Linux Loadable Kernel Module HOWTO

Bryan Henderson

2006−09−24

Revision History

Revision v1.09	2006−09−24	Revised by: bjh
Fix typos.		
Revision v1.08	2006−03−03	Revised by: bjh
Add copyright information.		
Revision v1.07	2005−07−20	Revised by: bjh
Add some 2.6 info and disclaimers. Update references to Linux Device Drivers book, Linux Kernel Module Programming Guide.		
Revision v1.06	2005−01−12	Revised by: bjh
Cover Linux 2.6 briefly. Update hello.c and reference to Lkmpg. Add information about perils of unloading. Mention dmesg as way to see kernel messages.		
Revision v1.05	2004−01−05	Revised by: bjh
Add information on module.h and −DMODULE. Fix tldb.org to tldp.org. Add information on kallsyms.		
Revision v1.04	2003−10−10	Revised by: bjh
Fix typo: AHA154x should be AHA152x Add information on what module names the kernel module loader calls for. Add information on what an LKM does when you first load it. Add information on loop module. Change linuxdoc.org to tldp.org.		
Revision v1.03	2003−07−03	Revised by: bjh
Update on kernels that don't load into vmalloc space. Add explanation of "deleted" state of an LKM. Explain GPLONLY.		
Revision v1.02	2002−05−21	Revised by: bjh
Correct explanation of symbol versioning. Correct author of Linux Device Drivers. Add info about memory allocation penalty of LKM vs bound−in. Add LKM−to−LKM symbol matching requirement. Add open source licensing issue in LKM symbol resolution. Add SMP symbol versioning info.		
Revision v1.01	2001−08−18	Revised by: bjh
Add material on various features created in the last few years: kernel module loader, ksymoops symbols, kernel−version−dependent LKM file location.		
Revision v1.00	2001−06−14	Revised by: bjh
Initial release.		

This is the HOWTO for Linux loadable kernel modules (LKMs). It explains what they are and how to use and create them. It also includes documentation of parameters and other details of use of some particular modules.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1. Preface

Copyright and license information, as well as credits, are at the end of this document.

This HOWTO is maintained by Bryan Henderson, bryanh@giraffe−data.com. You can get the current version of this HOWTO from the Linux Documentation Project.

This document is mainly geared toward Linux 2.4. It covers secondarily earlier Linux. It covers newer Linux only partially, but the author's goal to cover it completely in a future release. See in particular Section 12. The subject changed quite a bit over the course of Linux 2.5 releases, and this document ignores 2.5, since the 2.5 releases were not considered production releases. Instead, the document treats all the changes in 2.5 releases as if they were new with Linux 2.6.

See Section 17 for a history of the document.

2. Introduction to Linux Loadable Kernel Modules

If you want to add code to a Linux kernel, the most basic way to do that is to add some source files to the kernel source tree and recompile the kernel. In fact, the kernel configuration process consists mainly of choosing which files to include in the kernel to be compiled.

But you can also add code to the Linux kernel while it is running. A chunk of code that you add in this way is called a loadable kernel module. These modules can do lots of things, but they typically are one of three things: 1) device drivers; 2) filesystem drivers; 3) system calls. The kernel isolates certain functions, including these, especially well so they don't have to be intricately wired into the rest of the kernel.

2.1. Terminology

Loadable kernel modules are often called just kernel modules or just modules, but those are rather misleading terms because there are lots of kinds of modules in the world and various pieces built into the base kernel can easily be called modules. We use the term loadable kernel module or LKM for the particular kinds of modules this HOWTO is about.

Some people think of LKMs as outside of the kernel. They speak of LKMs communicating with the kernel. This is a mistake; LKMs (when loaded) are very much part of the kernel. The correct term for the part of the kernel that is bound into the image that you boot, i.e. all of the kernel *except* the LKMs, is "base kernel." LKMs communicate with the base kernel.

In some other operating systems, the equivalent of a Linux LKM is called a "kernel extension."

Now what is "Linux"? Well, first of all, the name is used for two entirely different things, and only one of them is really relevant here:

1. The kernel and related items distributed as a package by Linus Torvalds.
2. A class of operating systems that traditionally are based on the Linux kernel.

Only the first of these is really useful in discussing LKMs. But even choosing this definition, people are often confused when it comes to LKMs. Is an LKM part of Linux or not? Though an LKM is always part of the kernel, it is part of Linux if it is distributed in the Linux kernel package, and not otherwise. Thus, if you have loaded into your kernel a device driver LKM that came with your device, you can't, strictly speaking, say that your kernel is Linux. Rather, it's a slight extension of Linux. As you might expect, it is commonplace to use the name "Linux" approximately –– Lots of variations on Linux are in use and are widely distributed, and referred to as "Linux." In this document, though, we will stick to the strictest definition in the interest of clarity.

2.2. History of Loadable Kernel Modules

LKMs did not exist in Linux in the beginning. Anything we use an LKM for today was built into the base kernel at kernel build time instead. LKMs have been around at least since Linux 1.2 (1995).

