**Introduction to linux device driver**

## Linux Architecture

Linux is primarily divided into **User Space** & **Kernel Space**. These two components interact through a System Call Interface – which are predefined and matured interface to Linux Kernel for User space applications. The below image will give you the basic understanding.

### [Linux Device Driver Part 1 - Introduction](https://i1.wp.com/www.embetronicx.com/wp-content/uploads/2017/08/kernel-space-vs-user-space.png)Kernel Space

Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services.

### User Space

User Space is where the user applications are executed.

# Linux Kernel Modules

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.  
Custom codes can be added to Linux kernels via two methods.

* The basic way is to add the code to the kernel source tree and recompile the kernel.
* A more efficient way is to do this is by adding code to the kernel while it is running. This process is called loading the module, where module refers to the code that we want to add to the kernel.

Since we are loading these codes at runtime and they are not part of the official Linux kernel, these are called loadable kernel module(LKM), which is different from the “base kernel”. Base kernel is located in /boot directory and is always loaded when we boot our machine whereas LKMs are loaded after the base kernel is already loaded. Nonetheless, these LKM are very much part of our kernel and they communicate with base kernel to complete their functions.

LKMs can perform a variety of task, but basically they come under three main categories,

* Device drivers
* Filesystem drivers
* System calls

## Device drivers

A device driver is designed for a specific piece of hardware. The kernel uses it to communicate with that piece of hardware without having to know any details of how the hardware works.

## Filesystem drivers

A filesystem driver interprets the contents of a filesystem (which is typically the contents of a disk drive) as files and directories and such. There are lots of different ways of storing files and directories and such on disk drives, on network servers, and in other ways. For each way, you need a filesystem driver. For example, there’s a filesystem driver for the ext2 filesystem type used almost universally on Linux disk drives. There is one for the MS-DOS filesystem too, and one for NFS.

## System calls

User space programs use system calls to get services from the kernel. For example, there are system calls to read a file, to create a new process, and to shut down the system. Most system calls are integral to the system and very standard, so are always built into the base kernel (no LKM option).

But you can invent a system call of your own and install it as an LKM. Or you can decide you don’t like the way Linux does something and override an existing system call with an LKM of your own.

## Advantages of LKM

* One major advantage they have is that we don’t need to keep rebuilding the kernel every time we add a new device or if we upgrade an old device. This saves time and also helps in keeping our base kernel error free.
* LKMs are very flexible, in the sense that they can be loaded and unloaded with a single line of command. This helps in saving memory as we load the LKM only when we need them.

## Differences Between Kernel Modules and User Programs

* **Kernel modules have separate address space.** A module runs in kernel space. An application runs in user space. System software is protected from user programs. Kernel space and user space have their own memory address spaces.
* **Kernel modules have higher execution privilege.** Code that runs in kernel space has greater privilege than code that runs in user space.
* **Kernel modules do not execute sequentially.** A user program typically executes sequentially and performs a single task from beginning to end. A kernel module does not execute sequentially. A kernel module registers itself in order to serve future requests.
* **Kernel modules use different header files.** Kernel modules require a different set of header files than user programs require.

## Difference Between Kernel Drivers and Kernel Modules

* A kernel module is a bit of compiled code that can be inserted into the kernel at run-time, such as withinsmod or modprobe.
* A driver is a bit of code that runs in the kernel to talk to some hardware device. It “drives” the hardware. Most every bit of hardware in your computer has an associated driver.

# Device Driver

A device driver is a particular form of software application that is designed to enable interaction with hardware devices. Without the required device driver, the corresponding hardware device fails to work.  
A device driver usually communicates with the hardware by means of the communications subsystem or computer bus to which the hardware is connected. Device drivers are operating system-specific and hardware-dependent. A device driver acts as a translator between the hardware device and the programs or operating systems that use it.

## Types

In the traditional classification, there are three kinds of device:

* Character device
* Block device
* Network device

In Linux everything is a file. I mean Linux treat everything as a File even hardware.

## Character Device

A char file is a hardware file which reads/write data in character by character fashion. Some classic examples are keyboard, mouse, serial printer. If a user use a char file for writing data no other user can use same char file to write data which blocks access to other user. Character files uses synchronize Technic to write data. Of you observe char files are used for communication purpose and they can not be mounted.

## Block Device

A block file is a hardware file which read/write data in blocks instead of character by character. This type of files are very much useful when we want to write/read data in bulk fashion. All our disks such are HDD, USB and CDROMs are block devices. This is the reason when we are formatting we consider block size. The write of data is done in asynchronous fashion and it is CPU intensive activity. These devices files are used to store data on real hardware and can be mounted so that we can access the data we written.

## Network Device

A network device is, so far as Linux’s network subsystem is concerned, an entity that sends and receives packets of data. This is normally a physical device such as an ethernet card. Some network devices though are software only such as the loopback device which is used for sending data to yourself.

This is all about the basics of Linux and device driver. We will move onto Linux Device Driver Programming in our [next tutorial](https://www.embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-2-first-device-driver/).

Material:

# 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.

One of the many advantages of free operating systems, as typified by Linux, is that

their internals are open for all to view. The operating system, once a dark and mysterious

area whose code was restricted to a small number of programmers, can now be

readily examined, understood, and modified by anybody with the requisite skills.

Linux has helped to democratize operating systems. The Linux kernel remains a

large and complex body of code, however, and would-be kernel hackers need an

entry point where they can approach the code without being overwhelmed by complexity.

Often, device drivers provide that gateway.

Device drivers take on a special role in the Linux kernel. They are distinct “black

boxes” that 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.

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. Individuals may need to

know about drivers in order to gain access to a particular device that is of interest to

them. 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.

This book teaches you how to write your own drivers and how to hack around in

related parts of the kernel. We have taken a device-independent approach; the programming

techniques and interfaces are presented, whenever possible, without being

tied to any specific device. Each driver is different; as a driver writer, you need to understand your specific device well. But most of the principles and basic techniques

are the same for all drivers. This book cannot teach you about your device,

but it gives you a handle on the background you need to make your device work.

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

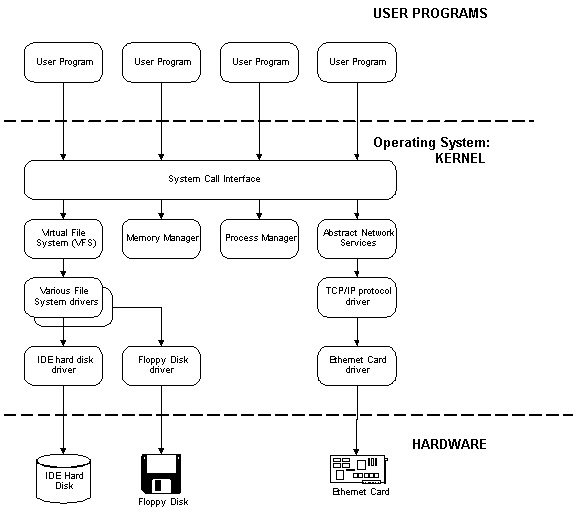
tested.

This chapter doesn’t actually get into writing code. However, we introduce some

background concepts about the Linux kernel that you’ll be glad you know later, when we do launch into programming.

### 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

The Role of the Device Driver

As a programmer, you are able to make your own choices about your driver, and

choose an acceptable trade-off between the programming time required and the flexibility

of the result. Though it may appear strange to say that a driver is “flexible,” we

like this word because it emphasizes that the role of a device driver is providing

*mechanism*, not *policy*.

The distinction between mechanism and policy is one of the best ideas behind the

Unix design. Most programming problems can indeed be split into two parts: “what

capabilities are to be provided” (the mechanism) and “how those capabilities can be

used” (the policy). If the two issues are addressed by different parts of the program,

or even by different programs altogether, the software package is much easier to

develop and to adapt to particular needs.

For example, Unix management of the graphic display is split between the X server,

which knows the hardware and offers a unified interface to user programs, and the

window and session managers, which implement a particular policy without knowing

anything about the hardware. People can use the same window manager on different

hardware, and different users can run different configurations on the same

workstation. Even completely different desktop environments, such as KDE and

GNOME, can coexist on the same system. Another example is the layered structure

of TCP/IP networking: the operating system offers the socket abstraction, which

implements no policy regarding the data to be transferred, while different servers are

in charge of the services (and their associated policies). Moreover, a server like *ftpd*

provides the file transfer mechanism, while users can use whatever client they prefer;

both command-line and graphic clients exist, and anyone can write a new user interface

to transfer files.

Where drivers are concerned, the same separation of mechanism and policy applies.

The floppy driver is policy free—its role is only to show the diskette as a continuous

array of data blocks. Higher levels of the system provide policies, such as who may

access the floppy drive, whether the drive is accessed directly or via a filesystem, and

whether users may mount filesystems on the drive. Since different environments usually

need to use hardware in different ways, it’s important to be as policy free as

possible.

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. The driver should deal

with making the hardware available, leaving all the issues about *how* to use the hardware

to the applications. A driver, then, is flexible if it offers access to the hardware

capabilities without adding constraints. Sometimes, however, some policy decisions

must be made. For example, a digital I/O driver may only offer byte-wide access to

the hardware in order to avoid the extra code needed to handle individual bits.

You can also look at your driver from a different perspective: it is a software layer

that lies between the applications and the actual device. This privileged role of the

driver allows the driver programmer to choose exactly how the device should appear:

different drivers can offer different capabilities, even for the same device. The actual

driver design should be a balance between many different considerations. For

instance, a single device may be used concurrently by different programs, and the

driver programmer has complete freedom to determine how to handle concurrency.

You could implement memory mapping on the device independently of its hardware

capabilities, or you could provide a user library to help application programmers

implement new policies on top of the available primitives, and so forth. One major

consideration is the trade-off between the desire to present the user with as many

options as possible and the time you have to write the driver, as well as the need to

keep things simple so that errors don’t creep in.

Policy-free drivers have a number of typical characteristics. These include support for

both synchronous and asynchronous operation, the ability to be opened multiple

times, the ability to exploit the full capabilities of the hardware, and the lack of software

layers to “simplify things” or provide policy-related operations. Drivers of this

sort not only work better for their end users, but also turn out to be easier to write

and maintain as well. Being policy-free is actually a common target for software

designers.

Many device drivers, indeed, are released together with user programs to help with

configuration and access to the target device. Those programs can range from simple

utilities to complete graphical applications. Examples include the *tunelp* program,

which adjusts how the parallel port printer driver operates, and the graphical *cardctl*

utility that is part of the PCMCIA driver package. Often a client library is provided as

well, which provides capabilities that do not need to be implemented as part of the

driver itself.

The scope of this book is the kernel, so we try not to deal with policy issues or with

application programs or support libraries. Sometimes we talk about different policies

and how to support them, but we won’t go into much detail about programs

using the device or the policies they enforce. You should understand, however, that

user programs are an integral part of a software package and that even policy-free

packages are distributed with configuration files that apply a default behavior to the

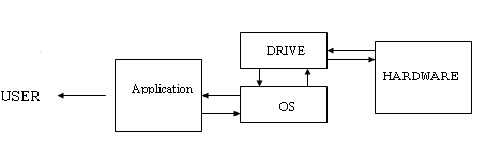
underlying mechanisms.

Role of device drivers:

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.

Of drivers and buses

A driver drives, manages, controls, directs and monitors the entity under its command. What a bus driver does with a bus, a device driver does with a computer device (any piece of hardware connected to a computer) like a mouse, keyboard, monitor, hard disk, Web-camera, clock, and more.

Further, a “pilot” could be a person or even an automatic system monitored by a person (an auto-pilot system in airliners, for example). Similarly, a specific piece of hardware could be controlled by a piece of software (a device driver), or could be controlled by another hardware device, which in turn could be managed by a software device driver. In the latter case, such a controlling device is commonly called a device controller. This, being a device itself, often also needs a driver, which is commonly referred to as a bus driver.

General examples of device controllers include hard disk controllers, display controllers, and audio controllers that in turn manage devices connected to them. More technical examples would be an IDE controller, PCI controller, USB controller, SPI controller, I2C controller, etc. Pictorially, this whole concept can be depicted as in Figure 1.

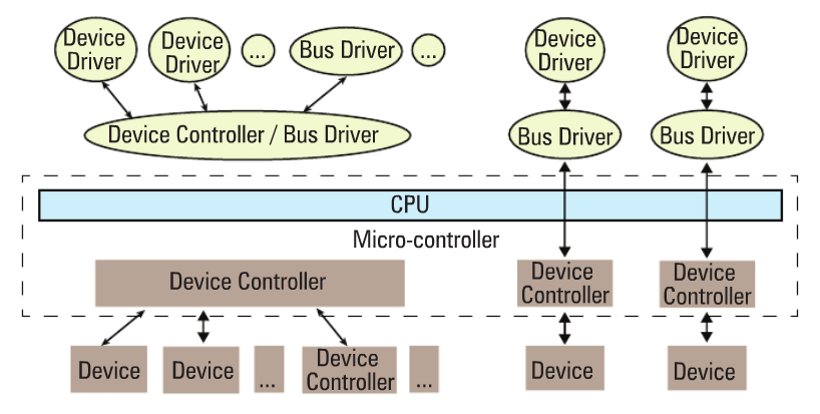


Figure 1: Device and driver interaction

Device controllers are typically connected to the CPU through their respectively named buses (collection of physical lines) — for example, the PCI bus, the IDE bus, etc. In today’s embedded world, we encounter more micro-controllers than CPUs; these are the CPU plus various device controllers built onto a single chip. This effective embedding of device controllers primarily reduces cost and space, making it suitable for embedded systems. In such cases, the buses are integrated into the chip itself. Does this change anything for the drivers, or more generically, on the software front?

The answer is, not much — except that the bus drivers corresponding to the embedded device controllers are now developed under the architecture-specific umbrella.

Drivers have two parts

Bus drivers provide hardware-specific interfaces for the corresponding hardware protocols, and are the bottom-most horizontal software layers of an operating system (OS). Over these sit the actual device drivers. These operate on the underlying devices using the horizontal layer interfaces, and hence are device-specific. However, the whole idea of writing these drivers is to provide an abstraction to the user, and so, at the other “end”, these do provide an interface (which varies from OS to OS). In short, a device driver has two parts, which are: a) device-specific, and b) OS-specific. Refer to Figure 2.

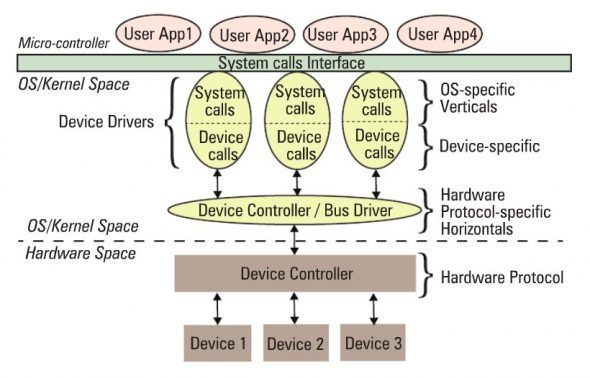
[](http://www.opensourceforu.com/wp-content/uploads/2010/11/ldd2.jpg)

Figure 2: Linux device driver partition

The device-specific portion of a device driver remains the same across all operating systems, and is more about understanding and decoding the device data sheets than software programming. A data sheet for a device is a document with technical details of the device, including its operation, performance, programming, etc. — in short a device user manual.

