

Linux Kernel Module Management Utilities

LinuXpert Systems

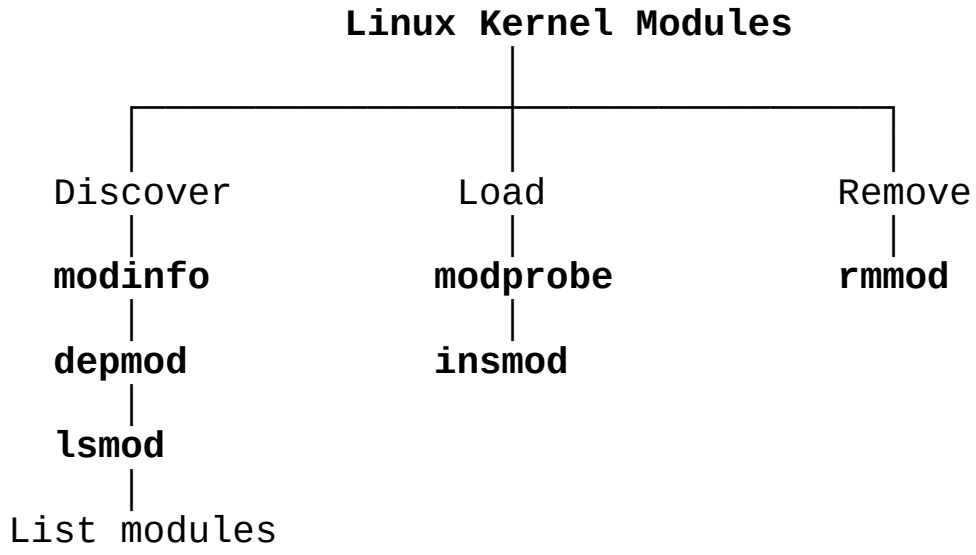
WhatsApp: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. The six core Linux Kernel Module Utilities 1
2. What is a Linux kernel module ? 2
3. Where are kernel modules stored ? 3
4. Kernel module file extensions 3
5. **`lsmod`** - List loaded kernel modules 4
6. **`modinfo`** - Examine a kernel module 5
7. **`insmod`** - Insert a kernel module 7
8. **`insmod`** versus **`modprobe`** 8
9. **`rmmod`** 9
10. **`rmmod`** versus **`modprobe -r`** 9
11. **`modprobe`** 10
12. Module **`aliases`** 11
13. **`depmod`** 12
14. Why is **`depmod`** necessary ? 13
15. When would you run **`depmod`** ? 13
16. The complete workflow 14
17. Practical laboratory exercise 15
18. Investigating a real driver 16
19. Module configuration 17
20. Blacklisting a module 17
21. Module parameters 18
22. Where kmod fits ? 18
23. Comparing the six utilities 19
24. The most important distinction 19
25. Kernel module lifecycle 20
26. Useful commands for practice 21
27. A useful teaching exercise 22
28. Six-command cheat sheet 23

1. The six core Linux kernel module management utilities as a single workflow:



The six utilities are:

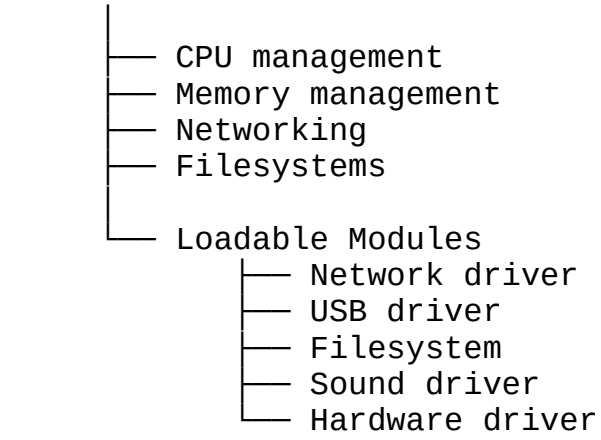
lsmod	→	List loaded modules
modinfo	→	Examine a module
modprobe	→	Load/unload modules intelligently
insmod	→	Insert a module directly
rmmod	→	Remove a module
depmod	→	Build module dependency information

2. What is a Linux kernel module ?

A **kernel module** is a piece of code that can be dynamically loaded into or removed from the Linux kernel while the system is running.

Instead of compiling every possible driver and kernel feature permanently into the kernel, Linux can keep many features as modules.