Device drivers and such were always quite modular, though. When LKMs were invented, only a small amount of work was needed on these modules to make them buildable as LKMs. However, it had to be done on each and every one, so it took some time. Since about 2000, virtually everything that makes sense as an LKM has at least had the option of being an LKM.

2.3. The Case For Loadable Kernel Modules

You often have a choice between putting a module into the kernel by loading it as an LKM or binding it into the base kernel. LKMs have a lot of advantages over binding into the base kernel and I recommend them wherever possible.

One advantage is that you don't have to rebuild your kernel as often. This saves you time and spares you the possibility of introducing an error in rebuilding and reinstalling the base kernel. Once you have a working base kernel, it is good to leave it untouched as long as possible.

Another advantage is that LKMs help you diagnose system problems. A bug in a device driver which is bound into the kernel can stop your system from booting at all. And it can be really hard to tell which part of the base kernel caused the trouble. If the same device driver is an LKM, though, the base kernel is up and running before the device driver even gets loaded. If your system dies after the base kernel is up and running, it's an easy matter to track the problem down to the trouble−making device driver and just not load that device driver until you fix the problem.

LKMs can save you memory, because you have to have them loaded only when you're actually using them. All parts of the base kernel stay loaded all the time. And in real storage, not just virtual storage.

LKMs are much faster to maintain and debug. What would require a full reboot to do with a filesystem driver built into the kernel, you can do with a few quick commands with LKMs. You can try out different parameters or even change the code repeatedly in rapid succession, without waiting for a boot.

LKMs are not slower, by the way, than base kernel modules. Calling either one is simply a branch to the memory location where it resides. [1]

Sometimes you *have* to build something into the base kernel instead of making it an LKM. Anything that is necessary to get the system up far enough to load LKMs must obviously be built into the base kernel. For example, the driver for the disk drive that contains the root filesystem must be built into the base kernel.

2.4. What LKMs Can't Do

There is a tendency to think of LKMs like user space programs. They do share a lot of their properties, but LKMs are definitely not user space programs. They are part of the kernel. As such, they have free run of the system and can easily crash it.

2.5. What LKMs Are Used For

There are six main things LKMs are used for:

- 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. For example, there is a device driver for ATA disk drives. There is one for NE2000 compatible Ethernet cards. To use any device, the kernel must contain a device driver for it.
- 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.
- Network drivers. A network driver interprets a network protocol. It feeds and consumes data streams at various layers of the kernel's networking function. For example, if you want an IPX link in your network, you would use the IPX driver.
- TTY line disciplines. These are essentially augmentations of device drivers for terminal devices.
- Executable interpreters. An executable interpreter loads and runs an executable. Linux is designed to be able to run executables in various formats, and each must have its own executable interpreter.

---

3. Making Loadable Kernel Modules

An LKM lives in a single ELF object file (normally named like "serial.o"). You typically keep all your LKM object files in a particular directory (near your base kernel image makes sense). When you use the **insmod** program to insert an LKM into the kernel, you give the name of that object file.

For the LKMs that are part of Linux, you build them as part of the same kernel build process that generates the base kernel image. See the README file in the Linux source tree. In short, after you make the base kernel image with a command such as **make zImage**, you will make all the LKMs with the command

```
make modules
```

This results in a bunch of LKM object files (*.o) throughout the Linux source tree. (In older versions of Linux, there would be symbolic links in the `modules` directory of the Linux source tree pointing to all those LKM object files). These LKMs are ready to load, but you probably want to install them in some appropriate directory. The conventional place is described in Section 5.6. The command **make modules_install** will copy them all over to the conventional locations.

Part of configuring the Linux kernel (at build time) is choosing which parts of the kernel to bind into the base kernel and which parts to generate as separate LKMs. In the basic question−and−answer configuration (**make config**), you are asked, for each optional part of the kernel, whether you want it bound into the kernel (a "Y" response), created as an LKM (an "M" response), or just skipped completely (an "N" response). Other configuration methods are similar.

As explained in Section 2.3, you should have only the bare minimum bound into the base kernel. And only skip completely the parts that you're sure you'll never want. There is very little to lose by building an LKM that you won't use. Some compile time, some disk space, some chance of a problem in the code killing the kernel build. That's it.

As part of the configuration dialog you also must choose whether to use symbol versioning or not. This choice affects building both the base kernel and the LKMs and it is crucial you get it right. See Section 6.

LKMs that are not part of Linux (i.e. not distributed with the Linux kernel) have their own build procedures which I will not cover. The goal of any such procedure, though, is always to end up with an ELF object file.

You don't necessarily have to rebuild all your LKMs and your base kernel image at the same time (e.g. you could build just the base kernel and use LKMs you built earlier with it) but it is always a good idea. See Section 6.

---

4. LKM Utilities

The programs you need to load and unload and otherwise work with LKMs are in the package modutils. You can find this package in this directory.