Later, I shall show some examples of decoding data sheets as well. However, the OS-specific portion is the one that is tightly coupled with the OS mechanisms of user interfaces, and thus differentiates a Linux device driver from a Windows device driver and from a MacOS device driver.

Verticals

In Linux, a device driver provides a “system call” interface to the user; this is the boundary line between the so-called kernel space and user-space of Linux, as shown in Figure 2. Figure 3 provides further classification.

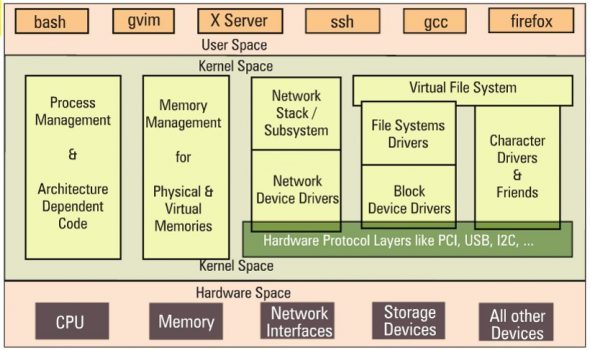
[](http://www.opensourceforu.com/wp-content/uploads/2010/11/ldd3.jpg)

Figure 3: Linux kernel overview

Based on the OS-specific interface of a driver, in Linux, a driver is broadly classified into three verticals:

* Packet-oriented or the network vertical
* Block-oriented or the storage vertical
* Byte-oriented or the character vertical

The CPU vertical and memory vertical, taken together with the other three verticals, give the complete overview of the Linux kernel, like any textbook definition of an OS: “An OS performs 5 management functions: CPU/process, memory, network, storage, device I/O.” Though these two verticals could be classified as device drivers, where CPU and memory are the respective devices, they are treated differently, for many reasons.

These are the core functionalities of any OS, be it a micro-kernel or a monolithic kernel. More often than not, adding code in these areas is mainly a Linux porting effort, which is typically done for a new CPU or architecture. Moreover, the code in these two verticals cannot be loaded or unloaded on the fly, unlike the other three verticals. Henceforth, when we talk about Linux device drivers, we mean to talk only about the latter three verticals in Figure 3.

Let’s get a little deeper into these three verticals. The network vertical consists of two parts: a) the network protocol stack, and b)the network interface card (NIC) device drivers, or simply network device drivers, which could be for Ethernet, Wi-Fi, or any other network horizontals. Storage, again, consists of two parts: a) File-system drivers, to decode the various formats on different partitions, and b) Block device drivers for various storage (hardware) protocols, i.e., horizontals like IDE, SCSI, MTD, etc.

With this, you may wonder if that is the only set of devices for which you need drivers (or for which Linux has drivers). Hold on a moment; you certainly need drivers for the whole lot of devices that interface with the system, and Linux does have drivers for them. However, their byte-oriented cessibility puts all of them under the character vertical — this is, in reality, the majority bucket. In fact, because of the vast number of drivers in this vertical, character drivers have been further sub-classified — so you have tty drivers, input drivers, console drivers, frame-buffer drivers, sound drivers, etc. The typical horizontals here would be RS232, PS/2, VGA, I2C, I2S, SPI, etc.

Multiple-vertical drivers

One final note on the complete picture (placement of all the drivers in the Linux driver ecosystem): the horizontals like USB, PCI, etc, span below multiple verticals. Why is that?

Simple — you already know that you can have a USB Wi-Fi dongle, a USB pen drive, and a USB-to-serial converter — all are USB, but come under three different verticals!

In Linux, bus drivers or the horizontals, are often split into two parts, or even two drivers: a) device controller-specific, and b) an abstraction layer over that for the verticals to interface, commonly called cores. A classic example would be the USB controller drivers ohci, ehci, etc., and the USB abstraction, usbcore.

So, a device driver is a piece of software that drives a device, though there are so many classifications. In case it drives only another piece of software, we call it just a driver. Examples are file-system drivers, usbcore, etc. Hence, all device drivers are drivers, but all drivers are not device drivers.

### 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.

Version Numbering

Before digging into programming, we should comment on the version numbering

scheme used in Linux and which versions are covered by this book.

First of all, note that *every* software package used in a Linux system has its own

release number, and there are often interdependencies across them: you need a particular

version of one package to run a particular version of another package. The

creators of Linux distributions usually handle the messy problem of matching packages,

and the user who installs from a prepackaged distribution doesn’t need to deal

with version numbers. Those who replace and upgrade system software, on the other

hand, are on their own in this regard. Fortunately, almost all modern distributions

support the upgrade of single packages by checking interpackage dependencies; the

distribution’s package manager generally does not allow an upgrade until the dependencies

are satisfied.

To run the examples we introduce during the discussion, you won’t need particular

versions of any tool beyond what the 2.6 kernel requires; any recent Linux distribution

can be used to run our examples. We won’t detail specific requirements,

because the file *Documentation/Changes* in your kernel sources is the best source of

such information if you experience any problems.

As far as the kernel is concerned, the even-numbered kernel versions (i.e., 2.6.*x*) are

the stable ones that are intended for general distribution. The odd versions (such as

2.7.*x*), on the contrary, are development snapshots and are quite ephemeral; the latest

of them represents the current status of development, but becomes obsolete in a

few days or so.

This book covers Version 2.6 of the kernel. Our focus has been to show all the features

available to device driver writers in 2.6.10, the current version at the time we

are writing. This edition of the book does not cover prior versions of the kernel. For

those of you who are interested, the second edition covered Versions 2.0 through 2.4

in detail. That edition is still available online at *http://lwn.net/Kernel/LDD2/*.

Kernel programmers should be aware that the development process changed with 2.6.

The 2.6 series is now accepting changes that previously would have been considered

too large for a “stable” kernel. Among other things, that means that internal kernel

programming interfaces can change, thus potentially obsoleting parts of this book;

for this reason, the sample code accompanying the text is known to work with 2.6.10,

but some modules don’t compile under earlier versions. Programmers wanting to

keep up with kernel programming changes are encouraged to join the mailing lists

and to make use of the web sites listed in the bibliography. There is also a web page

maintained at *http://lwn.net/Articles/2.6-kernel-api/*, which contains information

about API changes that have happened since this book was published.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

License Terms | 11

This text doesn’t talk specifically about odd-numbered kernel versions. General users

never have a reason to run development kernels. Developers experimenting with new

features, however, want to be running the latest development release. They usually

keep upgrading to the most recent version to pick up bug fixes and new implementations

of features. Note, however, that there’s no guarantee on experimental kernels,\*

and nobody helps you if you have problems due to a bug in a noncurrent odd-numbered

kernel. Those who run odd-numbered versions of the kernel are usually skilled

enough to dig in the code without the need for a textbook, which is another reason

why we don’t talk about development kernels here.

Another feature of Linux is that it is a platform-independent operating system, not

just “a Unix clone for PC clones” anymore: it currently supports some 20 architectures.

This book is platform independent as far as possible, and all the code samples

have been tested on at least the x86 and x86-64 platforms. Because the code has been

tested on both 32-bit and 64-bit processors, it should compile and run on all other

platforms. As you might expect, the code samples that rely on particular hardware

don’t work on all the supported platforms, but this is always stated in the source

code.

**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.

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. Although 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:

*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 complex 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, disks may

be formatted with the Linux-standard ext3 filesystem, the commonly used FAT

filesystem or several others.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Classes of Devices and Modules | 5

*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 drive. 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.

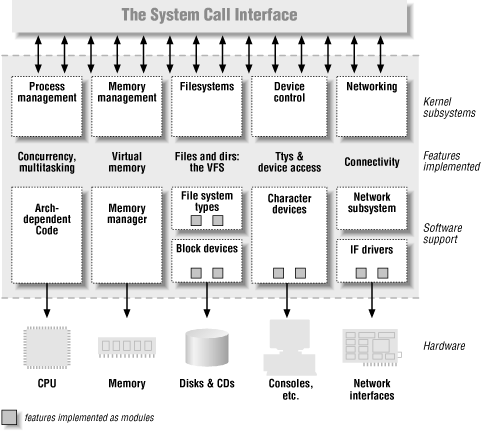
### Features of linux kernel:

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 Linux way of looking at devices distinguishes between three fundamental 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*. This division of modules into different

types, or classes, is not a rigid one; the programmer can choose to build huge

modules implementing different drivers in a single chunk of code. Good programmers,

nonetheless, usually create a different module for each new functionality they

implement, because decomposition is a key element of scalability and extendability.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

6 | Chapter 1: An Introduction to Device Drivers

The three classes are:

*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*.

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

*features implemented as modules*

**Process**

**management**

**Memory**

**management**

**Filesystems Device**

**control**

**Networking**

**Archdependent**

**code**

**Memory**

**manager**

**Character**

**devices**

**Network**

**subsystem**

**CPU Memory**

**Concurrency,**

**multitasking**

**Virtual**

**memory**

**Files and dirs:**

**the VFS**

*Kernel*

*subsystems*

*Features*

*implemented*

*Software*

*support*

*Hardware*

**Block devices IF drivers**

**File system**

**types**

**Ttys &**

**device access Connectivity**

**Disks & CDs Consoles,**

**etc.**

**Network**

**interfaces**

**The System Call Interface**

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Classes of Devices and Modules | 7

*Block devices*

Like char devices, block devices are accessed by filesystem nodes in the */dev*

directory. A block device is a device (e.g., a disk) that can host a filesystem. In

most Unix systems, a block device can only handle I/O operations that transfer

one or more whole blocks, which are usually 512 bytes (or a larger power of

two) bytes in length. Linux, instead, 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. Block drivers have a completely

different interface to the kernel than char drivers.

*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. Many network connections

(especially those using TCP) are stream-oriented, but network devices are, usually,

designed around the transmission and receipt of packets. A network driver

knows nothing about individual connections; it only handles packets.

Not being a stream-oriented device, a network interface isn’t easily mapped to a

node in the filesystem, as */dev/tty1* is. The Unix way to provide access to interfaces

is still by assigning a unique name to them (such as eth0), but that name

doesn’t have a corresponding entry in the filesystem. Communication between

the kernel and a network device driver is completely different from that used

with char and block drivers. Instead of *read* and *write*, the kernel calls functions

related to packet transmission.

There are other ways of classifying driver modules that are orthogonal to the above

device types. In general, some types of drivers work with additional layers of kernel

support functions for a given type of device. For example, one can talk of universal

serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB

device is driven by a USB module that works with the USB subsystem, but the device

itself shows up in the system as a char device (a USB serial port, say), a block device

(a USB memory card reader), or a network device (a USB Ethernet interface).

Other classes of device drivers have been added to the kernel in recent times, including

FireWire drivers and I2O drivers. In the same way that they handled USB and

SCSI drivers, kernel developers collected class-wide features and exported them to

driver implementers to avoid duplicating work and bugs, thus simplifying and

strengthening the process of writing such drivers.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

8 | Chapter 1: An Introduction to Device Drivers

In addition to device drivers, other functionalities, both hardware and software, are

modularized in the kernel. One common example is filesystems. A filesystem type

determines how information is organized on a block device in order to represent a

tree of directories and files. Such an entity is not a device driver, in that there’s no

explicit device associated with the way the information is laid down; the filesystem

type is instead a software driver, because it maps the low-level data structures to

high-level data structures. It is the filesystem that determines how long a filename

can be and what information about each file is stored in a directory entry. The filesystem

module must implement the lowest level of the system calls that access directories

and files, by mapping filenames and paths (as well as other information, such

as access modes) to data structures stored in data blocks. Such an interface is completely

independent of the actual data transfer to and from the disk (or other

medium), which is accomplished by a block device driver.

If you think of how strongly a Unix system depends on the underlying filesystem,

you’ll realize that such a software concept is vital to system operation. The ability to

decode filesystem information stays at the lowest level of the kernel hierarchy and is

of utmost importance; even if you write a block driver for your new CD-ROM, it is

useless if you are not able to run *ls* or *cp* on the data it hosts. Linux supports the concept

of a filesystem module, whose software interface declares the different operations

that can be performed on a filesystem inode, directory, file, and superblock. It’s

quite unusual for a programmer to actually need to write a filesystem module,

because the official kernel already includes code for the most important filesystem

types.

### 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.

Security Issues

Security is an increasingly important concern in modern times. We will discuss security-

related issues as they come up throughout the book. There are a few general concepts,

however, that are worth mentioning now.

Any security check in the system is enforced by kernel code. If the kernel has security

holes, then the system as a whole has holes. In the official kernel distribution,

only an authorized user can load modules; the system call *init\_module* checks if the

invoking process is authorized to load a module into the kernel. Thus, when running

an official kernel, only the superuser,\* or an intruder who has succeeded in

becoming privileged, can exploit the power of privileged code.

When possible, driver writers should avoid encoding security policy in their code.

Security is a policy issue that is often best handled at higher levels within the kernel,

under the control of the system administrator. There are always exceptions, however.

\* Technically, only somebody with the CAP\_SYS\_MODULE capability can perform this operation. We discuss

capabilities in Chapter 6.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Security Issues | 9

As a device driver writer, you should be aware of situations in which some types of

device access could adversely affect the system as a whole and should provide adequate

controls. For example, device operations that affect global resources (such as

setting an interrupt line), which could damage the hardware (loading firmware, for

example), or that could affect other users (such as setting a default block size on a

tape drive), are usually only available to sufficiently privileged users, and this check

must be made in the driver itself.

Driver writers must also be careful, of course, to avoid introducing security bugs.

The C programming language makes it easy to make several types of errors. Many

current security problems are created, for example, by *buffer overrun* errors, in which

the programmer forgets to check how much data is written to a buffer, and data ends

up written beyond the end of the buffer, thus overwriting unrelated data. Such errors

can compromise the entire system and must be avoided. Fortunately, avoiding these

errors is usually relatively easy in the device driver context, in which the interface to

the user is narrowly defined and highly controlled.

Some other general security ideas are worth keeping in mind. Any input received

from user processes should be treated with great suspicion; never trust it unless you

can verify it. Be careful with uninitialized memory; any memory obtained from the

kernel should be zeroed or otherwise initialized before being made available to a user

process or device. Otherwise, information leakage (disclosure of data, passwords,

etc.) could result. If your device interprets data sent to it, be sure the user cannot

send anything that could compromise the system. Finally, think about the possible

effect of device operations; if there are specific operations (e.g., reloading the firmware

on an adapter board or formatting a disk) that could affect the system, those

operations should almost certainly be restricted to privileged users.

Be careful, also, when receiving software from third parties, especially when the kernel

is concerned: because everybody has access to the source code, everybody can

break and recompile things. Although you can usually trust precompiled kernels

found in your distribution, you should avoid running kernels compiled by an

untrusted friend—if you wouldn’t run a precompiled binary as root, then you’d better

not run a precompiled kernel. For example, a maliciously modified kernel could

allow anyone to load a module, thus opening an unexpected back door via *init\_module*.

Note that the Linux kernel can be compiled to have no module support whatsoever,

thus closing any module-related security holes. In this case, of course, all needed

drivers must be built directly into the kernel itself. It is also possible, with 2.2 and

later kernels, to disable the loading of kernel modules after system boot via the capability

mechanism.

Loadable Modules

One of the good features of Linux is the ability to extend at runtime the set of features

offered by the kernel. This means that you can add functionality to the kernel

(and remove functionality as well) while the system is up and running.

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.

Figure 1-1 identifies different classes of modules in charge of specific tasks—a module

is said to belong to a specific class according to the functionality it offers. The

placement of modules in Figure 1-1 covers the most important classes, but is far from

complete because more and more functionality in Linux is being modularized.