For example:



This provides several advantages:

- Smaller core kernel
- Drivers can be loaded when needed
- Hardware support can be added dynamically
- Kernel functionality can be removed when no longer required
- Distribution kernels can support a huge range of hardware

3. Where are kernel modules stored ?

On a typical Linux system, modules are stored under:

```
/lib/modules/<kernel-version>/
```

For example:

```
ls /lib/modules
```

You might see:

```
6.8.0-xx-generic
```

Then:

```
ls /lib/modules/$(uname -r)
```

A typical structure might look like:

```
/lib/modules/6.8.0-xx-generic/
├── kernel/
├── modules.alias
├── modules.dep
├── modules.dep.bin
├── modules.symbols
├── modules.symbols.bin
├── modules.alias.bin
└── ...
```

The actual module files are normally somewhere underneath:

```
/lib/modules/<kernel-version>/kernel/
```

---

4. Kernel module file extensions

You may encounter:

```
.ko
.ko.xz
.ko.zst
.ko.gz
```

`ko` means **kernel object**.

For example:

```
e1000e.ko
```

or:

```
e1000e.ko.zst
```

The compression is an implementation detail; tools such as `modprobe` and `modinfo` can normally handle compressed modules transparently.

---

5. `lsmod` — List loaded kernel modules

`lsmod` displays the kernel modules that are **currently loaded into the running kernel**.

Basic command:

```
lsmod
```

Typical output:

```
Module                  Size  Used by
bridge                311296  1
snd_hda_intel          53248  2
snd_hda_codec         180224  1 snd_hda_intel
```

The columns are:

```
Module
Size
Used by
```

Understanding `Used by`

Suppose you see:

```
snd_hda_codec  180224  1 snd_hda_intel
```

This indicates that another module is using it.

Conceptually:

```
snd_hda_intel
      │
      ▼
snd_hda_codec
```

This dependency relationship becomes important when trying to remove modules.

Finding a particular module

```
lsmod | grep e1000e
```

or:

```
lsmod | grep nvme
```

What actually is `lsmod`?

On modern Linux systems, `lsmod` is generally associated with the **kmod** project/package.

You can investigate:

```
which lsmod
```

and:

```
ls -l $(which lsmod)
```

You can also look at its documentation:

```
man lsmod
```

---

6. `modinfo` — Examine a kernel module

`modinfo` provides detailed information about a kernel module.

Basic usage:

```
modinfo <module>
```

For example:

```
modinfo e1000e
```

You may see information such as:

```
filename:
license:
description:
author:
alias:
depends:
vermagic:
```

filename

Shows where the module resides.

For example:

```
filename:
/lib/modules/.../kernel/drivers/net/ethernet/intel/e1000e/e1000e.ko
```

---

description

Describes the functionality of the module.

For example:

```
description: Intel(R) PRO/1000 Network Driver
```

---

license

Shows the module's declared license.

For example:

```
license: GPL
```

---

author

Shows the module author information when available.

---

depends

This is particularly useful.

For example:

```
depends: crc32c
```

It tells you about other modules that this module depends upon.

Conceptually:

```
Module A
    ├── depends on Module B
    └── depends on Module C
```

`vermagic`

`vermagic` contains kernel-version/build compatibility information.

This helps explain why a module built for one kernel may not work with another.

Display only the filename

```
modinfo -n e1000e
```

This is useful in scripts.

Display module parameters

```
modinfo -p e1000e
```

This can show parameters that can be supplied when loading the module.

---

7. `insmod` — Insert a kernel module

`insmod` directly inserts a module into the running kernel.

Basic syntax:

```
sudo insmod module.ko
```

For example:

```
sudo insmod mydriver.ko
```

Notice something important:

`insmod` normally expects the actual module file.

For example:

```
sudo insmod /path/to/mydriver.ko
```

rather than simply:

```
sudo insmod mydriver
```

---

8. `insmod` versus `modprobe`

This distinction is fundamental.

`insmod`

```
You
 │
 ▼
insmod
 │
 ▼
specific .ko file
 │
 ▼
kernel
```

It performs a relatively direct insertion.

`modprobe`

```
You
 │
 ▼
modprobe
 │
 ├── locate module
 ├── examine dependencies
 ├── load dependencies
 └── load requested module
          │
          ▼
       kernel
```

Therefore:

Use **`modprobe`** for normal system administration; use **`insmod`** mainly when you specifically need direct insertion of a module file.