This package contains the following programs to help you use LKMs:

insmod
: Insert an LKM into the kernel.

rmmod
: Remove an LKM from the kernel.

depmod
: Determine interdependencies between LKMs.

kerneld
: Kerneld daemon program

ksyms
: Display symbols that are exported by the kernel for use by new LKMs.

lsmod
: List currently loaded LKMs.

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

modprobe
: 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.

Changes to the kernel often require changes to modutils, so be sure you're using a current version of modutils whenever you upgrade your kernel. modutils is always backward compatible (it works with older kernels), so there's no such thing as having too new a modutils.

Warning: **modprobe** invokes **insmod** and has its location hardcoded as `/sbin/insmod`. There may be other instances in modutils of the PATH not being used to find programs. So either modify the source code of modutils before you build it, or make sure you install the programs in their conventional directories.

---

5. How To Insert And Remove LKMs

The basic programs for inserting and removing LKMs are **insmod** and **rmmod**. See their man pages for details.

Inserting an LKM is conceptually easy: Just type, as superuser, a command like

```
insmod serial.o
```

(`serial.o` contains the device driver for serial ports (UARTs)).

However, I would be misleading you if I said the command just works. It is very common, and rather maddening, for the command to fail either with a message about a module/kernel version mismatch or a pile of unresolved symbols.

If it does work, though, the way to prove to yourself that you know what you're doing is to look at `/proc/modules` as described in Section 5.5.

Note that the examples in this section are from Linux 2.4. In Linux 2.6, the technical aspects of loading LKMs are considerably different, and the most visible manifestation of this is that the LKM file has a suffix of ".ko" instead of ".o". From the user point of view, it looks quite similar, though.

Now lets look at a more difficult insertion. If you try

```
insmod msdos.o
```

you will probably get a raft of error messages like:

```
  msdos.o: unresolved symbol fat_date_unix2dos
  msdos.o: unresolved symbol fat_add_cluster1
  msdos.o: unresolved symbol fat_put_super
  ...
```

This is because msdos.o contains external symbol references to the symbols mentioned and there are no such symbols exported by the kernel. To prove this, do a

```
cat /proc/ksyms
```

to list every symbol that is exported by the kernel (i.e. available for binding to LKMs). You will see that 'fat_date_unix2dos' is nowhere in the list.

(In Linux 2.6, there is no `/proc/ksyms`. Use `/proc/kallsyms` instead; the format is like the output of **nm**: look for symbols labelled "t").

How do you get it into the list? By loading another LKM, one which defines those symbols and exports them. In this case, it is the LKM in the file `fat.o`. So do

```
  insmod fat.o
```

and then see that "fat_date_unix2dos" is in `/proc/ksyms`. Now redo the

```
insmod msdos.o
```

and it works. Look at `/proc/modules` and see that both LKMs are loaded and one depends on the other:

```
msdos                  5632   0 (unused)
fat                   30400   0 [msdos]
```

How did I know `fat.o` was the module I was missing? Just a little ingenuity. A more robust way to address this problem is to use **depmod** and **modprobe** instead of **insmod**, as discussed below.

When your symbols look like "fat_date_unix2dos_R83fb36a1", the problem may be more complex than just getting prerequisite LKMs loaded. See Section 6.

When the error message is "kernel/module version mismatch," see Section 6.

Often, you need to pass parameters to the LKM when you insert it. For example, a device driver wants to know the address and IRQ of the device it is supposed to drive. Or the network driver wants to know how much diagnostic tracing you want it to do. Here is an example of that:

```
insmod ne.o io=0x300 irq=11
```

Here, I am loading the device driver for my NE2000−like Ethernet adapter and telling it to drive the Ethernet adapter at IO address 0x300, which generates interrupts on IRQ 11.

There are no standard parameters for LKMs and very few conventions. Each LKM author decides what parameters **insmod** will take for his LKM. Hence, you will find them documented in the documentation of the LKM. This HOWTO also compiles a lot of LKM parameter information in Section 15. For general information about LKM parameters, see Section 8.

To remove an LKM from the kernel, the command is like

```
rmmod ne
```

There is a command **lsmod** to list the currently loaded LKMs, but all it does is dump the contents of `/proc/modules`, with column headings, so you may just want to go to the horse's mouth and forget about **lsmod**.

5.1. Could Not Find Kernel Version...

A common error is to try to insert an object file which is not an LKM. For example, you configure your kernel to have the USB core module bound into the base kernel instead of generated as an LKM. In that case, you end up with a file `usbcore.o`, which looks pretty much the same as the `usbcore.o` you would get if you built it as an LKM. But you can't **insmod** that file.

So do you get an error message telling you that you should have configured the kernel to make USB core function an LKM? Of course not. This is Unix, and explanatory error messages are seen as a sign of weakness. The error message is

```
$ insmod usbcore.o
usbcore.o: couldn't find the kernel version this module was compiled for
```

What **insmod** is telling you is that it looked in `usbcore.o` for a piece of information any legitimate LKM would have −− the kernel version with which the LKM was intended to be used −− and it didn't find it. We know now that the reason it didn't find it is that the file isn't an LKM. See Section 10.2 for information on how you can see what **insmod** is seeing and confirm that the file is not in fact an LKM.

