. To write code that will work across systems and kernel versions, however, you must avoid direct accesses and instead use the following functions.

unsigned readb(address);

unsigned readw(address);

unsigned readl(address);

These macros are used to retrieve 8-bit, 16-bit, and 32-bit data values from I/O memory. The advantage of using macros is the typelessness of the argument: address is cast before being used, because the value "is not clearly either an integer or a pointer, and we will accept both'' (from *asm-alpha/io.h*). Neither the reading nor the writing functions check the validity of address, because they are meant to be as fast as pointer dereferencing (we already know that sometimes they actually expand into pointer dereferencing).

void writeb(unsigned value, address);

void writew(unsigned value, address);

void writel(unsigned value, address);

### 

### Pausing I/O

Some platforms—most notably the i386—can have problems when the processor tries to transfer data too quickly to or from the bus. The problems can arise when the processor is overclocked with respect to the peripheral bus (think ISA here) and can show up when the device board is too slow. The solution is to insert a small delay after each I/O instruction if another such instruction follows. On the x86, the pause is achieved by performing an out b instruction to port 0x80 (normally but not always unused), or by busy waiting. See the *io.h* file under your platform’s *asm* subdirectory for details.

If your device misses some data, or if you fear it might miss some, you can use pausing

functions in place of the normal ones. The pausing functions are exactly like those listed previously, but their names end in *\_p*; they are called *inb\_p*, *outb\_p*, and so on. The functions are defined for most supported architectures, although they often expand to the same code as nonpausing I/O, because there is no need for the extra pause if the architecture runs with a reasonably modern peripheral bus Platform.

End of Chapter 5

# CHAPTER 7

## INTERRUPT HANDLING

### Interrupts and Exceptions

An *interrupt* is usually defined as an event that alters the sequence of instructions executed by

a processor. Such events correspond to electrical signals generated by hardware circuits both

inside and outside of the CPU chip.

Interrupts are often divided into *synchronous* and *asynchronous* interrupts:

**Synchronousinterrupts** are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.

***Asynchronous* interrupts** are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

### Intel 80x86 microprocessor manuals designate synchronous and asynchronous interrupts as exceptions and interrupts, respectively.

We'll adopt this classification, although we’ll occasionally use the term "interrupt signal" to designate both types together (synchronous as well as asynchronous).

Interrupts are issued by interval timers and I/O devices; for instance, the arrival of a keystroke from a user sets off an interrupt.

Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a page fault or a request (via an int instruction) for a kernel service. One word of caution before moving on: we cover in this chapter only "classic" interrupts common to all PCs; we do not cover the nonstandard interrupts of some architecture.

### Installing an Interrupt Handler

If you want to actually “see” interrupts being generated, writing to the hardware device isn’t enough; a software handler must be configured in the system. If the Linux kernel hasn’t been told to expect your interrupt, it simply acknowledges and ignores it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in *<linux/interrupt.h>*, implement the interrupt registration interface:

int request\_irq ( unsigned int irq,

irqreturn\_t (\*handler)( int, void \*, struct pt\_regs \*),

unsigned long flags,

const char \*dev\_name, void \*dev\_id);

void free\_irq (unsigned int irq, void \*dev\_id);

The value returned from request\_irqto the requesting function is either 0 to indicate success or a negative error code, as usual. It’s not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line.

**The arguments to the functions are as follows:**

**unsigned int irq :** The interrupt number being requested.

**irqreturn\_t (\*handler) (int, void \*, struct pt\_regs \*) :** The pointer to the handling function being installed.

We discuss the arguments to this function and its return value later in this chapter.

**unsigned long flags :** As you might expect, a bit mask of options related to interrupt management.

**const char \*dev\_name :** The string passed to *request\_irq* is used in /proc/interrupts to show the owner of the interrupt

**void \*dev\_id :** This pointer is used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). When no sharing is in force, dev\_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea

The correct place to call request\_irq is when the device is first opened, before the hardware is instructed to generate interrupts. The place to call free\_irq is the last time the device is closed, after the hardware is told not to interrupt the processor any more.

The bits that can be set in flags are as follows:

**SA\_INTERRUPT**

When set, this indicates a "fast'' interrupt handler. Fast handlers are executed with interrupts disabled.

**SA\_SHIRQ**

This bit signals that the interrupt can be shared between devices.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea.

If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used.

Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

The correct place to call *request\_irq* is when the device is first opened, *before* the hardware is instructed to generate interrupts.

The place to call *free\_irq* is the last time the device is closed, *after* the hardware is told not to interrupt the processor any more.