### 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

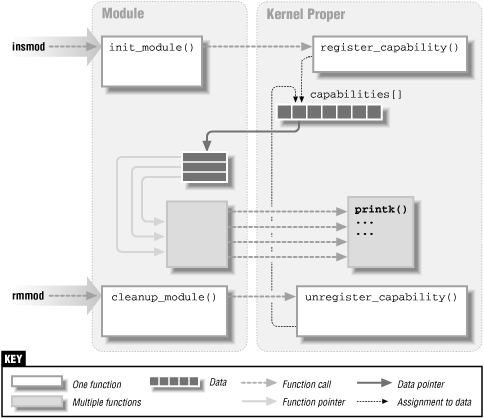
on currently loaded modules can also be found in the sysfs virtual filesystem

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

Building and Running

Modules

It’s almost time to begin programming. This chapter introduces all the essential concepts

about modules and kernel programming. In these few pages, we build and run

a complete (if relatively useless)module, and look at some of the basic code shared

by all modules. Developing such expertise is an essential foundation for any kind of

modularized driver. To avoid throwing in too many concepts at once, this chapter

talks only about modules, without referring to any specific device class.

All the kernel items (functions, variables, header files, and macros)that are introduced

here are described in a reference section at the end of the chapter.

Setting Up Your Test System

Starting with this chapter, we present example modules to demonstrate programming

concepts. (All of these examples are available on O’Reilly’s FTP site, as

explained in Chapter 1.)Building, loading, and modifying these examples are a good

way to improve your understanding of how drivers work and interact with the kernel.

The example modules should work with almost any 2.6.x kernel, including those

provided by distribution vendors. However, we recommend that you obtain a “mainline”

kernel directly from the *kernel.org* mirror network, and install it on your system.

Vendor kernels can be heavily patched and divergent from the mainline; at

times, vendor patches can change the kernel API as seen by device drivers. If you are

writing a driver that must work on a particular distribution, you will certainly want

to build and test against the relevant kernels. But, for the purpose of learning about

driver writing, a standard kernel is best.

Regardless of the origin of your kernel, building modules for 2.6.x requires that you

have a configured and built kernel tree on your system. This requirement is a change

from previous versions of the kernel, where a current set of header files was sufficient.

2.6 modules are linked against object files found in the kernel source tree; the

result is a more robust module loader, but also the requirement that those object files

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

16 | Chapter 2: Building and Running Modules

be available. So your first order of business is to come up with a kernel source tree

(either from the *kernel.org* network or your distributor’s kernel source package),

build a new kernel, and install it on your system. For reasons we’ll see later, life is

generally easiest if you are actually running the target kernel when you build your

modules, though this is not required.

You should also give some thought to where you do your module

experimentation, development, and testing. We have done our best to

make our example modules safe and correct, but the possibility of

bugs is always present. Faults in kernel code can bring about the

demise of a user process or, occasionally, the entire system. They do

not normally create more serious problems, such as disk corruption.

Nonetheless, it is advisable to do your kernel experimentation on a

system that does not contain data that you cannot afford to lose, and

that does not perform essential services. Kernel hackers typically keep

a “sacrificial” system around for the purpose of testing new code.

So, if you do not yet have a suitable system with a configured and built kernel source

tree on disk, now would be a good time to set that up. We’ll wait. Once that task is

taken care of, you’ll be ready to start playing with kernel modules.

The Hello World Module

Many programming books begin with a “hello world” example as a way of showing

the simplest possible program. This book deals in kernel modules rather than programs;

so, for the impatient reader, the following code is a complete “hello world”

module:

#include <linux/init.h>

#include <linux/module.h>

MODULE\_LICENSE("Dual BSD/GPL");

static int hello\_init(void)

{

printk(KERN\_ALERT "Hello, world\n");

return 0;

}

static void hello\_exit(void)

{

printk(KERN\_ALERT "Goodbye, cruel world\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

This module defines two functions, one to be invoked when the module is loaded

into the kernel (*hello\_init*)and one for when the module is removed (*hello\_exit*). The

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

The Hello World Module | 17

*module\_init* and *module\_exit* lines use special kernel macros to indicate the role of

these two functions. Another special macro (*MODULE\_LICENSE*)is used to tell the

kernel that this module bears a free license; without such a declaration, the kernel

complains when the module is loaded.

The *printk* function is defined in the Linux kernel and made available to modules; it

behaves similarly to the standard C library function *printf*. The kernel needs its own

printing function because it runs by itself, without the help of the C library. The

module can call *printk* because, after *insmod* has loaded it, the module is linked to

the kernel and can access the kernel’s public symbols (functions and variables, as

detailed in the next section). The string KERN\_ALERT is the priority of the message.\*

We’ve specified a high priority in this module, because a message with the default

priority might not show up anywhere useful, depending on the kernel version you

are running, the version of the *klogd* daemon, and your configuration. You can

ignore this issue for now; we explain it in Chapter 4.

You can test the module with the *insmod* and *rmmod* utilities, as shown below. Note

that only the superuser can load and unload a module.

% **make**

make[1]: Entering directory `/usr/src/linux-2.6.10'

CC [M] /home/ldd3/src/misc-modules/hello.o

Building modules, stage 2.

MODPOST

CC /home/ldd3/src/misc-modules/hello.mod.o

LD [M] /home/ldd3/src/misc-modules/hello.ko

make[1]: Leaving directory `/usr/src/linux-2.6.10'

% **su**

root# **insmod ./hello.ko**

Hello, world

root# **rmmod hello**

Goodbye cruel world

root#

Please note once again that, for the above sequence of commands to work, you must

have a properly configured and built kernel tree in a place where the makefile is able

to find it (*/usr/src/linux-2.6.10* in the example shown). We get into the details of how

modules are built in the section “Compiling and Loading.”

According to the mechanism your system uses to deliver the message lines, your output

may be different. In particular, the previous screen dump was taken from a text

console; if you are running *insmod* and *rmmod* from a terminal emulator running

under the window system, you won’t see anything on your screen. The message goes

to one of the system log files, such as */var/log/messages* (the name of the actual file

\* The priority is just a string, such as <1>, which is prepended to the *printk* format string. Note the lack of a

comma after KERN\_ALERT; adding a comma there is a common and annoying typo (which, fortunately, is

caught by the compiler).

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

18 | Chapter 2: Building and Running Modules

varies between Linux distributions). The mechanism used to deliver kernel messages

is described in Chapter 4.

As you can see, writing a module is not as difficult as you might expect—at least, as

long as the module is not required to do anything worthwhile. The hard part is

understanding your device and how to maximize performance. We go deeper into

modularization throughout this chapter and leave device-specific issues for later

chapters.

Kernel Modules Versus Applications

Before we go further, it’s worth underlining the various differences between a kernel

module and an application.

While most small and medium-sized applications perform a single task from beginning

to end, every kernel module just registers itself in order to serve future requests,

and its initialization function terminates immediately. In other words, the task of the

module’s initialization function 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 module’s exit function (*hello\_exit* in the example)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.” This kind of approach to programming is similar to eventdriven

programming, but while not all applications are event-driven, each and every

kernel module is. Another major difference between event-driven applications and

kernel code is in the exit function: whereas an application that terminates can be lazy

in releasing resources or avoids clean up altogether, the exit function of a module

must carefully undo everything the *init* function built up, or the pieces remain

around until the system is rebooted.

Incidentally, 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.

As a programmer, you know that an application can call functions it doesn’t define:

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.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Kernel Modules Versus Applications | 19

Because no library is linked to modules, source files should never include the usual

header files, *<stdarg.h>* and very special situations being the only exceptions. 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 the kernel source tree

you have set up and configured; most of the relevant headers live in *include/linux* and

*include/asm*, but other subdirectories of *include* have been added to host material

associated to specific kernel subsystems.

The role of individual kernel headers is introduced throughout the book as each of

them is needed.

Another important difference between kernel programming and application programming

is in how each environment handles faults: whereas a segmentation fault

is harmless during application development and a debugger can always be used to

trace the error to the problem in the source code, a kernel fault kills the current process

at least, if not the whole system. We see how to trace kernel errors in Chapter 4.

User Space and Kernel Space

A module runs in *kernel space*, whereas applications run in *user space*. This concept

is at the base of operating systems theory.

The role of the operating system, in practice, is to provide programs with a consistent

view of the computer’s hardware. In addition, the operating system must

account for independent operation of programs and protection against unauthorized

access to resources. This nontrivial task is possible only if the CPU enforces protection

of system software from the applications.

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

insmod init function

blk\_init\_queue()

add\_disk()

request()

block\_device ops

cleanup

rmmod function

del\_gendisk()

blk\_cleanup\_queue()

request\_queue\_

struct

gendisk

Data operation

Data pointer

Function call

Function pointer

Multiple functions

Single functions

Data

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

20 | Chapter 2: Building and Running Modules

Every modern processor is able to enforce this behavior. The chosen approach is to

implement different operating modalities (or levels)in the CPU itself. The levels have

different roles, and some operations are disallowed at the lower levels; program code

can switch from one level to another only through a limited number of gates. Unix

systems are designed to take advantage of this hardware feature, using two such levels.

All current processors have at least two protection levels, and some, like the x86

family, have more levels; when several levels exist, the highest and lowest levels are

used. Under Unix, the kernel executes in the highest level (also called *supervisor*

*mode*), where everything is allowed, whereas applications execute in the lowest level

(the so-called *user mode*), where the processor regulates direct access to hardware

and unauthorized access to memory.

We usually refer to the execution modes as *kernel space* and *user space*. These terms

encompass not only the different privilege levels inherent in the two modes, but also

the fact that each mode can have its own memory mapping—its own address

space—as well.

Unix transfers execution from user space to kernel space whenever an application

issues a system call or is suspended by a hardware interrupt. Kernel code executing a

system call is working in the context of a process—it operates on behalf of the calling

process and is able to access data in the process’s address space. Code that handles

interrupts, on the other hand, is asynchronous with respect to processes and is

not related to any particular process.

The role of a module is to extend kernel functionality; modularized code runs in kernel

space. Usually a driver performs both the tasks outlined previously: some functions

in the module are executed as part of system calls, and some are in charge of

interrupt handling.

Concurrency in the Kernel

One way in which kernel programming differs greatly from conventional application

programming is the issue of concurrency. Most applications, with the notable exception

of multithreading applications, typically run sequentially, from the beginning to

the end, without any need to worry about what else might be happening to change

their environment. Kernel code does not run in such a simple world, and even the

simplest kernel modules must be written with the idea that many things can be happening

at once.

There are a few sources of concurrency in kernel programming. Naturally, Linux systems

run multiple processes, more than one of which can be trying to use your driver

at the same time. Most devices are capable of interrupting the processor; interrupt

handlers run asynchronously and can be invoked at the same time that your driver is

trying to do something else. Several software abstractions (such as kernel timers,

introduced in Chapter 7)run asynchronously as well. Moreover, of course, Linux

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Kernel Modules Versus Applications | 21

can run on symmetric multiprocessor (SMP)systems, with the result that your driver

could be executing concurrently on more than one CPU. Finally, in 2.6, kernel code

has been made preemptible; this change causes even uniprocessor systems to have

many of the same concurrency issues as multiprocessor systems.

As a result, Linux kernel code, including driver code, must be *reentrant*—it must be

capable of running in more than one context at the same time. Data structures must

be carefully designed to keep multiple threads of execution separate, and the code

must take care to access shared data in ways that prevent corruption of the data.

Writing code that handles concurrency and avoids race conditions (situations in

which an unfortunate order of execution causes undesirable behavior)requires

thought and can be tricky. Proper management of concurrency is required to write

correct kernel code; for that reason, every sample driver in this book has been written

with concurrency in mind. The techniques used are explained as we come to

them; Chapter 5 has also been dedicated to this issue and the kernel primitives available

for concurrency management.

A common mistake made by driver programmers is to assume that concurrency is

not a problem as long as a particular segment of code does not go to sleep (or

“block”). Even in previous kernels (which were not preemptive), this assumption

was not valid on multiprocessor systems. In 2.6, kernel code can (almost)never

assume that it can hold the processor over a given stretch of code. If you do not write

your code with concurrency in mind, it will be subject to catastrophic failures that

can be exceedingly difficult to debug.

The Current Process

Although kernel modules don’t execute sequentially as applications do, 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. An example of this technique is presented in Chapter 6.

Actually, current is not truly a global variable. The need to support SMP systems

forced the kernel developers to develop a mechanism that finds the current process on

the relevant CPU. This mechanism must also be fast, since references to current happen

frequently. The result is an architecture-dependent mechanism that, usually,

hides a pointer to the task\_struct structure on the kernel stack. The details of the

implementation remain hidden to other kernel subsystems though, and a device

driver can just include *<linux/sched.h>* and refer to the current process. For example,

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

22 | Chapter 2: Building and Running Modules

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

(trimmed to 15 characters if need be) that is being executed by the current process.

A Few Other Details

Kernel programming differs from user-space programming in many ways. We’ll

point things out as we get to them over the course of the book, but there are a few

fundamental issues which, while not warranting a section of their own, are worth a

mention. So, as you dig into the kernel, the following issues should be kept in mind.

Applications are laid out in virtual memory with a very large stack area. The stack, of

course, is used to hold the function call history and all automatic variables created by

currently active functions. The kernel, instead, has a very small stack; it can be as

small as a single, 4096-byte page. Your functions must share that stack with the

entire kernel-space call chain. Thus, it is never a good idea to declare large automatic

variables; if you need larger structures, you should allocate them dynamically

at call time.

Often, as you look at the kernel API, you will encounter function names starting with

a double underscore (\_\_). Functions so marked are generally a low-level component

of the interface and should be used with caution. Essentially, the double underscore

says to the programmer: “If you call this function, be sure you know what you are

doing.”

Kernel code cannot do floating point arithmetic. Enabling floating point would

require that the kernel save and restore the floating point processor’s state on each

entry to, and exit from, kernel space—at least, on some architectures. Given that

there really is no need for floating point in kernel code, the extra overhead is not

worthwhile.

Compiling and Loading

The “hello world” example at the beginning of this chapter included a brief demonstration

of building a module and loading it into the system. There is, of course, a lot

more to that whole process than we have seen so far. This section provides more

detail on how a module author turns source code into an executing subsystem within

the kernel.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Compiling and Loading | 23

Compiling Modules

As the first step, we need to look a bit at how modules must be built. The build process

for modules differs significantly from that used for user-space applications; the

kernel is a large, standalone program with detailed and explicit requirements on how

its pieces are put together. The build process also differs from how things were done

with previous versions of the kernel; the new build system is simpler to use and produces

more correct results, but it looks very different from what came before. The

kernel build system is a complex beast, and we just look at a tiny piece of it. The files

found in the *Documentation/kbuild* directory in the kernel source are required reading

for anybody wanting to understand all that is really going on beneath the surface.

There are some prerequisites that you must get out of the way before you can build

kernel modules. The first is to ensure that you have sufficiently current versions of the

compiler, module utilities, and other necessary tools. The file *Documentation/Changes*

in the kernel documentation directory always lists the required tool versions; you

should consult it before going any further. Trying to build a kernel (and its modules)

with the wrong tool versions can lead to no end of subtle, difficult problems. Note

that, occasionally, a version of the compiler that is too new can be just as problematic

as one that is too old; the kernel source makes a great many assumptions about the

compiler, and new releases can sometimes break things for a while.