If this is a module you created yourself with the intention of it being an LKM, the next question you have is: Why isn't an LKM? The most usual cause of this is that you did not include `linux/module.h` at the top of your source code and/or did not define the `MODULE>` macro. `MODULE` is intended to be set via the compile command (`−DMODULE`) and determine whether the compilation produces an LKM or an object file for binding into the base kernel. If your module is like most modern modules and can be built *only* as an LKM, then you should just define it in your source code (`#define MODULE`) before you include `include/module.h`.

5.2. What Happens When An LKM Loads

So you've successfully loaded an LKM, and verified that via `/proc/modules`. But how do you know it's working? That's up to the LKM, and varies according to what kind of LKM it is, but here are some of the more common actions of an LKM upon being loaded.

The first thing a device driver LKM does after loading (which is what the module would do at boot time if it were bound into the base kernel) is usually to search the system for a device it knows how to drive. Just how it does this search varies from one driver to the next, and can usually be controlled by module parameters. But in any case, if the driver doesn't find any device it is capable of driving, it causes the load to fail. Otherwise, the driver registers itself as the driver for a particular major number and you can start using the device it found via a device special file that specifies that major number. It may also register itself as the handler for the interrupt level that the device uses. It may also send setup commands to the device, so you may see lights blink or something like that.

You can see that a device driver has registered itself in the file `/proc/devices`. You can see that the device driver is handling the device's interrupts in `/proc/interrupts`.

A nice device driver issues kernel messages telling what devices it found and is prepared to drive. (Kernel messages in most systems end up on the console and in the file `/var/log/messages`. You can also display recent ones with the **dmesg** program). Some drivers, however, are silent. A nice device driver also gives you (in kernel messages) some details of its search when it fails to find a device, but many just fail the load without explanation, and what you get is a list of guesses from **insmod** as to what the problem might have been.

A network device (interface) driver works similarly, except that the LKM registers a device name of its choosing (e.g. `eth0`) rather than a major number. You can see the currently registered network device names in `/proc/net/dev`

A filesystem driver, upon loading, registers itself as the driver for a filesystem type of a certain name. For example, the `msdos` driver registers itself as the driver for the filesystem type named `msdos`. (LKM authors typically name the LKM the same as the filesystem type it will drive).

5.3. Intelligent Loading Of LKMs − Modprobe

Once you have module loading and unloading figured out using **insmod** and **rmmod**, you can let the system do more of the work for you by using the higher level program **modprobe**. See the **modprobe** man page for details.

The main thing that **modprobe** does is automatically load the prerequisites of an LKM you request. It does this with the help of a file that you create with **depmod** and keep on your system.

Example:

```
modprobe msdos
```

This performs an **insmod** of `msdos.o`, but before that does an **insmod** of `fat.o`, since you have to have `fat.o` loaded before you can load `msdos.o`.

The other major thing **modprobe** does for you is to find the object module containing the LKM given just the name of the LKM. For example, **modprobe msdos** might load `/lib/2.4.2−2/fs/msdos.o`. In fact, **modprobe**'s argument may be a totally symbolic name that you have associated with some actual module. For example, **modprobe eth0** loads the appropriate network device driver to create and drive your `eth0` device, assuming you set that up properly in `modules.conf`. Check out the man pages for **modprobe** and the configuration file `modules.conf` (usually `/etc/modules.conf`) for details on the search rules **modprobe** uses.

modprobe is especially important because it is by default the program that the kernel module loader uses to load an LKM on demand. So if you use automatic module loading, you will need to set up `modules.conf` properly or things will not work. See Section 5.4.

depmod scans your LKM object files (typically all the `.o` files in the appropriate `/lib/modules` subdirectory) and figures out which LKMs prerequire (refer to symbols in) other LKMs. It generates a dependency file (typically named `modules.dep`), which you normally keep in `/lib/modules` for use by **modprobe**.

You can use **modprobe** to remove stacks of LKMs as well.

Via the LKM configuration file (typically `/etc/modules.conf`), you can fine tune the dependencies and do other fancy things to control LKM selections. And you can specify programs to run when you insert and remove LKMs, for example to initialize a device driver.

If you are maintaining one system and memory is not in short supply, it is probably easier to avoid **modprobe** and the various files and directories it needs, and just do raw **insmod**s in a startup script.

5.4. Automatic LKM Loading and Unloading

5.4.1. Automatic Loading

You can cause an LKM to be loaded automatically when the kernel first needs it. You do this with either the kernel module loader, which is part of the Linux kernel, or the older version of it, a `kerneld` daemon.

As an example, let's say you run a program that executes an open system call for a file in an MS−DOS filesystem. But you don't have a filesystem driver for the MS−DOS filesystem either bound into your base kernel or loaded as an LKM. So the kernel does not know how to access the file you're opening on the disk.

The kernel recognizes that it has no filesystem driver for MS−DOS, but that one of the two automatic module loading facilities are available and uses it to cause the LKM to be loaded. The kernel then proceeds with the open.