### Fast Handler:

Fast interrupts were those that could be handled very quickly

fast interrupts (those that were requested with the SA\_INTERRUPT flag) are executed with all other interrupts disabled on the current processor.

Other processors can still handle interrupts, though you will never see two processors handling the same IRQ at the same time

A fast handler runs with interrupt reporting disabled in the microprocessor, and the interrupt being serviced is disabled in the interrupt controller.

### Implementing a Handler

So far, we've learned to register an interrupt handler, but not to write one. Actually, there's nothing unusual about a handler -- it's ordinary C code.

The only peculiarity is that a handler runs at interrupt time and therefore suffers some restrictions on what it can do. These restrictions are the same as those we saw with task queues. A handler can't transfer data to or from user space, because it doesn't execute in the context of a process. Handlers also cannot do anything that would sleep, such as calling sleep\_on, allocating memory with anything other than GFP\_ATOMIC, or locking a semaphore. Finally, handlers cannot call schedule.

The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board; most hardware devices won't generate other interrupts until their "interrupt-pending'' bit has been cleared. Some devices don't require this step because they don't have an "interrupt-pending'' bit; such devices are a minority, although the parallel port is one of them. For that reason, short does not have to clear such a bit.

A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for, such as the arrival of new data.

To stick with the frame grabber example, a process could acquire a sequence of images by continuously reading the device; the read call blocks before reading each frame, while the interrupt handler awakens the process as soon as each new frame arrives. This assumes that the grabber interrupts the processor to signal successful arrival of each new frame.

The programmer should be careful to write a routine that executes in a minimum of time, independent of its being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a tasklet or task queue to schedule computation at a safer time.

### 

### Handler Arguments and Return Value:

Though *short* ignores them, three arguments are passed to an interrupt handler: irq, dev\_id, and regs. Let’s look at the role of each. The interrupt number (int irq) is useful as information you may print in your log messages, if any. The second argument, void \*dev\_id, is a sort of client data; a void \* argument is passed to *request\_irq*, and this same pointer is then passed backas and argument to the handler when the interrupt happens. You usually pass a pointer to your device data structure in dev\_id, so a driver that manages several instances of the same device doesn’t need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.

The last argument, struct pt\_regs \*regs, is rarely used. It holds a snapshot of the processor’s context before the processor entered interrupt code. The registers can be used for monitoring and debugging; they are not normally needed for regular device

driver tasks.

Interrupt handlers should return a value indicating whether there was actually an interrupt to handle. If the handler found that its device did, indeed, need attention, it should return IRQ\_HANDLED; otherwise the return value should be IRQ\_NONE. You can also generate the return value with this macro:

**IRQ\_RETVAL(handled)**

Where handled is nonzero if you were able to handle the interrupt. The return value is used by the kernel to detect and suppress spurious interrupts. If your device gives you no way to tell whether it really interrupted, you should return IRQ\_HANDLED.

### Tasklets and Bottom-Half Processing

One of the main problems with interrupt handling is how to perform longish tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt -- the one you register with request\_irq. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The use of the term bottom half in the 2.4 kernel can be a bit confusing, in that it can mean either the second half of an interrupt handler or one of the mechanisms used to implement this second half, or both. When we refer to a bottom half we are speaking generally about a bottom half; the old Linux bottom-half implementation is referred to explicitly with the acronym BH.

But what is a bottom half useful for?

The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half -- that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

One thing to keep in mind with bottom-half processing is that all of the restrictions that apply to interrupt handlers also apply to bottom halves. Thus, bottom halves cannot sleep, cannot access user space, and cannot invoke the scheduler.

### Writing a BH Bottom Half

It's quite apparent from the list of available bottom halves in "The BH Mechanism" that a driver implementing a bottom half should attach its code to IMMEDIATE\_BH by using the immediate queue.

When IMMEDIATE\_BH is marked, the function in charge of the immediate bottom half just consumes the immediate queue. If your interrupt handler queues its BH handler to tq\_immediate and marks the IMMEDIATE\_BH bottom half, the queued task will be called at just the right time. Because in all kernels we are interested in you can queue the same task multiple times without trashing the task queue, you can queue your bottom half every time the top-half handler runs. We'll see this behavior in a while.

Drivers with exotic configurations -- multiple bottom halves or other setups that can't easily be handled with a plain tq\_immediate -- can be satisfied by using a custom task queue. The interrupt handler queues the tasks in its own queue, and when it's ready to run them, a simple queue-consuming function is inserted into the immediate queue.

End of Chapter 7

# 