If you still do not have a kernel tree handy, or have not yet configured and built that

kernel, now is the time to go do it. You cannot build loadable modules for a 2.6 kernel

without this tree on your filesystem. It is also helpful (though not required)to be

actually running the kernel that you are building for.

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

In fact, for the “hello world” example shown earlier in this chapter, a single

line will suffice:

obj-m := hello.o

Readers who are familiar with *make*, but not with the 2.6 kernel build system, are

likely to be wondering how this makefile works. The above line is not how a traditional

makefile looks, after all. The answer, of course, is that the kernel build system

handles the rest. 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,

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

24 | Chapter 2: Building and Running Modules

your *~/kernel-2.6* directory, the *make* command required to build your module

(typed in the directory containing the module source and makefile) would be:

make -C ~/kernel-2.6 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. The trick is to write your makefile as follows:

# If KERNELRELEASE is defined, we've been invoked from the

# kernel build system and can use its language.

ifneq ($(KERNELRELEASE),)

obj-m := hello.o

# Otherwise we were called directly from the command

# line; invoke the kernel build system.

else

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

PWD := $(shell pwd)

default:

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

endif

Once again, we are seeing the extended GNU *make* syntax in action. This makefile is

read twice on a typical build. When the makefile is invoked from the command line,

it notices that the KERNELRELEASE variable has not been set. It locates the kernel source

directory by taking advantage of the fact that the symbolic link *build* in the installed

modules directory points back at the kernel build tree. If you are not actually running

the kernel that you are building for, you can supply a KERNELDIR= option on the

command line, set the KERNELDIR environment variable, or rewrite the line that sets

KERNELDIR in the makefile. Once the kernel source tree has been found, the makefile

invokes the default: target, which runs a second *make* command (parameterized in

the makefile as $(MAKE))to invoke the kernel build system as described previously.

On the second reading, the makefile sets obj-m, and the kernel makefiles take care of

actually building the module.

This mechanism for building modules may strike you as a bit unwieldy and obscure.

Once you get used to it, however, you will likely appreciate the capabilities that have

been programmed into the kernel build system. Do note that the above is not a complete

makefile; a real makefile includes the usual sort of targets for cleaning up

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Compiling and Loading | 25

unneeded files, installing modules, etc. See the makefiles in the example source

directory for a complete example.

Loading and Unloading Modules

After the module is built, the next step is loading it into the kernel. As we’ve already

pointed out, *insmod* does the job for you. The program loads the module code and

data into the kernel, which, in turn, performs a function similar to that of *ld*, in that

it links any unresolved symbol in the module to the symbol table of the kernel.

Unlike the linker, however, the kernel doesn’t modify the module’s disk file, but

rather an in-memory copy. *insmod* accepts a number of command-line options (for

details, see the manpage), and it can assign values to parameters in your module

before linking it to the current kernel. Thus, if a module is correctly designed, it can

be configured at load time; load-time configuration gives the user more flexibility

than compile-time configuration, which is still used sometimes. Load-time configuration

is explained in the section “Module Parameters,” later in this chapter.

Interested readers may want to look at how the kernel supports *insmod*: it relies on a

system call defined in *kernel/module.c*. The function *sys\_init\_module* allocates kernel

memory to hold a module (this memory is allocated with *vmalloc*; see the section

“vmalloc and Friends” in Chapter 8); it then copies the module text into that memory

region, resolves kernel references in the module via the kernel symbol table, and

calls the module’s initialization function to get everything going.

If you actually look in the kernel source, you’ll find that the names of the system calls

are prefixed with sys\_. This is true for all system calls and no other functions; it’s

useful to keep this in mind when grepping for the system calls in the sources.

The *modprobe* utility is worth a quick mention. *modprobe*, like *insmod*, loads a module

into the kernel. It differs in that it will look at the module to be loaded to see

whether it references any symbols that are not currently defined in the kernel. If any

such references are found, *modprobe* looks for other modules in the current module

search path that define the relevant symbols. When *modprobe* finds those modules

(which are needed by the module being loaded), it loads them into the kernel as well.

If you use *insmod* in this situation instead, the command fails with an “unresolved

symbols” message left in the system logfile.

As mentioned before, modules may be removed from the kernel with the *rmmod* utility.

Note that module removal fails if the kernel believes that the module is still in

use (e.g., a program still has an open file for a device exported by the modules), or if

the kernel has been configured to disallow module removal. It is possible to configure

the kernel to allow “forced” removal of modules, even when they appear to be

busy. If you reach a point where you are considering using this option, however,

things are likely to have gone wrong badly enough that a reboot may well be the better

course of action.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

26 | Chapter 2: Building and Running Modules

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*.

Version Dependency

Bear in mind that your module’s code has to be recompiled for each version of the

kernel that it is linked to—at least, in the absence of modversions, not covered here

as they are more for distribution makers than developers. Modules are strongly tied

to the data structures and function prototypes defined in a particular kernel version;

the interface seen by a module can change significantly from one kernel version to

the next. This is especially true of development kernels, of course.

The kernel does not just assume that a given module has been built against the

proper kernel version. One of the steps in the build process is to link your module

against a file (called *vermagic.o*)from the current kernel tree; this object contains a

fair amount of information about the kernel the module was built for, including the

target kernel version, compiler version, and the settings of a number of important

configuration variables. When an attempt is made to load a module, this information

can be tested for compatibility with the running kernel. If things don’t match,

the module is not loaded; instead, you see something like:

# **insmod hello.ko**

Error inserting './hello.ko': -1 Invalid module format

A look in the system log file (*/var/log/messages* or whatever your system is configured

to use) will reveal the specific problem that caused the module to fail to load.

If you need to compile a module for a specific kernel version, you will need to use the

build system and source tree for that particular version. A simple change to the

KERNELDIR variable in the example makefile shown previously does the trick.

Kernel interfaces often change between releases. If you are writing a module that is

intended to work with multiple versions of the kernel (especially if it must work

across major releases), you likely have to make use of macros and #ifdef constructs

to make your code build properly. This edition of this book only concerns itself with

one major version of the kernel, so you do not often see version tests in our example

code. But the need for them does occasionally arise. In such cases, you want to make

use of the definitions found in *linux/version.h*. This header file, automatically

included by *linux/module.h*, defines the following macros:

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Compiling and Loading | 27

UTS\_RELEASE

This macro expands to a string describing the version of this kernel tree. For

example, "2.6.10".

LINUX\_VERSION\_CODE

This macro expands to the binary representation of the kernel version, one byte

for each part of the version release number. For example, the code for 2.6.10 is

132618 (i.e., 0x02060a).\* With this information, you can (almost)easily determine

what version of the kernel you are dealing with.

KERNEL\_VERSION(major,minor,release)

This is the macro used to build an integer version code from the individual numbers

that build up a version number. For example, KERNEL\_VERSION(2,6,10)

expands to 132618. This macro is very useful when you need to compare the

current version and a known checkpoint.

Most dependencies based on the kernel version can be worked around with preprocessor

conditionals by exploiting KERNEL\_VERSION and LINUX\_VERSION\_CODE. Version

dependency should, however, not clutter driver code with hairy #ifdef conditionals;

the best way to deal with incompatibilities is by confining them to a specific header

file. As a general rule, code which is explicitly version (or platform)dependent

should be hidden behind a low-level macro or function. High-level code can then

just call those functions without concern for the low-level details. Code written in

this way tends to be easier to read and more robust.

Platform Dependency

Each computer platform has its peculiarities, and kernel designers are free to exploit

all the peculiarities to achieve better performance in the target object file.

Unlike application developers, who must link their code with precompiled libraries

and stick to conventions on parameter passing, kernel developers can dedicate some

processor registers to specific roles, and they have done so. Moreover, kernel code

can be optimized for a specific processor in a CPU family to get the best from the target

platform: unlike applications that are often distributed in binary format, a custom

compilation of the kernel can be optimized for a specific computer set.

For example, the IA32 (x86)architecture has been subdivided into several different

processor types. The old 80386 processor is still supported (for now), even though

its instruction set is, by modern standards, quite limited. The more modern processors

in this architecture have introduced a number of new capabilities, including

faster instructions for entering the kernel, interprocessor locking, copying data, etc.

Newer processors can also, when operated in the correct mode, employ 36-bit (or

\* This allows up to 256 development versions between stable versions.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

28 | Chapter 2: Building and Running Modules

larger)physical addresses, allowing them to address more than 4 GB of physical

memory. Other processor families have seen similar improvements. The kernel,

depending on various configuration options, can be built to make use of these additional

features.

Clearly, if a module is to work with a given kernel, it must be built with the same

understanding of the target processor as that kernel was. Once again, the *vermagic.o*

object comes in to play. When a module is loaded, the kernel checks the processorspecific

configuration options for the module and makes sure they match the running

kernel. If the module was compiled with different options, it is not loaded.

If you are planning to write a driver for general distribution, you may well be wondering

just how you can possibly support all these different variations. The best

answer, of course, is to release your driver under a GPL-compatible license and contribute

it to the mainline kernel. Failing that, distributing your driver in source form

and a set of scripts to compile it on the user’s system may be the best answer. Some

vendors have released tools to make this task easier. If you must distribute your

driver in binary form, you need to look at the different kernels provided by your target

distributions, and provide a version of the module for each. Be sure to take into

account any errata kernels that may have been released since the distribution was

produced. Then, there are licensing issues to be considered, as we discussed in the

section “License Terms” in Chapter 1. As a general rule, distributing things in source

form is an easier way to make your way in the world.

The Kernel Symbol Table

We’ve seen how *insmod* resolves undefined symbols against the table of public kernel

symbols. The table contains the addresses of global kernel items—functions and

variables—that are needed to implement modularized drivers. When a module is

loaded, any symbol exported by the module becomes part of the kernel symbol table.

In the usual case, a module implements its own functionality without the need to

export any symbols at all. You need to export symbols, however, whenever other

modules may benefit from using them.

New modules can use symbols exported by your module, and you can stack new

modules on top of other modules. Module stacking is implemented in the mainstream

kernel sources as well: the *msdos* filesystem relies on symbols exported by the

*fat* module, and each input USB device module stacks on the *usbcore* and *input* modules.

Module stacking is useful in complex projects. If a new abstraction is implemented in

the form of a device driver, it might offer a plug for hardware-specific implementations.

For example, the video-for-linux set of drivers is split into a generic module that

exports symbols used by lower-level device drivers for specific hardware. According to

your setup, you load the generic video module and the specific module for your

installed hardware. Support for parallel ports and the wide variety of attachable

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

The Kernel Symbol Table | 29

devices is handled in the same way, as is the USB kernel subsystem. Stacking in the

parallel port subsystem is shown in Figure 2-2; the arrows show the communications

between the modules and with the kernel programming interface.

When using stacked modules, it is helpful to be aware of the *modprobe* utility. As we

described earlier, *modprobe* functions in much the same way as *insmod*, but it also

loads any other modules that are required by the module you want to load. Thus,

one *modprobe* command can sometimes replace several invocations of *insmod*

(although you’ll still need *insmod* when loading your own modules from the current

directory, because *modprobe* looks only in the standard installed module directories).

Using stacking to split modules into multiple layers can help reduce development

time by simplifying each layer. This is similar to the separation between mechanism

and policy that we discussed in Chapter 1.

The Linux kernel header files provide a convenient way to manage the visibility of

your symbols, thus reducing namespace pollution (filling the namespace with names

that may conflict with those defined elsewhere in the kernel)and promoting proper

information hiding. If your module needs to export symbols for other modules to

use, the following macros should be used.

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. Symbols

must be exported in the global part of the module’s file, outside of any function,

because the macros expand to the declaration of a special-purpose variable that

is expected to be accessible globally. This variable is stored in a special part of the

module executible (an “ELF section”)that is used by the kernel at load time to find

the variables exported by the module. (Interested readers can look at *<linux/module.h>*

for the details, even though the details are not needed to make things work.)

*Figure 2-2. Stacking of parallel port driver modules*

**Port sharing**

**and device**

**registration**

**Low-level**

**device**

**operations**

lp

parport

parport\_pc **Kernel API**

**(Message**

**printing, driver**

**registration,**

**port allocation,**

**etc.)**

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

30 | Chapter 2: Building and Running Modules

Preliminaries

We are getting closer to looking at some actual module code. But first, we need to

look at some other things that need to appear in your module source files. The kernel

is a unique environment, and it imposes its own requirements on code that

would interface with it.

Most kernel code ends up including a fairly large number of header files to get definitions

of functions, data types, and variables. We’ll examine these files as we come to

them, but there are a few that are specific to modules, and must appear in every loadable

module. Thus, just about all module code has the following:

#include <linux/module.h>

#include <linux/init.h>

*module.h* contains a great many definitions of symbols and functions needed by loadable

modules. You need *init.h* to specify your initialization and cleanup functions, as

we saw in the “hello world” example above, and which we revisit in the next section.

Most modules also include *moduleparam.h* to enable the passing of parameters

to the module at load time; we will get to that shortly.

It is not strictly necessary, but your module really should specify which license

applies to its code. Doing so is just a matter of including a MODULE\_LICENSE line:

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. As

we mentioned in the section “License Terms” in Chapter 1, kernel developers tend to

be unenthusiastic about helping users who experience problems after loading proprietary

modules.

Other descriptive definitions that can be contained within a module include

MODULE\_AUTHOR (stating who wrote the module), MODULE\_DESCRIPTION (a human-readable

statement of what the module does), MODULE\_VERSION (for a code revision number;

see the comments in *<linux/module.h>* for the conventions to use in creating

version strings), MODULE\_ALIAS (another name by which this module can be known),

and MODULE\_DEVICE\_TABLE (to tell user space about which devices the module supports).

We’ll discuss MODULE\_ALIAS in Chapter 11 and MODULE\_DEVICE\_TABLE in

Chapter 12.

The various MODULE\_ declarations can appear anywhere within your source file outside

of a function. A relatively recent convention in kernel code, however, is to put

these declarations at the end of the file.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Initialization and Shutdown | 31

Initialization and Shutdown

As already mentioned, the module initialization function registers any facility offered

by the module. By *facility*, we mean a new functionality, be it a whole driver or a new

software abstraction, that can be accessed by an application. The actual definition of

the initialization function always looks like:

static int \_\_init initialization\_function(void)

{

/\* Initialization code here \*/

}

module\_init(initialization\_function);

Initialization functions should be declared static, since they are not meant to be visible

outside the specific file; there is no hard rule about this, though, as no function is

exported to the rest of the kernel unless explicitly requested. The \_\_init token in the

definition may look a little strange; it is a hint to the kernel that the given function is

used only at initialization time. The module loader drops the initialization function

after the module is loaded, making its memory available for other uses. There is

a similar tag (\_\_initdata)for data used only during initialization. Use of \_\_init and

\_\_initdata is optional, but it is worth the trouble. Just be sure not to use them for

any function (or data structure)you will be using after initialization completes. You

may also encounter \_\_devinit and \_\_devinitdata in the kernel source; these translate

to \_\_init and \_\_initdata only if the kernel has not been configured for hotpluggable

devices. We will look at hotplug support in Chapter 14.

The use of *module\_init* is mandatory. This macro adds a special section to the module’s

object code stating where the module’s initialization function is to be found.

Without this definition, your initialization function is never called.

Modules can register many different types of facilities, including different kinds of

devices, filesystems, cryptographic transforms, and more. For each facility, there is a

specific kernel function that accomplishes this registration. The arguments passed to

the kernel registration functions are usually pointers to data structures describing the

new facility and the name of the facility being registered. The data structure usually

contains pointers to module functions, which is how functions in the module body

get called.