Automatic kernel module loading is really not worth the complexity in most modern systems. It may make sense in a very small memory system, because you can keep parts of the kernel in memory only when you need them. But the amount of memory these modules uses is so cheap today that you will normally be a lot better off just loading all the modules you might need via startup scripts and leaving them loaded.

Red Hat Linux uses automatic module loading via the kernel module loader.

Both the kernel module loader and `kerneld` use **modprobe**, ergo **insmod**, to insert LKMs. See Section 5.3.

5.4.1.1. Kernel Module Loader

There is some documentation of the kernel module loader in the file `Documentation/kmod.txt` in the Linux 2.4 source tree. This section is more complete and accurate than that file. You can also look at its source code in `kernel/kmod.c`.

The kernel module loader is an optional part of the Linux kernel. You get it if you select the CONFIG_KMOD feature when you configure the kernel at build time.

When a kernel that has the kernel module loader needs an LKM, it creates a user process (owned by the superuser, though) that executes **modprobe** to load the LKM, then exits. By default, it finds **modprobe** as `/sbin/modprobe`, but you can set up any program you like as **modprobe** by writing its file name to `/proc/sys/kernel/modprobe`. For example:

```
# echo "sbin/mymodprobe" >/proc/sys/kernel/modprobe
```

The kernel module loader passes the following arguments to the **modprobe**: Argument Zero is the full file name of **modprobe**. The regular arguments are `-s`, `-k`, and the name of the LKM that the kernel wants. `-s` is the user−hostile form of `--syslog`; `-k` is the cryptic way to say `--autoclean`. I.e. messages from **modprobe** will go to syslog and the loaded LKM will have the "autoclean" flag set.

The most important part of the **modprobe** invocation is, of course, the module name. Note that the "module name" argument to **modprobe** is not necessarily a real module name. It is often a symbolic name representing the role that module plays and you use an `alias` statement in `modules.conf` to tell what LKM gets loaded. For example, if your Ethernet adapter requires the `3c59x` LKM, you would have probably need the line

```
alias eth0 3c59x
```

in `/etc/modules.conf`. Here is what the kernel module loader uses for a module name in some of the more popular cases (there are about 20 cases in which the kernel calls on the kernel module loader to load a module):

- When you try access a device and no device driver has registered to serve that device's major number, the kernel requests the module by the name `block-major-N` or `char-major-N` where `N` is the major number in decimal without leading zeroes.
- When you try to access a network interface (maybe by running **ifconfig** against it) and no network device driver has registered to serve an interface by that name, the kernel requests the module named the same as the interface name (e.g. `eth0`). This applies to drivers for non−physical interfaces such as `ppp0` as well.
- When you try to access a socket in a protocol family which no protocol driver has registered to drive, the kernel requests the module named `net-pf-N`, where `N` is the protocol family number (in decimal without leading zeroes).
- When you try to NFS export a directory or otherwise access the NFS server via the NFS system call, the kernel requests the module named `nfsd`.
- The ATA device driver (named `ide`) loads the relevant drivers for classes of ATA devices by the names: `ide-disk`, `ide-cd`, `ide-floppy`, `ide-tape`, and `ide-scsi`.

The kernel module loader runs **modprobe** with the following environment variables (only): HOME=`/`; TERM=`linux`; PATH=`/sbin:/usr/sbin:/bin:/usr/bin`.

The kernel module loader was new in Linux 2.2 and was designed to take the place of `kerneld`. It does not, however, have all the features of `kerneld`.

In Linux 2.2, the kernel module loader creates the above mentioned process directly. In Linux 2.4, the kernel module loader submits the module loading work to Keventd and it runs as a child process of Keventd.

The kernel module loader is a pretty strange beast. It violates layering as Unix programmers generally understand it and consequently is inflexible, hard to understand, and not robust. Many system designers would bristle just at the fact that it has the PATH hardcoded. You may prefer to use `kerneld` instead, or not bother with automatic loading of LKMs at all.

5.4.1.2. Kerneld

`kerneld` is explained at length in the Kerneld mini−HOWTO, available from the Linux Documentation Project.

`kerneld` is a user process, which runs the kerneld program from the modutils package. `kerneld` sets up an IPC message channel with the kernel. When the kernel needs an LKM, it sends a message on that channel to `kerneld` and `kerneld` runs **modprobe** to load the LKM, then sends a message back to the kernel to say that it is done.

5.4.2. Automatic Unloading − Autoclean

5.4.2.1. The Autoclean Flag

Each loaded LKM has an autoclean flag which can be set or unset. You control this flag with parameters to the `init_module` system call. Assuming you do that via **insmod**, you use the `−−autoclean` option.

You can see the state of the autoclean flag in `/proc/modules`. Any LKM that has the flag set has the legend `autoclean` next to it.

5.4.2.2. Removing The Autoclean LKMs

The purpose of the autoclean flag is to let you automatically remove LKMs that haven't been used in a while (typically 1 minute). So by using automatic module loading and unloading, you can keep loaded only parts of the kernel that are presently needed, and save memory.