---

9. `rmmod` — Remove a kernel module

`rmmod` removes a loaded module.

Basic syntax:

```
sudo rmmod <module>
```

For example:

```
sudo rmmod e1000e
```

However, removal will fail if the module is currently being used.

You may get an error similar to:

```
rmmod: ERROR: Module e1000e is in use
```

This is intentional.

Removing a driver that is actively being used can destabilize the system.

Check whether a module is loaded

```
lsmod | grep e1000e
```

You can also inspect its users through the `Used by` information.

---

10. `rmmod` versus `modprobe -r`

There are two common ways of removing a module:

```
sudo rmmod e1000e
```

and:

```
sudo modprobe -r e1000e
```

`modprobe -r` is generally more convenient because it understands the module dependency structure and can remove modules that are no longer needed.

Conceptually:

```
rmmod
  │
  └── Direct removal

modprobe -r
  │
  └── Dependency-aware removal
```

---

11. `modprobe` — Intelligent module management

`modprobe` is probably the **most important module-management command** for a Linux administrator.

It can:

- Load modules
- Remove modules
- Resolve dependencies
- Apply module configuration
- Handle module aliases
- Locate modules in the appropriate kernel module tree

Loading a module

```
sudo modprobe e1000e
```

Then:

```
lsmod | grep e1000e
```

Removing a module

```
sudo modprobe -r e1000e
```

Why is `modprobe` better than `insmod` ?

Suppose:

```
Module A
    │
    ├── Module B
    │
    └── Module C
```

If you execute:

```
insmod A.ko
```

you may encounter a missing dependency.

With:

```
sudo modprobe A
```

`modprobe` can determine the dependencies and load them appropriately.

Conceptually:

```
modprobe A
    │
    ├── load B
    ├── load C
    └── load A
```

---

12. Module aliases

Linux hardware can be associated with **module aliases**.

For example, a PCI device may have an identifier that maps to a particular driver.

`modprobe` can use these aliases to determine which module should be loaded.

This is one reason why plugging in hardware can automatically result in the appropriate kernel driver being loaded.

The process is broadly:

```
Hardware
    ↓
Kernel detects device
    ↓
Device identifier
    ↓
Module alias
    ↓
modprobe
    ↓
Driver module
```

---

13. `depmod` — Generate module dependency information

`depmod` is different from the other commands.

It doesn't primarily load or remove modules.

Instead, it **generates dependency information for kernel modules**.

Basic command:

```
sudo depmod
```

Usually you will encounter:

```
sudo depmod -a
```

The `-a` option processes all modules.

You can also specify a kernel version:

```
sudo depmod -a 6.8.0-xx-generic
```

14. Why is `depmod` necessary ?

Suppose you have:

```
driverA
   │
   └── depends on driverB
```

Linux needs to know this relationship.

`depmod` analyzes the modules and creates dependency information.

One important generated file is:

`modules.dep`

and often its binary counterpart:

`modules.dep.bin`

You can examine:

```
cat /lib/modules/$(uname -r)/modules.dep
```

It may contain relationships such as:

```
moduleA.ko: moduleB.ko moduleC.ko
```

The exact contents depend on your kernel and installed modules.

---

15. When would you run `depmod` ?

A common situation is when you **install a new kernel module manually**.

For example:

```
sudo cp mydriver.ko /lib/modules/$(uname -r)/extra/
```

Then:

```
sudo depmod -a
```

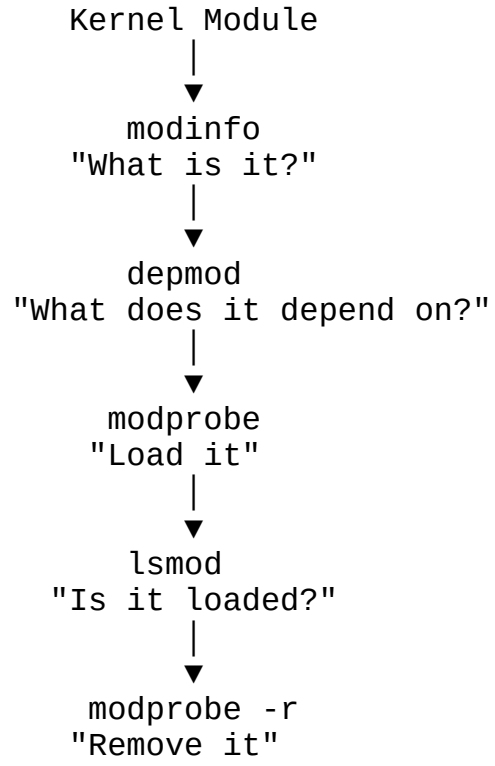
Now the module dependency/index information has been regenerated.