The items that can be registered go beyond the list of device types mentioned in

Chapter 1. They include, among others, serial ports, miscellaneous devices, sysfs

entries, */proc* files, executable domains, and line disciplines. Many of those registrable

items support functions that aren’t directly related to hardware but remain in the

“software abstractions” field. Those items can be registered, because they are integrated

into the driver’s functionality anyway (like */proc* files and line disciplines for

example).

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

32 | Chapter 2: Building and Running Modules

There are other facilities that can be registered as add-ons for certain drivers, but

their use is so specific that it’s not worth talking about them; they use the stacking

technique, as described in the section “The Kernel Symbol Table.” If you want to

probe further, you can grep for EXPORT\_SYMBOL in the kernel sources, and find the

entry points offered by different drivers. Most registration functions are prefixed with

register\_, so another possible way to find them is to grep for register\_ in the kernel

source.

The Cleanup Function

Every nontrivial module also requires a cleanup function, which unregisters interfaces

and returns all resources to the system before the module is removed. This

function is defined as:

static void \_\_exit cleanup\_function(void)

{

/\* Cleanup code here \*/

}

module\_exit(cleanup\_function);

The cleanup function has no value to return, so it is declared void. The \_\_exit modifier

marks the code as being for module unload only (by causing the compiler to

place it in a special ELF section). If your module is built directly into the kernel,

or if your kernel is configured to disallow the unloading of modules, functions

marked \_\_exit are simply discarded. For this reason, a function marked \_\_exit can

be called *only* at module unload or system shutdown time; any other use is an error.

Once again, the *module\_exit* declaration is necessary to enable to kernel to find your

cleanup function.

If your module does not define a cleanup function, the kernel does not allow it to be

unloaded.

Error Handling During Initialization

One thing you must always bear in mind when registering facilities with the kernel

is that the registration could fail. Even the simplest action often requires memory

allocation, and the required memory may not be available. So module code must

always check return values, and be sure that the requested operations have actually

succeeded.

If any errors occur when you register utilities, the first order of business is to decide

whether the module can continue initializing itself anyway. Often, the module can

continue to operate after a registration failure, with degraded functionality if necessary.

Whenever possible, your module should press forward and provide what capabilities

it can after things fail.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Initialization and Shutdown | 33

If it turns out that your module simply cannot load after a particular type of failure,

you must undo any registration activities performed before the failure. Linux doesn’t

keep a per-module registry of facilities that have been registered, so the module must

back out of everything itself if initialization fails at some point. If you ever fail to

unregister what you obtained, the kernel is left in an unstable state; it contains internal

pointers to code that no longer exists. In such situations, the only recourse, usually,

is to reboot the system. You really do want to take care to do the right thing

when an initialization error occurs.

Error recovery is sometimes best handled with the goto statement. We normally hate

to use goto, but in our opinion, this is one situation where it is useful. Careful use of

goto in error situations can eliminate a great deal of complicated, highly-indented,

“structured” logic. Thus, in the kernel, goto is often used as shown here to deal with

errors.

The following sample code (using fictitious registration and unregistration functions)

behaves correctly if initialization fails at any point:

int \_\_init my\_init\_function(void)

{

int err;

/\* registration takes a pointer and a name \*/

err = register\_this(ptr1, "skull");

if (err) goto fail\_this;

err = register\_that(ptr2, "skull");

if (err) goto fail\_that;

err = register\_those(ptr3, "skull");

if (err) goto fail\_those;

return 0; /\* success \*/

fail\_those: unregister\_that(ptr2, "skull");

fail\_that: unregister\_this(ptr1, "skull");

fail\_this: return err; /\* propagate the error \*/

}

This code attempts to register three (fictitious)facilities. The goto statement is used

in case of failure to cause the unregistration of only the facilities that had been successfully

registered before things went bad.

Another option, requiring no hairy goto statements, is keeping track of what has

been successfully registered and calling your module’s cleanup function in case of

any error. The cleanup function unrolls only the steps that have been successfully

accomplished. This alternative, however, requires more code and more CPU time, so

in fast paths you still resort to goto as the best error-recovery tool.

The return value of *my\_init\_function*, err, is an error code. In the Linux kernel, error

codes are negative numbers belonging to the set defined in *<linux/errno.h>*. If you

want to generate your own error codes instead of returning what you get from other

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

34 | Chapter 2: Building and Running Modules

functions, you should include *<linux/errno.h>* in order to use symbolic values such

as -ENODEV, -ENOMEM, and so on. It is always good practice to return appropriate error

codes, because user programs can turn them to meaningful strings using *perror* or

similar means.

Obviously, the module cleanup function must undo any registration performed by

the initialization function, and it is customary (but not usually mandatory)to unregister

facilities in the reverse order used to register them:

void \_\_exit my\_cleanup\_function(void)

{

unregister\_those(ptr3, "skull");

unregister\_that(ptr2, "skull");

unregister\_this(ptr1, "skull");

return;

}

If your initialization and cleanup are more complex than dealing with a few items,

the goto approach may become difficult to manage, because all the cleanup code

must be repeated within the initialization function, with several labels intermixed.

Sometimes, therefore, a different layout of the code proves more successful.

What you’d do to minimize code duplication and keep everything streamlined is to

call the cleanup function from within the initialization whenever an error occurs.

The cleanup function then must check the status of each item before undoing its registration.

In its simplest form, the code looks like the following:

struct something \*item1;

struct somethingelse \*item2;

int stuff\_ok;

void my\_cleanup(void)

{

if (item1)

release\_thing(item1);

if (item2)

release\_thing2(item2);

if (stuff\_ok)

unregister\_stuff( );

return;

}

int \_\_init my\_init(void)

{

int err = -ENOMEM;

item1 = allocate\_thing(arguments);

item2 = allocate\_thing2(arguments2);

if (!item2 || !item2)

goto fail;

err = register\_stuff(item1, item2);

if (!err)

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Module Parameters | 35

stuff\_ok = 1;

else

goto fail;

return 0; /\* success \*/

fail:

my\_cleanup( );

return err;

}

As shown in this code, you may or may not need external flags to mark success of the

initialization step, depending on the semantics of the registration/allocation function

you call. Whether or not flags are needed, this kind of initialization scales well to a

large number of items and is often better than the technique shown earlier. Note,

however, that the cleanup function cannot be marked \_\_exit when it is called by

nonexit code, as in the previous example.

Module-Loading Races

Thus far, our discussion has skated over an important aspect of module loading: race

conditions. If you are not careful in how you write your initialization function, you

can create situations that can compromise the stability of the system as a whole. We

will discuss race conditions later in this book; for now, a couple of quick points will

have to suffice.

The first is that you should always remember that some other part of the kernel can

make use of any facility you register immediately after that registration has completed.

It is entirely possible, in other words, that the kernel will make calls into your

module while your initialization function is still running. So your code must be prepared

to be called as soon as it completes its first registration. Do not register any

facility until all of your internal initialization needed to support that facility has been

completed.

You must also consider what happens if your initialization function decides to fail,

but some part of the kernel is already making use of a facility your module has registered.

If this situation is possible for your module, you should seriously consider not

failing the initialization at all. After all, the module has clearly succeeded in exporting

something useful. If initialization must fail, it must carefully step around any possible

operations going on elsewhere in the kernel until those operations have

completed.

Module Parameters

Several parameters that a driver needs to know can change from system to system.

These can vary from the device number to use (as we’ll see in the next chapter)to

numerous aspects of how the driver should operate. For example, drivers for SCSI

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

36 | Chapter 2: Building and Running Modules

adapters often have options controlling the use of tagged command queuing, and the

Integrated Device Electronics (IDE)drivers allow user control of DMA operations. If

your driver controls older hardware, it may also need to be told explicitly where to

find that hardware’s I/O ports or I/O memory addresses. The kernel supports these

needs by making it possible for a driver to designate parameters that may be changed

when the driver’s module is loaded.

These parameter values can be assigned at load time by *insmod* or *modprobe*; 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. Our vastly more functional module then, at load time, greets whom

not just once, but howmany times. Such a module could then be loaded with a command

line such as:

insmod hellop howmany=10 whom="Mom"

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

invbool

A boolean (true or false)value (the associated variable should be of type int).

The invbool type inverts the value, so that true values become false and vice

versa.

charp

A char pointer value. Memory is allocated for user-provided strings, and the

pointer is set accordingly.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Doing It in User Space | 37

int

long

short

uint

ulong

ushort

Basic integer values of various lengths. The versions starting with u are for

unsigned values.

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.

Doing It in User Space

A Unix programmer who’s addressing kernel issues for the first time might be nervous

about writing a module. Writing a user program that reads and writes directly

to the device ports may be easier.

Indeed, there are some arguments in favor of user-space programming, and sometimes

writing a so-called user-space device driver is a wise alternative to kernel hacking.

In this section, we discuss some of the reasons why you might write a driver in

\* As of this writing, there is talk of moving parameters elsewhere within sysfs, however.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

38 | Chapter 2: Building and Running Modules

user space. This book is about kernel-space drivers, however, so we do not go

beyond this introductory discussion.

The advantages of user-space drivers are:

• The full C library can be linked in. The driver can perform many exotic tasks

without resorting to external programs (the utility programs implementing usage

policies that are usually distributed along with the driver itself).

• The programmer can run a conventional debugger on the driver code without

having to go through contortions to debug a running kernel.

• If a user-space driver hangs, you can simply kill it. Problems with the driver are

unlikely to hang the entire system, unless the hardware being controlled is *really*

misbehaving.

• User memory is swappable, unlike kernel memory. An infrequently used device

with a huge driver won’t occupy RAM that other programs could be using,

except when it is actually in use.

• A well-designed driver program can still, like kernel-space drivers, allow concurrent

access to a device.

• If you must write a closed-source driver, the user-space option makes it easier for

you to avoid ambiguous licensing situations and problems with changing kernel

interfaces.

For example, USB drivers can be written for user space; see the (still young)libusb

project at libusb.sourceforge.net and “gadgetfs” in the kernel source. Another example

is the X server: it knows exactly what the hardware can do and what it can’t, and

it offers the graphic resources to all X clients. Note, however, that there is a slow but

steady drift toward frame-buffer-based graphics environments, where the X server

acts only as a server based on a real kernel-space device driver for actual graphic

manipulation.

Usually, the writer of a user-space driver implements a server process, taking over

from the kernel the task of being the single agent in charge of hardware control. Client

applications can then connect to the server to perform actual communication

with the device; therefore, a smart driver process can allow concurrent access to the

device. This is exactly how the X server works.

But the user-space approach to device driving has a number of drawbacks. The most

important are:

• Interrupts are not available in user space. There are workarounds for this limitation

on some platforms, such as the *vm86* system call on the IA32 architecture.

• Direct access to memory is possible only by *mmap*ping */dev/mem*, and only a

privileged user can do that.

• Access to I/O ports is available only after calling *ioperm* or *iopl*. Moreover, not

all platforms support these system calls, and access to */dev/port* can be too slow

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Quick Reference | 39

to be effective. Both the system calls and the device file are reserved to a privileged

user.

• Response time is slower, because a context switch is required to transfer information

or actions between the client and the hardware.

• Worse yet, if the driver has been swapped to disk, response time is unacceptably

long. Using the *mlock* system call might help, but usually you’ll need to lock

many memory pages, because a user-space program depends on a lot of library

code. *mlock*, too, is limited to privileged users.

• The most important devices can’t be handled in user space, including, but not

limited to, network interfaces and block devices.

As you see, user-space drivers can’t do that much after all. Interesting applications

nonetheless exist: for example, support for SCSI scanner devices (implemented by

the *SANE* package)and CD writers (implemented by *cdrecord* and other tools). In

both cases, user-level device drivers rely on the “SCSI generic” kernel driver, which

exports low-level SCSI functionality to user-space programs so they can drive their

own hardware.

One case in which working in user space might make sense is when you are beginning

to deal with new and unusual hardware. This way you can learn to manage your

hardware without the risk of hanging the whole system. Once you’ve done that,

encapsulating the software in a kernel module should be a painless operation.

Quick Reference

This section summarizes the kernel functions, variables, macros, and */proc* files that

we’ve touched on in this chapter. It is meant to act as a reference. Each item is listed

after the relevant header file, if any. A similar section appears at the end of almost

every chapter from here on, summarizing the new symbols introduced in the chapter.

Entries in this section generally appear in the same order in which they were

introduced in the chapter:

*insmod*

*modprobe*

*rmmod*

User-space utilities that load modules into the running kernels and remove

them.

#include <linux/init.h>

module\_init(init\_function);

module\_exit(cleanup\_function);

Macros that designate a module’s initialization and cleanup functions.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

40 | Chapter 2: Building and Running Modules

\_\_init

\_\_initdata

\_\_exit

\_\_exitdata

Markers for functions (\_\_init and \_\_exit)and data (\_\_initdata and \_\_exitdata)

that are only used at module initialization or cleanup time. Items marked for initialization

may be discarded once initialization completes; the exit items may be

discarded if module unloading has not been configured into the kernel. These

markers work by causing the relevant objects to be placed in a special ELF section

in the executable file.

#include <linux/sched.h>

One of the most important header files. This file contains definitions of much of

the kernel API used by the driver, including functions for sleeping and numerous

variable declarations.

struct task\_struct \*current;

The current process.

current->pid

current->comm

The process ID and command name for the current process.

obj-m

A makefile symbol used by the kernel build system to determine which modules

should be built in the current directory.

*/sys/module*

*/proc/modules*

*/sys/module* is a sysfs directory hierarchy containing information on currentlyloaded

modules. */proc/modules* is the older, single-file version of that information.

Entries contain the module name, the amount of memory each module

occupies, and the usage count. Extra strings are appended to each line to specify

flags that are currently active for the module.

*vermagic.o*

An object file from the kernel source directory that describes the environment a

module was built for.

#include <linux/module.h>

Required header. It must be included by a module source.

#include <linux/version.h>

A header file containing information on the version of the kernel being built.

LINUX\_VERSION\_CODE

Integer macro, useful to #ifdef version dependencies.

*This is the Title of the Book, eMatter Edition*

*Copyright © 2005 O’Reilly & Associates, Inc. All rights reserved.*

Quick Reference | 41

EXPORT\_SYMBOL (symbol);

EXPORT\_SYMBOL\_GPL (symbol);

Macro used to export a symbol to the kernel. The second form exports without

using versioning information, and the third limits the export to GPL-licensed

modules.

MODULE\_AUTHOR(author);

MODULE\_DESCRIPTION(description);

MODULE\_VERSION(version\_string);

MODULE\_DEVICE\_TABLE(table\_info);

MODULE\_ALIAS(alternate\_name);

Place documentation on the module in the object file.

module\_init(init\_function);

module\_exit(exit\_function);

Macros

# Writing Your First Linux Driver in the Classroom

This article, which is part of the [series on Linux device drivers](http://www.opensourceforu.com/tag/linux-device-drivers-series/), deals with the concept of dynamically loading drivers, first writing a Linux driver, before building and then loading it.

Shweta and Pugs reached their classroom late, to find their professor already in the middle of a lecture. Shweta sheepishly asked for his permission to enter. An annoyed Professor Gopi responded, “Come on! You guys are late again; what is your excuse, today?”