This is less important than it once was, with memory being much cheaper. If you don't need to save memory, you shouldn't bother with the complexity of module loader processes. Just load everything you might need via an initialization script and keep it loaded.

There is a form of the `delete_module` system call that says, "remove all LKMs that have the autoclean flag set and haven't been used in a while." Kerneld typically calls this once per minute. You can call it explicitly with an **rmmod −−all** command.

As the kernel module loader does not do any removing of LKMs, if you use that you might want to have a cron job that does a **rmmod −−all** periodically.

5.5. /proc/modules

To see the presently loaded LKMs, do

```
cat /proc/modules
```

You see a line like

```
serial                  24484   0
```

The left column is the name of the LKM, which is normally the name of the object file from which you loaded it, minus the ".o" suffix. You can, however, choose any name you like with an option on **insmod**.

The "24484" is the size in bytes of the LKM in memory.

The "0" is the use count. It tells how many things presently depend on the LKM being loaded. Typical "things" are open devices or mounted fileystems. It is important because you cannot remove an LKM unless the use count is zero. The LKM itself maintains this count, but the module manager uses it to decide whether to permit an unload.

There is an exception to the above description of the use count. You may see −1 in the use count column. What that means is that this LKM does not use use counts to determine when it is OK to unload. Instead, the LKM has registered a subroutine that the module manager can call that will return an indication of whether or not it is OK to unload the LKM. In this case, the LKM ought to provide you with some custom interface, and some documentation, to determine when the LKM is free to be unloaded.

Do not confuse use count with "dependencies", which are described below.

Here is another example, with more information:

```
lp                      5280   0  (unused)
parport_pc              7552   1
parport                 7600   1  [lp parport_pc]
```

The stuff in square brackets ("[lp parport_pc]") describes dependencies. Here, the modules `lp` and `parport_pc` both refer to addresses within module `parport` (via external symbols that `parport` exports). So `lp` and `parport_pc` are "dependent" on (and are "dependencies of") `parport`.

You cannot unload an LKM that has dependencies. But you can remove those dependencies by unloading the dependent LKMs.

The "(unused)" legend means the LKM has never been used, i.e. it has never been in a state where it could not be unloaded. The kernel tracks this information for one simple reason: to assist in automatic LKM unloading policy. In a system where LKMs are loaded and unloaded automatically (see Section 5.4), you don't want to automatically load an LKM and then, before the guy who needed it loaded has a chance to use it, unload it because it is not in use.

Here is something you won't normally see:

```
mydriver                8154   0  (deleted)
```

This is an LKM that is in "deleted" state. It's something of a misnomer −− what it means is that the LKM is in the process of being unloaded. You can no longer load LKMs that depend on it, but it's still present in the system. Unloading an LKM is usually close to instantaneous, so if you see this status, you probably have a broken LKM. Its cleanup routine probably got into an infinite loop or stall or crashed (causing a kernel oops). If that's the case, the only way to clear this status is to reboot.

There are similar statuses "initializing" and "uninitialized".

The legend "(autoclean)" refers to the autoclean flag, discussed in Section 5.4.

5.6. Where Are My LKM Files On My System?

The LKM world is flexible enough that the files you need to load could live just about anywhere on your system, but there is a convention that most systems follow: The LKM .o files are in the directory `/lib/modules`, divided into subdirectories. There is one subdirectory for each version of the kernel, since LKMs are specific to a kernel (see Section 6). Each subdirectory contains a complete set of LKMs.

The subdirectory name is the value you get from the **uname −−release** command, for example `2.2.19`. Section 6.3 tells how you control that value.

When you build Linux, a standard **make modules** and **make modules_install** should install all the LKMs that are part of Linux in the proper release subdirectory.

If you build a lot of kernels, another organization may be more helpful: keep the LKMs together with the base kernel and other kernel−related files in a subdirectory of /boot. The only drawback of this is that you cannot have /boot reside on a tiny disk partition. In some systems, /boot is on a special tiny "boot partition" and contains only enough files to get the system up to the point that it can mount other filesystems.

6. Unresolved Symbols

The most common and most frustrating failure in loading an LKM is a bunch of error messages about unresolved symbols, like this:

```
msdos.o: unresolved symbol fat_date_unix2dos
msdos.o: unresolved symbol fat_add_cluster1
msdos.o: unresolved symbol fat_put_super
...
```

There are actually a bunch of different problems that result in this symptom. In any case, you can get closer to the problem by looking at `/proc/ksyms` and confirming that the symbols in the message are indeed not in the list.

6.1. Some LKMs Prerequire Other LKMs

One reason you get this is because you have not loaded another LKM that contains instructions or data that your LKM needs to access. A primary purpose of **modprobe** is to avoid this failure. See Section 5.3.

6.2. An LKM Must Match The Base Kernel

The designers of loadable kernel modules realized there would be a problem with having the kernel in multiple files, possibly distributed independently of one another. What if the LKM `mydriver.o` was written and compiled to work with the Linux 1.2.1 base kernel, and then someone tried to load it into a Linux 1.2.2 kernel? What if there was a change between 1.2.1 and 1.2.2 in the way a kernel subroutine that `mydriver.o` calls works? These are internal kernel subroutines, so what's to stop them from changing from one release to the next? You could end up with a broken kernel.