You can subsequently use:

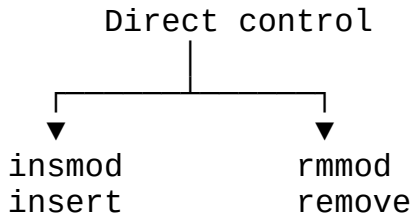
```
sudo modprobe mydriver
```

16. The complete workflow

These six commands make much more sense when viewed as one workflow.



`insmod` and **`rmmod`** provide more direct alternatives:



17. Practical laboratory exercise

Let's assume your system has the `loop` module.

First:

```
modinfo loop
```

Examine its information.

Then:

```
lsmod | grep '^loop'
```

If it is loaded, you'll see something similar to:

```
loop    ...
```

If it isn't loaded, you can try:

```
sudo modprobe loop
```

Then:

```
lsmod | grep '^loop'
```

Now remove it:

```
sudo modprobe -r loop
```

Then verify:

```
lsmod | grep '^loop'
```

This gives you the basic lifecycle:

```
modinfo
   ↓
modprobe
   ↓
lsmod
   ↓
modprobe -r
   ↓
lsmod
```

Do this only with a module you know is safe to remove. Don't experiment by removing essential modules such as storage, filesystem, network, or graphics drivers on a production system.

18. Investigating a real driver

A very useful exercise is to start with a hardware device and work backwards.

First:

```
lspci -k
```

You may see:

```
Kernel driver in use: ...
Kernel modules: ...
```

Suppose the driver is:

```
e1000e
```

Now:

```
modinfo e1000e
```

Then:

```
lsmod | grep e1000e
```

Then inspect the module file:

```
modinfo -n e1000e
```

Then investigate dependencies:

```
modinfo e1000e | grep depends
```

This creates a very useful learning chain:

```
Hardware
   ↓
lspci
   ↓
Kernel driver
   ↓
modinfo
   ↓
Module file
   ↓
lsmod
  ↓
Loaded state
```

19. Module configuration

Module configuration is commonly found under:

```
/etc/modprobe.d/
```

For example:

```
ls /etc/modprobe.d/
```

You may find files such as:

```
blacklist.conf
```

or distribution-specific configuration files.

These files can contain directives such as:

```
blacklist module_name
```

or:

```
options module_name parameter=value
```

This allows administrators to influence how modules are loaded.

---

20. Blacklisting a module

Suppose you don't want a particular module to load automatically.

A configuration file under:

```
/etc/modprobe.d/
```

can contain:

```
blacklist example_module
```

After changing module configuration, you may need to regenerate the appropriate initramfs depending on when the module is needed during boot.

On Ubuntu/Debian, that may involve:

```
sudo update-initramfs -u
```

Then reboot if appropriate.

Be careful: blacklisting a module can prevent essential hardware from working.

21. Module parameters

Some modules accept parameters.

First find them:

```
modinfo -p module_name
```

Then a module can potentially be loaded with parameters:

```
sudo modprobe module_name parameter=value
```

For example, conceptually:

```
modprobe driver option=value
```

The exact parameters are module-specific.

---

22. Where `kmod` fits

On modern Linux systems, these commands are closely associated with the kmod project:

```
kmod
 │
 ├── lsmod
 ├── modprobe
 ├── insmod
 ├── rmmod
 ├── modinfo
 └── depmod
```

You can check:

```
kmod --version
```

And:

```
man kmod
```

Interestingly, `kmod` itself acts as a common framework/interface for module management, while the familiar commands provide specialized interfaces.