Pugs hurriedly replied that they had been discussing the very topic for that day’s class — device drivers in Linux. Pugs was more than happy when the professor said, “Good! Then explain about dynamic loading in Linux. If you get it right, the two of you are excused!” Pugs knew that one way to make his professor happy was to criticise Windows.

He explained, “As we know, a typical driver installation on Windows needs a reboot for it to get activated. That is really not acceptable; suppose we need to do it on a server? That’s where Linux wins. In Linux, we can load or unload a driver on the fly, and it is active for use instantly after loading. Also, it is instantly disabled when unloaded. This is called dynamic loading and unloading of drivers in Linux.”

This impressed the professor. “Okay! Take your seats, but make sure you are not late again.” The professor continued to the class, “Now you already know what is meant by dynamic loading and unloading of drivers, so I’ll show you how to do it, before we move on to write our first Linux driver.”

## Dynamically loading drivers

These dynamically loadable drivers are more commonly called modules and built into individual files with a .ko(kernel object) extension. Every Linux system has a standard place under the root of the file system (/) for all the pre-built modules. They are organised similar to the kernel source tree structure, under /lib/modules/<kernel\_version>/kernel, where <kernel\_version> would be the output of the command uname -ron the system, as shown in Figure 1.

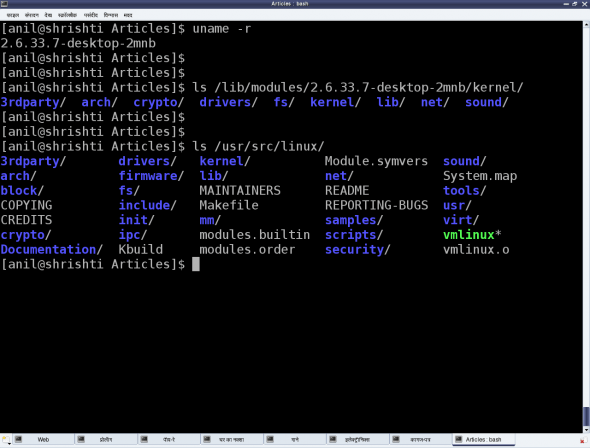
[](http://www.opensourceforu.com/wp-content/uploads/2010/12/figure_4_linux_modules.png)

Figure 1: Linux pre-built modules

To dynamically load or unload a driver, use these commands, which reside in the /sbin directory, and must be executed with root privileges:

* lsmod — lists currently loaded modules
* insmod <module\_file> — inserts/loads the specified module file
* modprobe <module> — inserts/loads the module, along with any dependencies
* rmmod <module> — removes/unloads the module

Let’s look at the FAT filesystem-related drivers as an example. Figure 2 demonstrates this complete process of experimentation. The module files would be fat.ko, vfat.ko, etc., in the fat (vfat for older kernels) directory under /lib/modules/`uname -r`/kernel/fs. If they are in compressed .gz format, you need to uncompress them with gunzip, before you can insmodthem.

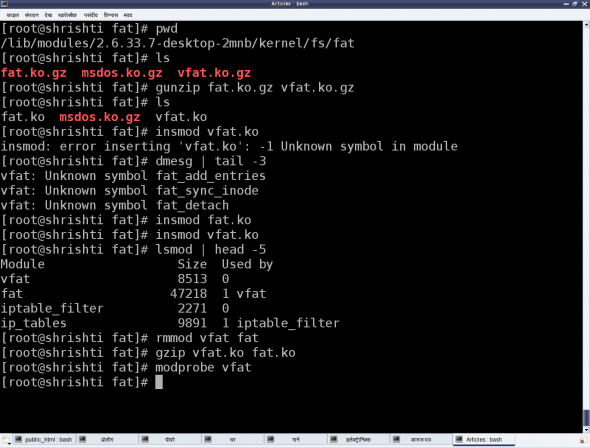
[](http://www.opensourceforu.com/wp-content/uploads/2010/12/figure_5_linux_module_operations.png)

Figure 2: Linux module operations

The vfat module depends on the fat module, so fat.ko needs to be loaded first. To automatically perform decompression and dependency loading, use modprobe instead. Note that you shouldn’t specify the .ko extension to the module’s name, when using the modprobe command. rmmod is used to unload the modules.

## Our first Linux driver

Before we write our first driver, let’s go over some concepts. A driver never runs by itself. It is similar to a library that is loaded for its functions to be invoked by a running application. It is written in C, but lacks a main() function. Moreover, it will be loaded/linked with the kernel, so it needs to be compiled in a similar way to the kernel, and the header files you can use are only those from the kernel sources, not from the standard /usr/include.

One interesting fact about the kernel is that it is an object-oriented implementation in C, as we will observe even with our first driver. Any Linux driver has a constructor and a destructor. The module’s constructor is called when the module is successfully loaded into the kernel, and the destructor when rmmod succeeds in unloading the module. These two are like normal functions in the driver, except that they are specified as the init and exitfunctions, respectively, by the macros module\_init() and module\_exit(), which are defined in the kernel header module.h.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | /\* ofd.c – Our First Driver code \*/  #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>    static int \_\_init ofd\_init(void) /\* Constructor \*/  {      printk(KERN\_INFO "Namaskar: ofd registered");      return 0;  }    static void \_\_exit ofd\_exit(void) /\* Destructor \*/  {      printk(KERN\_INFO "Alvida: ofd unregistered");  }    module\_init(ofd\_init);  module\_exit(ofd\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Our First Driver"); |

Given above is the complete code for our first driver; let’s call it ofd.c. Note that there is no stdio.h (a user-space header); instead, we use the analogous kernel.h (a kernel space header). printk() is the equivalent of printf(). Additionally, version.h is included for the module version to be compatible with the kernel into which it is going to be loaded. The MODULE\_\* macros populate module-related information, which acts like the module’s “signature”.

## Building our first Linux driver

Once we have the C code, it is time to compile it and create the module file ofd.ko. We use the kernel build system to do this. The following Makefile invokes the kernel’s build system from the kernel source, and the kernel’s Makefile will, in turn, invoke our first driver’s Makefile to build our first driver.

To build a Linux driver, you need to have the kernel source (or, at least, the kernel headers) installed on your system. The kernel source is assumed to be installed at /usr/src/linux. If it’s at any other location on your system, specify the location in the KERNEL\_SOURCE variable in this Makefile.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # Makefile – makefile of our first driver    # if KERNELRELEASE is defined, we've been invoked from the  # kernel build system and can use its language.  ifneq (${KERNELRELEASE},)      obj-m := ofd.o  # Otherwise we were called directly from the command line.  # Invoke the kernel build system.  else      KERNEL\_SOURCE := /usr/src/linux      PWD := $(shell pwd)  default:      ${MAKE} -C ${KERNEL\_SOURCE} SUBDIRS=${PWD} modules    clean:      ${MAKE} -C ${KERNEL\_SOURCE} SUBDIRS=${PWD} clean  endif |

With the C code (ofd.c) and Makefile ready, all we need to do is invoke make to build our first driver (ofd.ko).

|  |
| --- |
| $ make  make -C /usr/src/linux SUBDIRS=... modules  make[1]: Entering directory `/usr/src/linux'    CC [M]  .../ofd.o    Building modules, stage 2.    MODPOST 1 modules    CC      .../ofd.mod.o    LD [M]  .../ofd.ko  make[1]: Leaving directory `/usr/src/linux' |

## Summing up

Once we have the ofd.ko file, perform the usual steps as the root user, or with sudo.

|  |
| --- |
| # su  # insmod ofd.ko  # lsmod | head -10 |

lsmod should show you the ofd driver loaded.

While the students were trying their first module, the bell rang, marking the end of the session. Professor Gopi concluded, “Currently, you may not be able to observe anything other than the lsmod listing showing the driver has loaded. Where’s the printk output gone? Find that out for yourselves, in the lab session, and update me with your findings. Also note that our first driver is a template for any driver you would write in Linux. Writing a specialised driver is just a matter of what gets filled into its constructor and destructor. So, our further learning will be to enhance this driver to achieve specific driver functionalities.”

# Kernel C Extras in a Linux Driver

This article in the [series on Linux device drivers](http://www.opensourceforu.com/tag/linux-device-drivers-series/) deals with the kernel’s message logging, and kernel-specific GCC extensions.

Enthused by how Pugs impressed their professor in the last class, Shweta wanted to do so too. And there was soon an opportunity: finding out where the output of printk had gone. So, as soon as she entered the lab, she grabbed the best system, logged in, and began work. Knowing her professor well, she realised that he would have dropped a hint about the possible solution in the previous class itself. Going over what had been taught, she remembered the error output demonstration from insmod vfat.ko — running dmesg | tail. She immediately tried that, and found the printk output there.

But how did it come to be here? A tap on her shoulder roused her from her thoughts. “Shall we go for a coffee?” proposed Pugs.

“But I need to –“.

“I know what you’re thinking about,” interrupted Pugs. “Let’s go, I’ll explain you all about dmesg.”

## Kernel message logging

Over coffee, Pugs began his explanation.

As far as parameters are concerned, printf and printk are the same, except that when programming for the kernel, we don’t bother about the float formats %f, %lf and the like. However, unlike printf, printk is not designed to dump its output to some console.

In fact, it cannot do so; it is something in the background, and executes like a library, only when triggered either from hardware-space or user-space. All printk calls put their output into the (log) ring buffer of the kernel. Then, the syslog daemon running in user-space picks them up for final processing and redirection to various devices, as configured in the configuration file /etc/syslog.conf.

You must have observed the out-of-place macro KERN\_INFO, in the printk calls, in the [last article](http://www.opensourceforu.com/2010/12/writing-your-first-linux-driver/). That is actually a constant string, which gets concatenated with the format string after it, into a single string. Note that there is no comma (,) between them; they are not two separate arguments. There are eight such macros defined in linux/kernel.h in the kernel source, namely:

|  |
| --- |
| #define KERN\_EMERG "<0>"   /\* system is unusable                \*/  #define KERN\_ALERT "<1>"   /\* action must be taken immediately    \*/  #define KERN\_CRIT "<2>"    /\* critical conditions     \*/  #define KERN\_ERR "<3>"     /\* error conditions            \*/  #define KERN\_WARNING "<4>" /\* warning conditions      \*/  #define KERN\_NOTICE "<5>"  /\* normal but significant condition    \*/  #define KERN\_INFO "<6>"    /\* informational           \*/  #define KERN\_DEBUG "<7>"   /\* debug-level messages        \*/ |

Now depending on these log levels (i.e., the first three characters in the format string), the syslog user-space daemon redirects the corresponding messages to their configured locations. A typical destination is the log file /var/log/messages, for all log levels. Hence, all the printk outputs are, by default, in that file. However, they can be configured differently — to a serial port (like /dev/ttyS0), for instance, or to all consoles, like what typically happens for KERN\_EMERG.

Now, /var/log/messages is buffered, and contains messages not only from the kernel, but also from various daemons running in user-space. Moreover, this file is often not readable by a normal user. Hence, a user-space utility, dmesg, is provided to directly parse the kernel ring buffer, and dump it to standard output. Figure 1 shows snippets from the two.

Figure 1: Kernel’s message logging

## Kernel-specific GCC extensions

Shweta, frustrated since she could no longer show off as having discovered all these on her own, retorted, “Since you have explained all about printing in the kernel, why don’t you also tell me about the weird C in the driver as well — the special keywords \_\_init, \_\_exit, etc.”

These are not special keywords. Kernel C is not “weird C”, but just standard C with some additional extensions from the C compiler, GCC. Macros \_\_init and \_\_exit are just two of these extensions. However, these do not have any relevance in case we are using them for a dynamically loadable driver, but only when the same code gets built into the kernel. All functions marked with \_\_init get placed inside the init section of the kernel image automatically, by GCC, during kernel compilation; and all functions marked with \_\_exit are placed in the exit section of the kernel image.

What is the benefit of this? All functions with \_\_init are supposed to be executed only once during bootup (and not executed again till the next bootup). So, once they are executed during bootup, the kernel frees up RAM by removing them (by freeing the init section). Similarly, all functions in the exit section are supposed to be called during system shutdown.

Now, if the system is shutting down anyway, why do you need to do any cleaning up? Hence, the exit section is not even loaded into the kernel — another cool optimisation. This is a beautiful example of how the kernel and GCC work hand-in-hand to achieve a lot of optimisation, and many other tricks that we will see as we go along. And that is why the Linux kernel can only be compiled using GCC-based compilers — a closely knit bond.

## The kernel function’s return guidelines

While returning from coffee, Pugs kept praising OSS and the community that’s grown around it. Do you know why different individuals are able to come together and contribute excellently without any conflicts, and in a project as huge as Linux, at that? There are many reasons, but most important amongst them is that they all follow and abide by inherent coding guidelines.

Take, for example, the kernel programming guideline for returning values from a function. Any kernel function needing error handling, typically returns an integer-like type — and the return value again follows a guideline. For an error, we return a negative number: a minus sign appended with a macro that is available through the kernel header linux/errno.h, that includes the various error number headers under the kernel sources — namely, asm/errno.h, asm-generic/errno.h, asm-generic/errno-base.h.

For success, zero is the most common return value, unless there is some additional information to be provided. In that case, a positive value is returned, the value indicating the information, such as the number of bytes transferred by the function.

## Kernel C = pure C

Once back in the lab, Shweta remembered their professor mentioning that no /usr/include headers can be used for kernel programming. But Pugs had said that kernel C is just standard C with some GCC extensions. Why this conflict?

Actually this is not a conflict. Standard C is pure C — just the language. The headers are not part of it. Those are part of the standard libraries built in for C programmers, based on the concept of reusing code.

Does that mean that all standard libraries, and hence, all ANSI standard functions, are not part of “pure” C? Yes, that’s right. Then, was it really tough coding the kernel?

Well, not for this reason. In reality, kernel developers have evolved their own set of required functions, which are all part of the kernel code. The printk function is just one of them. Similarly, many string functions, memory functions, and more, are all part of the kernel source, under various directories like kernel, ipc, lib, and so on, along with the corresponding headers under the include/linux directory.

“Oh yes! That is why we need to have the kernel source to build a driver,” agreed Shweta.

“If not the complete source, at least the headers are a must. And that is why we have separate packages to install the complete kernel source, or just the kernel headers,” added Pugs.

“In the lab, all the sources are set up. But if I want to try out drivers on my Linux system in my hostel room, how do I go about it?” asked Shweta.

“Our lab has Fedora, where the kernel sources are typically installed under /usr/src/kernels/<kernel-version>, unlike the standard /usr/src/linux. Lab administrators must have installed it using the command-line yum install kernel-devel. I use Mandriva, and installed the kernel sources using urpmi kernel-source,” replied Pugs.

“But I have Ubuntu,” Shweta said.

“Okay! For that, just use apt-get utility to fetch the source — possibly apt-get install linux-source,” replied Pugs.

## Summing up

The lab session was almost over when Shweta suddenly asked, out of curiosity, “Hey Pugs, what’s the next topic we are going to learn in our Linux device drivers class?”

“Hmm… most probably character drivers,” threw back Pugs.

With this information, Shweta hurriedly packed her bag and headed towards her room to set up the kernel sources, and try out the next driver on her own. “In case you get stuck, just give me a call,” smiled Pugs

# Module Interactions

As Shweta and Pugs gear up for their final semester’s project on Linux drivers, they’re closing in on some final titbits of technical romancing. This mainly includes the various communications with a Linux module (dynamically loadable and unloadable driver) like accessing its variables, calling its functions, and passing parameters to it.