To address this problem, the creators of LKMs endowed them with a kernel version number. The special `.modinfo` section of the `mydriver.o` object file in this example has "1.2.1" in it because it was compiled using header files from Linux 1.2.1. Try to load it into a 1.2.2 kernel and **insmod** notices the mismatch and fails, telling you you have a kernel version mismatch.

But wait. What's the chance that there really is an incompatibility between Linux 1.2.1 and 1.2.2 that will affect `mydriver.o`? `mydriver.o` only calls a few subroutines and accesses a few data structures. Surely they don't change with every minor release. Must we recompile every LKM against the header files for the particular kernel into which we want to insert it?

To ease this burden, **insmod** has a *`-f`* option that "forces" **insmod** to ignore the kernel version mismatch and insert the module anyway. Because it is so unusual for there to be a significant difference between any two kernel versions, I recommend you always use *`-f`*. You will, however, still get a warning message about the mismatch. There's no way to shut that off.

But LKM designers still wanted to address the problem of incompatible changes that do occasionally happen. So they invented a very clever way to allow the LKM insertion process to be sensitive to the actual content of each kernel subroutine the LKM uses. It's called symbol versioning (or sometimes less clearly, "module versioning."). It's optional, and you select it when you configure the kernel via the "CONFIG_MODVERSIONS" kernel configuration option.

When you build a base kernel or LKM with symbol versioning, the various symbols exported for use by LKMs get defined as macros. The definition of the macro is the same symbol name plus a hexadecimal hash value of the parameter and return value types for the subroutine named by the symbol (based on an analysis by the program **genksyms** of the source code for the subroutine). So let's look at the `register_chrdev` subroutine. `register_chrdev` is a subroutine in the base kernel that device driver LKMs often call. With symbol versioning, there is a C macro definition like

```
#define register_chrdev register_chrdev_Rc8dc8350
```

This macro definition is in effect both in the C source file that defines `register_chrdev` and in any C source file that refers to `register_chrdev`, so while your eyes see `register_chrdev` as you read the code, the C preprocessor knows that the function is really called `register_chrdev_Rc8dc8350`.

What is the meaning of that garbage suffix? It is a hash of the data types of the parameters and return value of `register_chrdev`. No two combinations of parameter and return value types have the same hash value.

So let's say someone adds a paramater to `register_chrdev` between Linux 1.2.1 and Linux 1.2.2. In 1.2.1, `register_chrdev` is a macro for `register_chrdev_Rc8dc8350`, but in 1.2.2, it is a macro for `register_chrdev_R12f8dc01`. In `mydriver.o`, compiled with Linux 1.2.1 header files, there is an external reference to `register_chrdev_Rc8dc8350`, but there is no such symbol exported by the 1.2.2 base kernel. Instead, the 1.2.2 base kernel exports a symbol `register_chrdev_R12f8dc01`.

So if you try to insmod this 1.2.1 `mydriver.o` into this 1.2.2 base kernel, you will fail. And the error message isn't one about mismatched kernel versions, but simply "unresolved symbol reference."

As clever as this is, it actually works against you sometimes. The way **genksyms** works, it often generates different hash values for parameter lists that are essentially the same.

And symbol versioning doesn't even guarantee compatibility. It catches only a small subset of the kinds of changes in the definition of a function that can make it not backward compatible. If the way `register_chrdev` interprets one of its parameters changes in a non−backward−compatible way, its version suffix won't change −− the parameter still has the same C type.

And there's no way an option like *-f* on **insmod** can get around this.

So it is generally not wise to use symbol versioning.

Of course, if you have a base kernel that was compiled with symbol versioning, then you must have all your LKMs compiled likewise, and vice versa. Otherwise, you're guaranteed to get those "unresolved symbol reference" errors.

6.3. If You Run Multiple Kernels

Now that we've seen how you often have different versions of an LKM for different base kernels, the question arises as to what to do about a system that has multiple kernel versions (i.e. you can choose a kernel at boot time). You want to make sure that the LKMs built for Kernel A get inserted when you boot Kernel A, but the LKMs built for Kernel B get inserted when you boot Kernel B.

In particular, whenever you upgrade your kernel, if you're smart, you keep both the new kernel and the old kernel on the system until you're sure the new one works.

The most common way to do this is with the LKM−hunting feature of **modprobe**. **modprobe** understands the conventional LKM file organization described in Section 5.6 and loads LKMs from the appropriate subdirectory depending on the kernel that is running.

You set the **uname −−release** value, which is the name of the subdirectory in which **modprobe** looks, by editing the main kernel makefile when you build the kernel and setting the VERSION, PATCHLEVEL, SUBLEVEL, and EXTRAVERSION variables at the top.