23. Comparing the six utilities

Command	Primary function	Dependency aware?	Typical use
lsmod	List loaded modules	—	Inspect
modinfo	Display module metadata	Shows dependencies	Investigate
modprobe	Load/remove modules	**Yes**	Normal administration
insmod	Insert module file	**No**	Direct insertion
rmmod	Remove module	Limited/direct	Direct removal
depmod	Generate dependency database	Builds it	Module installation/update

24. The most important distinction

Students often confuse these three:

```
insmod
modprobe
depmod
```

Remember:

`insmod`

Insert this specific module file.

```
sudo insmod something.ko
```

`modprobe`

Find and manage this module intelligently.

```
sudo modprobe something
```

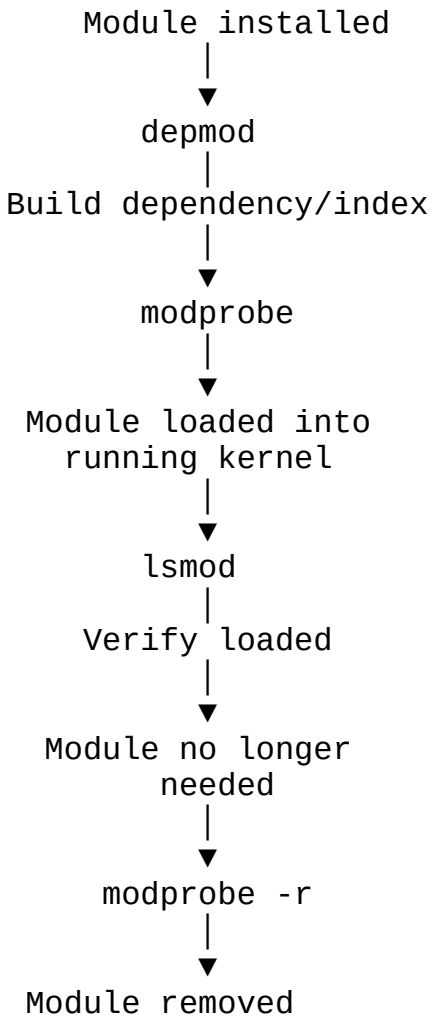
`depmod`

Build/update the information needed to understand module dependencies.

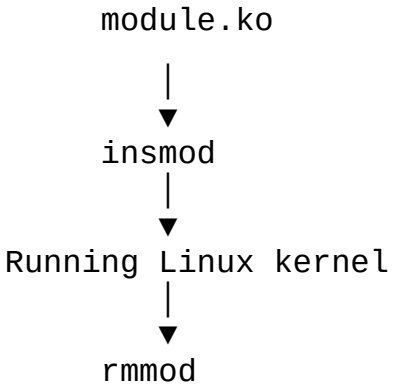
```
sudo depmod -a
```

25. Kernel module lifecycle

The complete lifecycle can be visualized as:



With direct operations:



26. Useful commands for practice

Practical command sheet:

```
# List loaded modules
lsmod

# Find a particular module
lsmod | grep module_name

# Get module information
modinfo module_name

# Show module file
modinfo -n module_name

# Show module parameters
modinfo -p module_name

# Load a module
sudo modprobe module_name

# Remove a module
sudo modprobe -r module_name

# Directly insert a module
sudo insmod module.ko

# Directly remove a module
sudo rmmod module_name

# Rebuild dependency information
sudo depmod -a

# Show kernel version
uname -r

# List installed modules
find /lib/modules/$(uname -r) -type f

# Show block/hardware information
lsblk

# Show PCI devices and drivers
lspci -k
```

27. A useful teaching exercise

"Identify the driver responsible for one piece of hardware on your Linux machine."

Have them perform:

```
lspci -k
```

Identify:

```
Kernel driver in use:
```

Then:

```
modinfo <driver>
```

Then:

```
lsmod | grep <driver>
```

Then:

```
modinfo -n <driver>
```

Finally:

```
modinfo <driver> | grep -E 'filename|description|license|depends|
vermagic'
```

You will then understand that:

```
Hardware
    ↓
Device identification
    ↓
Kernel driver
    ↓
Kernel module
    ↓
Module file (.ko)
    ↓
Module dependencies
    ↓
Running kernel
```

That is considerably more valuable than simply memorizing six commands.

28. Six-command cheat sheet

`lsmod`	`List currently loaded kernel modules`
`modinfo`	`Display information about a module`
`modprobe`	`Load/remove modules with dependencies`
`insmod`	`Directly insert a .ko module`
`rmmod`	`Directly remove a loaded module`
`depmod`	`Generate module dependency information`

The simplest way to remember them

`lsmod` → What is loaded?

`modinfo` → What is this module?

`modprobe` → Load/remove it intelligently.

`insmod` → Insert this module file.

`rmmod` → Remove this module.

`depmod` → Build the dependency information.