## Global variables and functions

One might wonder what the big deal is about accessing a module’s variables and functions from outside it. Just make them global, declare them extern in a header, include the header and access, right? In the general application development paradigm, it’s this simple — but in kernel development, it isn’t despite of recommendations to make everything static, by default there always have been cases where non-static globals may be needed.

A simple example could be a driver spanning multiple files, with function(s) from one file needing to be called in the other. Now, to avoid any kernel name-space collisions even with such cases, every module is embodied in its own namespace. And we know that two modules with the same name cannot be loaded at the same time. Thus, by default, zero collision is achieved. However, this also implies that, by default, nothing from a module can be made really global throughout the kernel, even if we want to. And so, for exactly such scenarios, the <linux/module.h>header defines the  
following macros:

* EXPORT\_SYMBOL(sym)
* EXPORT\_SYMBOL\_GPL(sym)
* EXPORT\_SYMBOL\_GPL\_FUTURE(sym)

Each of these exports the symbol passed as their parameter, additionally putting them in one of the default, \_gpl or \_gpl\_future sections, respectively. Hence, only one of them needs to be used for a particular symbol — though the symbol could be either a variable name or a function name. Here’s the complete code (our\_glob\_syms.c) to demonstrate this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include <linux/module.h>  #include <linux/device.h>    static struct class \*cool\_cl;  static struct class \*get\_cool\_cl(void)  {      return cool\_cl;  }  EXPORT\_SYMBOL(cool\_cl);  EXPORT\_SYMBOL\_GPL(get\_cool\_cl);    static int \_\_init glob\_sym\_init(void)  {      if (IS\_ERR(cool\_cl = class\_create(THIS\_MODULE, "cool")))      /\* Creates /sys/class/cool/ \*/      {          return PTR\_ERR(cool\_cl);      }      return 0;  }    static void \_\_exit glob\_sym\_exit(void)  {      /\* Removes /sys/class/cool/ \*/      class\_destroy(cool\_cl);  }    module\_init(glob\_sym\_init);  module\_exit(glob\_sym\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs.com>");  MODULE\_DESCRIPTION("Global Symbols exporting Driver"); |

Each exported symbol also has a corresponding structure placed into (each of) the kernel symbol table (\_\_ksymtab), kernel string table (\_\_kstrtab), and kernel CRC table (\_\_kcrctab) sections, marking it to be globally accessible.

Figure 1 shows a filtered snippet of the /proc/kallsyms kernel window, before and after loading the module our\_glob\_syms.ko, which has been compiled using the usual driver Makefile.

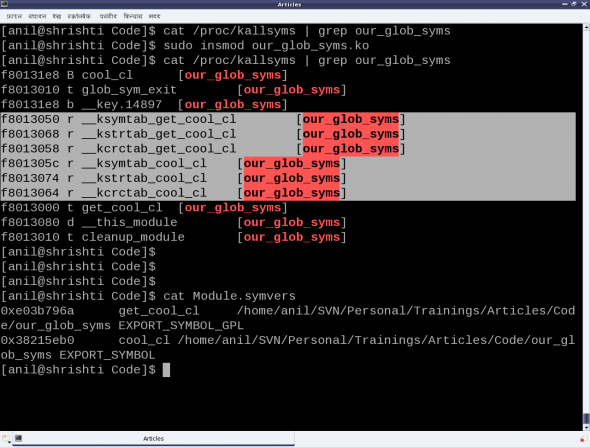
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_30_our_glob_syms.png)

Figure 1: Our global symbols module

The following code shows the supporting header file (our\_glob\_syms.h), to be included by modules using the exported symbols cool\_cl and get\_cool\_cl:

|  |
| --- |
| #ifndef OUR\_GLOB\_SYMS\_H  #define OUR\_GLOB\_SYMS\_H    #ifdef \_\_KERNEL\_\_  #include <linux/device.h>    extern struct class \*cool\_cl;  extern struct class \*get\_cool\_cl(void);  #endif    #endif |

Figure 1 also shows the file Module.symvers, generated by compiling the module our\_glob\_syms. This contains the various details of all the exported symbols in its directory. Apart from including the above header file, modules using the exported symbols should possibly have this file Module.symvers in their build directory.

Note that the <linux/device.h> header in the above examples is being included for the various class-related declarations and definitions, which have already been covered in the earlier discussion on character drivers.

## Module parameters

Being aware of passing command-line arguments to an application, it would be natural to ask if something similar can be done with a module — and the answer is, yes, it can. Parameters can be passed to a module while loading it, for instance, when using insmod. Interestingly enough, and in contrast to the command-line arguments to an application, these can be modified even later, through sysfs interactions.

The module parameters are set up using the following macro (defined in <linux/moduleparam.h>, included through <linux/module.h>):

|  |
| --- |
| module\_param(name, type, perm) |

Here, name is the parameter name, type is the type of the parameter, and perm refers to the permissions of the sysfs file corresponding to this parameter. The supported type values are: byte, short, ushort, int, uint, long, ulong, charp (character pointer), bool or invbool (inverted Boolean).

The following module code (module\_param.c) demonstrates a module parameter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <linux/module.h>  #include <linux/kernel.h>    static int cfg\_value = 3;  module\_param(cfg\_value, int, 0764);    static int \_\_init mod\_par\_init(void)  {      printk(KERN\_INFO "Loaded with %d\n", cfg\_value);      return 0;  }    static void \_\_exit mod\_par\_exit(void)  {      printk(KERN\_INFO "Unloaded cfg value: %d\n", cfg\_value);  }    module\_init(mod\_par\_init);  module\_exit(mod\_par\_exit);  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email@sarika-pugs.com>");  MODULE\_DESCRIPTION("Module Parameter demonstration Driver"); |

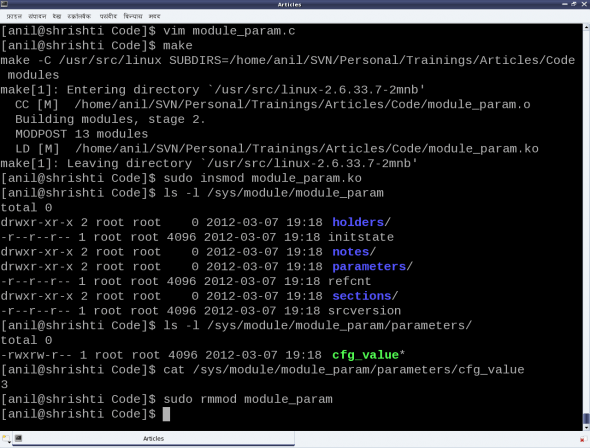
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_31_module_param.png)

Figure 2: Experiments with the module parameter

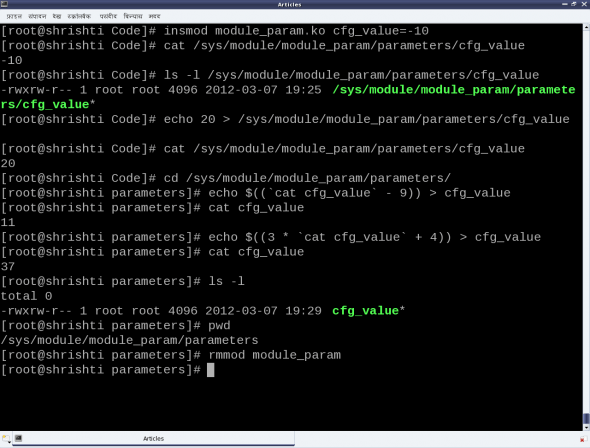
[](http://www.opensourceforu.com/wp-content/uploads/2012/05/figure_32_module_param_as_root.png)

Figure 3: Experiments with the module parameter (as root)

Note that before the parameter setup, a variable of the same name and compatible type needs to be defined. Subsequently, the following steps and experiments are shown in Figures 2 and 3:

* Building the driver (module\_param.ko file) using the usual driver Makefile
* Loading the driver using insmod (with and without parameters)
* Various experiments through the corresponding /sys entries
* And finally, unloading the driver using rmmod.

Note the following:

* Initial value (3) of cfg\_value becomes its default value when insmod is done without any parameters.
* Permission 0764 gives rwx to the user, rw- to the group, and r-- for the others on the file cfg\_valueunder the parameters of module\_param under /sys/module/.

Check for yourself:

* The output of dmesg/tail on every insmod and rmmod, for the printk outputs.
* Try writing into the /sys/module/module\_param/parameters/cfg\_value file as a normal (non-root) user.

## Summing up

With this, the duo have a fairly good understanding of Linux drivers, and are all set to start working on their final semester project. Any guesses what their project is about? Hint: They have picked up one of the most daunting Linux driver topics. Let us see how they fare with it next month.

# Kernel Window — Peeping through /proc

After many months, Shweta and Pugs got together for some peaceful technical romancing. All through, they had been using all kinds of kernel windows, especially through the /proc virtual filesystem (using cat), to help them decode various details of Linux device drivers. Here’s a non-exhaustive summary listing:

* /proc/modules — dynamically loaded modules
* /proc/devices — registered character and block major numbers
* /proc/iomem — on-system physical RAM and bus device addresses
* /proc/ioports — on-system I/O port addresses (especially for x86 systems)
* /proc/interrupts — registered interrupt request numbers
* /proc/softirqs — registered soft IRQs
* /proc/kallsyms — running kernel symbols, including from loaded modules
* /proc/partitions — currently connected block devices and their partitions
* /proc/filesystems — currently active filesystem drivers
* /proc/swaps — currently active swaps
* /proc/cpuinfo — information about the CPU(s) on the system
* /proc/meminfo — information about the memory on the system, viz., RAM, swap, …

## Custom kernel windows

“Yes, these have been really helpful in understanding and debugging Linux device drivers. But is it possible for us to also provide some help? Yes, I mean can we create one such kernel window through /proc?” asked Shweta.

“Why just one? You can have as many as you want. And it’s simple — just use the right set of APIs, and there you go.”

“For you, everything is simple,” Shweta grumbled.

“No yaar, this is seriously simple,” smiled Pugs. “Just watch me creating one for you,” he added.  
And in a jiffy, Pugs created the proc\_window.c file below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/proc\_fs.h>  #include <linux/jiffies.h>    static struct proc\_dir\_entry \*parent, \*file, \*link;  static int state = 0;    int time\_read(char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data) {      int len, val;      unsigned long act\_jiffies;        len = sprintf(page, "state = %d\n", state);      act\_jiffies = jiffies - INITIAL\_JIFFIES;      val = jiffies\_to\_msecs(act\_jiffies);      switch (state) {          case 0:              len += sprintf(page + len, "time = %ld jiffies\n", act\_jiffies);              break;          case 1:              len += sprintf(page + len, "time = %d msecs\n", val);              break;          case 2:              len += sprintf(page + len, "time = %ds %dms\n",                      val / 1000, val % 1000);              break;          case 3:              val /= 1000;              len += sprintf(page + len, "time = %02d:%02d:%02d\n",                      val / 3600, (val / 60) % 60, val % 60);              break;          default:              len += sprintf(page + len, "<not implemented>\n");              break;      }      len += sprintf(page + len, "{offset = %ld; count = %d;}\n", off, count);        return len;  }  int time\_write(struct file \*file, const char \_\_user \*buffer, unsigned long count, void \*data) {      if (count > 2)          return count;      if ((count == 2) && (buffer[1] != '\n'))          return count;      if ((buffer[0] < '0') || ('9' < buffer[0]))          return count;      state = buffer[0] - '0';      return count;  }    static int \_\_init proc\_win\_init(void) {      if ((parent = proc\_mkdir("anil", NULL)) == NULL) {          return -1;      }      if ((file = create\_proc\_entry("rel\_time", 0666, parent)) == NULL) {          remove\_proc\_entry("anil", NULL);          return -1;      }      file->read\_proc = time\_read;      file->write\_proc = time\_write;      if ((link = proc\_symlink("rel\_time\_l", parent, "rel\_time")) == NULL) {          remove\_proc\_entry("rel\_time", parent);          remove\_proc\_entry("anil", NULL);          return -1;      }      link->uid = 0;      link->gid = 100;      return 0;  }    static void \_\_exit proc\_win\_exit(void) {      remove\_proc\_entry("rel\_time\_l", parent);      remove\_proc\_entry("rel\_time", parent);      remove\_proc\_entry("anil", NULL);  }    module\_init(proc\_win\_init);  module\_exit(proc\_win\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Kernel window /proc Demonstration Driver"); |

And then Pugs did the following:

* Built the driver file (proc\_window.ko) using the usual driver’s Makefile.
* Loaded the driver using insmod.
* Showed various experiments using the newly created proc windows. (Refer to Figure 1.)
* And finally, unloaded the driver using rmmod.

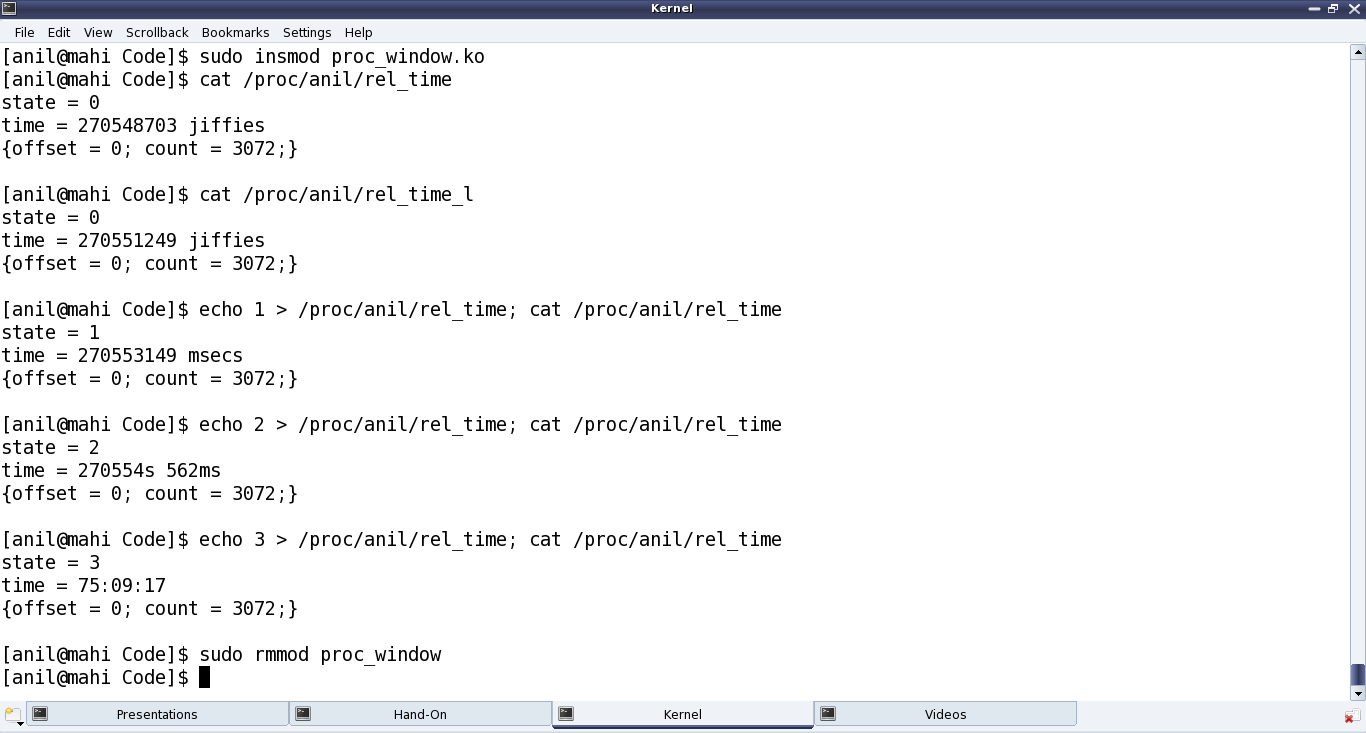


Figure 1: Peeping through /proc

## Demystifying the details

Starting from the constructor proc\_win\_init(), three proc entries have been created:

* Directory anil under /proc (i.e., NULL parent) with default permissions 0755, using proc\_mkdir()
* Regular file rel\_time in the above directory, with permissions 0666, using create\_proc\_entry()
* Soft link rel\_time\_l to the file rel\_time, in the same directory, using proc\_symlink()

The corresponding removal of these is done with remove\_proc\_entry() in the destructor, proc\_win\_exit(), in chronological reverse order.

For every entry created under /proc, a corresponding struct proc\_dir\_entry is created. For each, many of its fields could be further updated as needed:

* mode — Permissions of the file
* uid — User ID of the file
* gid — Group ID of the file

Additionally, for a regular file, the following two function pointers for reading and writing over the file could be provided, respectively:

* int (\*read\_proc)(char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data)
* int (\*write\_proc)(struct file \*file, const char \_\_user \*buffer, unsigned long count, void \*data)

write\_proc() is very similar to the character driver’s file operation write(). The above implementation lets the user write a digit from 0 to 9, and accordingly sets the internal state. read\_proc() in the above implementation provides the current state, and the time since the system has been booted up — in different units, based on the current state. These are jiffies in state 0; milliseconds in state 1; seconds and milliseconds in state 2; hours, minutes and seconds in state 3; and <not implemented> in other states.

And to check the computation accuracy, Figure 2 highlights the system uptime in the output of top. read\_proc‘s page parameter is a page-sized buffer, typically to be filled up with count bytes from offset off. But more often than not (because of less content), just the page is filled up, ignoring all other parameters.

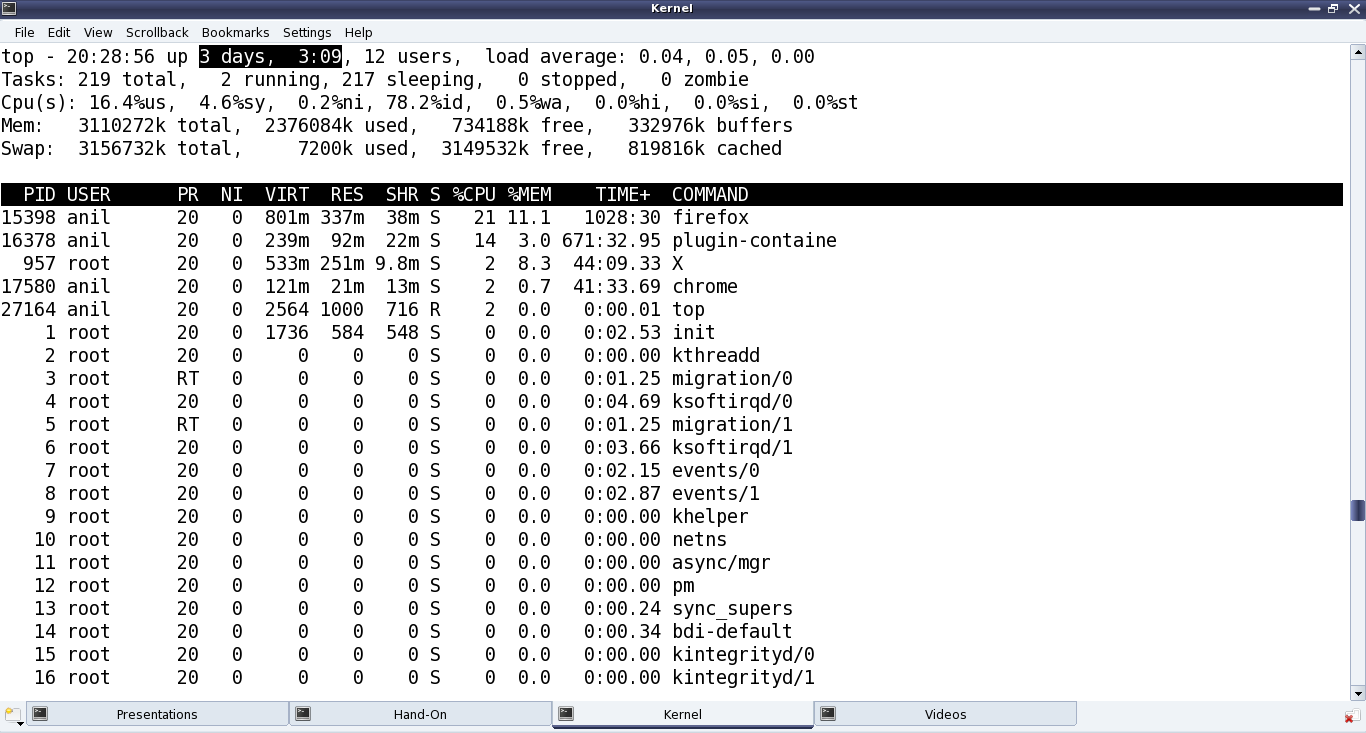


Figure 2: Comparison with top’s output

All the /proc-related structure definitions and function declarations are available through <linux/proc\_fs.h>. The jiffies-related function declarations and macro definitions are in <linux/jiffies.h>. As a special note, the actual jiffies are calculated by subtracting INITIAL\_JIFFIES, since on boot-up, jiffies is initialised to INITIAL\_JIFFIES instead of zero.

## Summing up

“Hey Pugs! Why did you set the folder name to anil? Who is this Anil? You could have used my name, or maybe yours,” suggested Shweta. “Ha! That’s a surprise. My real name is Anil; it’s just that everyone in college knows me as Pugs,” smiled Pugs.

# Kernel-Space Debuggers in Linux

Shweta, back from hospital, was relaxing in the library, reading various books. Ever since she learned of the ioctlway of debugging, she was impatient to find out more about debugging in kernel-space. She was curious about how and where to run the kernel-space debugger, if there was any. This was in contrast with application/user-space debugging, where we have the OS running underneath, and a shell or a GUI over it to run the debugger (like gdb, and the data display debugger, ddd). Then she came across this interesting kernel-space debugging mechanism using kgdb, provided as part of the kernel itself, since kernel 2.6.26.

## The debugger challenge in kernel-space

As we need some interface to be up to run a debugger to debug anything, a kernel debugger could be visualised in two possible ways:

* Put the debugger into the kernel itself, accessible via the usual console. For example, in the case of kdb, which was not official until kernel 2.6.35, one had to download source code (two sets of patches — one architecture-dependent, one architecture-independent) from [this FTP address](ftp://oss.sgi.com/projects/kdb/download/) and then patch these into the kernel source. However, since kernel 2.6.35, the majority of it is in the officially released kernel source. In either case, kdb support needs to be enabled in kernel source, with the kernel compiled, installed and booted with. The boot screen itself would give the kdb debugging interface.
* Put a minimal debugging server into the kernel; a client would connect to it from a remote host or local user-space over some interface (say serial or Ethernet). This is kgdb, the kernel’s gdb server, to be used with gdbas its client. Since kernel 2.6.26, its serial interface is part of the official kernel release. However, if you’re interested in a network interface, you still need to patch with one of the releases from the [kgdb project page](http://sourceforge.net/projects/kgdb/). In either case, you need to enable kgdb support in the kernel, recompile, install and boot the new kernel.

Please note that in both the above cases, the complete kernel source for the kernel to be debugged is needed, unlike for building modules, where just headers are sufficient. Here is how to play around with kgdb over the serial interface.

## Setting up the Linux kernel with kgdb

Here are the prerequisites: Either the kernel source package for the running kernel should be installed on your system, or a corresponding kernel source release should have been downloaded from [kernel.org](http://kernel.org/).

First of all, the kernel to be debugged needs to have kgdb enabled and built into it. To achieve that, the kernel source has to be configured with CONFIG\_KGDB=y. Additionally, for kgdb over serial, CONFIG\_KGDB\_SERIAL\_CONSOLE=y needs to be configured. And CONFIG\_DEBUG\_INFO is preferred for symbolic data to be built into the kernel, to make debugging with gdb more meaningful. CONFIG\_FRAME\_POINTER=y enables frame pointers in the kernel, allowing gdb to construct more accurate stack back-traces. All these options are available under “Kernel hacking” in the menu obtained in the kernel source directory (preferably as root, or using sudo), by issuing the following command:

|  |
| --- |
| $ make mrproper      # To clean up properly  $ make oldconfig     # Configure the kernel same as the current running one  $ make menuconfig    # Start the ncurses based menu for further configuration |

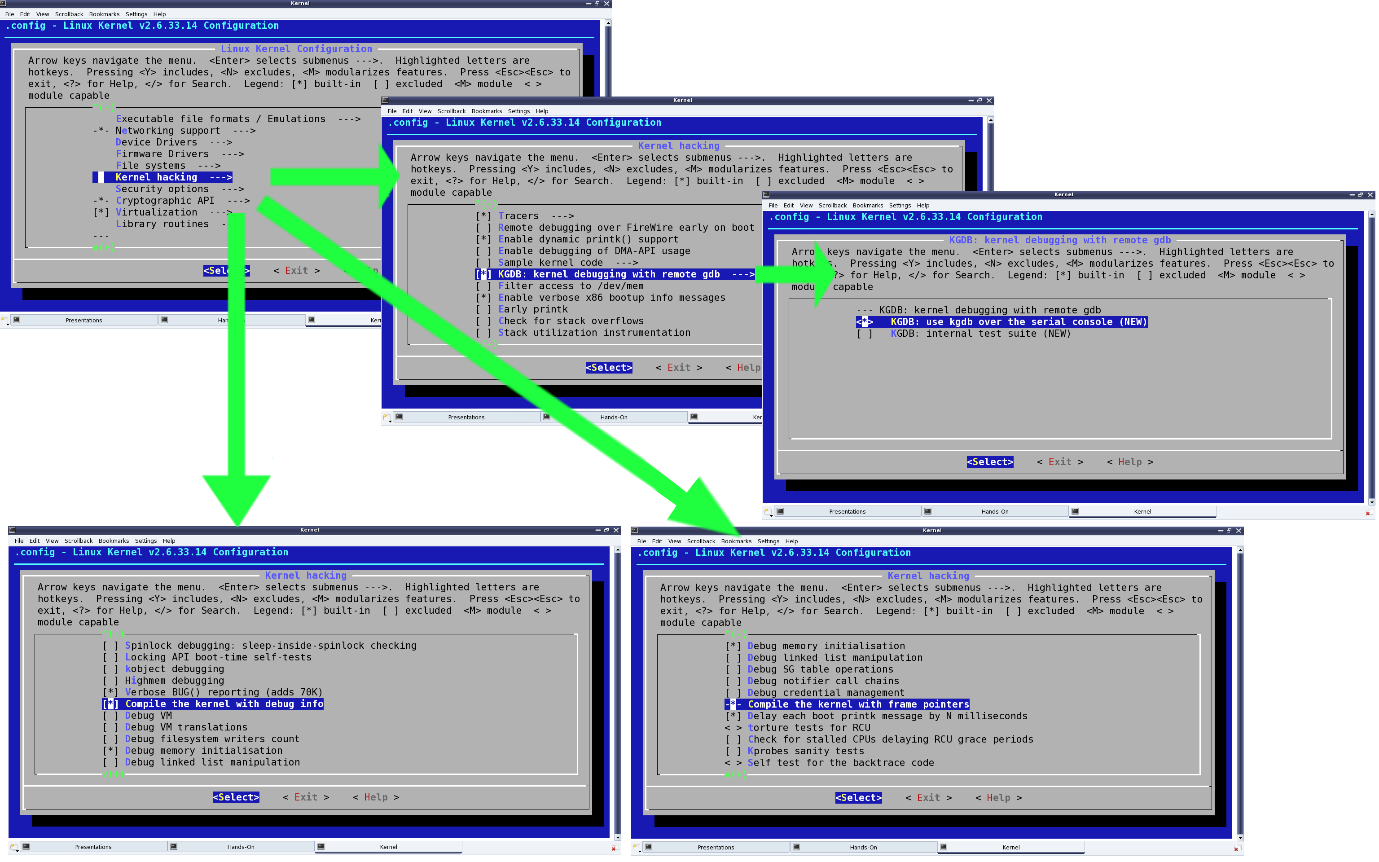


Figure 1: Configuring kernel options for kgdb

See the highlighted selections in Figure 1, for how and where these options would be:

* “KGDB: kernel debugging with remote gdb” –> CONFIG\_KGDB
* “KGDB: use kgdb over the serial console” –> CONFIG\_KGDB\_SERIAL\_CONSOLE
* “Compile the kernel with debug info” –> CONFIG\_DEBUG\_INFO
* “Compile the kernel with frame pointers” –> CONFIG\_FRAME\_POINTER

Once configuration is saved, build the kernel (run make), and then a make install to install it, along with adding an entry for the installed kernel in the GRUB configuration file. Depending on the distribution, the GRUB configuration file may be /boot/grub/menu.lst, /etc/grub.cfg, or something similar. Once installed, the kgdb-related kernel boot parameters need to be added to this new entry, as shown in the highlighted text in Figure 2.



Figure 2: GRUB configuration for kgdb

kgdboc is for gdb connecting over the console, and the basic format is kgdboc= <serial\_device>, <baud-rate>where:

* <serial\_device> is the serial device file (port) on the system running the kernel to be debugged
* <baud-rate> is the baud rate of this serial port

kgdbwait tells the kernel to delay booting till a gdb client connects to it; this parameter should be given only after kgdboc.

With this, we’re ready to begin. Make a copy of the vmlinux kernel image for use on the gdb client system. Reboot, and at the GRUB menu, choose the new kernel, and then it will wait for gdb to connect over the serial port.

All the above snapshots are with kernel version 2.6.33.14. The same should work for any 2.6.3x release of the kernel source. Also, the snapshots for kgdb are captured over the serial device file /dev/ttyS0, i.e., the first serial port.

## Setting up gdb on another system

Following are the prerequisites:

* Serial ports of the system to be debugged, and the other system to run gdb, should be connected using a null modem (i.e., a cross-over serial) cable.
* The vmlinux kernel image built, with kgdb enabled, needs to be copied from the system to be debugged, into the working directory on the system where gdb is going to be run.

To get gdb to connect to the waiting kernel, launch gdb from the shell and run these commands:

|  |
| --- |
| (gdb) file vmlinux  (gdb) set remote interrupt-sequence Ctrl-C  (gdb) set remotebaud 115200  (gdb) target remote /dev/ttyS0  (gdb) continue |

In the above commands, vmlinux is the kernel image copied from the system to be debugged.

## Debugging using gdb with kgdb

After this, it is all like debugging an application from gdb. One may stop execution using Ctrl+C, add break points using b[reak], stop execution using s[tep] or n[ext] … — the usual gdb way. There are enough GDB tutorials available online, if you need them. In fact, if you are not comfortable with text-based GDB, use any of the standard GUI tools over gdb, like ddd, Eclipse, etc.

## Summing up

By now, Shweta was excited about wanting to try out kgdb. Since she needed two systems to try it out, she went to the Linux device drivers’ lab. There, she set up the systems and ran gdb as described above.