6.4. SMP symbols

Besides the checksum mentioned above, the symbol version prefix contains "smp" if the symbol is defined in or referenced by code that was built for symmetric multiprocessing (SMP) machines. That means it was built for use on a system that may have more than one CPU. You choose whether to build in SMP capability or not via the Linux kernel configuration process (**make config**, etc.), to wit with the CONFIG_SMP configuration option.

So if you use symbol versioning, you will get unresolved symbols if the base kernel was built with SMP capability and the LKM you're inserting was not, or vice versa.

If you don't use symbol versioning, never mind.

Note that there's generally no reason to omit SMP capability from a kernel, even if you have only one CPU. Just because the capability is there doesn't mean you have to have multiple CPUs. However, there are some machines on which the SMP−capable kernel will not boot because it reaches the conclusion that there are zero CPUs!

6.5. You Are Not Licensed To Access The Symbol

The copyright owners of some kernel code license their programs to the public to make and use copies, but only in restricted ways. For example, the license may say you may only call your copy of the program from a program which is similarly licensed to the public.

(Is that confusing? Here's an example: Bob writes an LKM that provides data compression subroutines to other LKMs. He licenses his program to the public under the GNU Public License (GPL). According to some interpretations, that license says if you make a copy of Bob's LKM, you can't allow Mary's LKM to call its compression subroutines if Mary does not supply her source code to the world too. The idea is to encourage Mary to open up her source code).

To support and enforce such a license, the licensor can cause his program to export symbols under a special name that is the real name of the symbol plus the prefix "GPLONLY". A naive loader of a client LKM would not be able to resolve those symbols. Example: Bob's LKM provides the service bobsService() and declares it to be a GPL symbol. The LKM consequently exports bobsService() under the name GPLONLY_bobsService. If Mary's LKM refers to bobsService, the naive loader will not be able to find it, so will fail to load Mary's LKM.

However, a modern version of **insmod** knows to check for GPLONLY_bobsService if it can't find bobsService. But the modern **insmod** will refuse to do so unless Mary's LKM declares that it is licensed to the public under GPL.

The purpose of this appears to be to prevent anyone from accidentally violating a license (or from credibly claiming that he accidentally violated the license). It is not difficult to circumvent the restriction if you want to.

If you see this failure, it is probably because you're using an old loader (**insmode**) that doesn't know about GPLONLY.

The only other cause would be that the LKM author wrote the source code in such a way that it will never load into any Linux kernel, so there would be no point in the author distributing it.

6.6. An LKM Must Match Prerequisite LKMs

The same ways an LKM must be compatible with the base kernel, it must be compatible with any LKMs which it accesses (e.g. the first LKM calls a subroutine in the second). The preceding sections limit their discussions to the base kernel just to keep it simple.

7. How To Boot Without A Disk Device Driver

For most systems, the ATA disk device driver must be bound into the base kernel because the root filesystem is on an ATA disk [2] and the kernel cannot mount the root filesystem, much less read any LKMs from it, without the ATA disk driver. But if you really want the device driver for your root filesystem to be an LKM, here's how to do it with Initrd:

"Initrd" is the name of the "initial ramdisk" feature of Linux. With this, you have your loader (probably LILO or Grub) load a filesystem into memory (as a ramdisk) before starting the kernel. When it starts the kernel, it tells it to mount the ramdisk as the root filesystem. You put the disk device driver for your real root filesystem and all the software you need to load it in that ramdisk filesystem. Your startup programs (which live in the ramdisk) eventually mount the real (disk) filesystem as the root filesystem. Note that a ramdisk doesn't require any device driver.

This does not free you, however, from having to bind into the base kernel 1) the filesystem driver for the filesystem in your ramdisk, and 2) the executable interpreter for the programs in the ramdisk.

8. About Module Parameters

It is useful to compare parameters that get passed to LKMs and parameters that get passed to modules that are bound into the base kernel, especially since modules often can be run either way.

We've seen above that you pass parameters to an LKM by specifying something like `io=0x300` on the **insmod** command. For a module that is bound into the base kernel, you pass parameters to it via the kernel boot parameters. One common way to specify kernel boot parameters is at a **lilo** boot prompt. Another is with an `append` statement in the **lilo** configuration file.

The kernel initializes an LKM at the time you load it. It initializes a bound–in module at boot time.

Since there is only one string of kernel boot parameters, you need some way within that string to identify which parameters go to which modules. The rule for this is that if there is a module named `xyz`, then a kernel boot parameter named *`xyz`* is for that module. The value of a kernel boot parameter is an arbitrary string that makes sense only to the module.

This is why you sometimes see an LKM whose only parameter is its own name. E.g. you load the Mitsumi CDROM driver with a command like

```
insmod mcd mcd=0x340
```

It seems ridiculous to have the parameter named *`mcd`* instead of, say, *`io`*, but this is done for consistency with the case where you bind `mcd` into the base kernel, in which case you would select the I/O port address with the characters *`mcd=0x340`* in the kernel boot parameters.

9. Persistent Data

Some LKMs are set up to retain information from one load to the next. This is called persistent data. When you remove one of these LKMs with **rmmod**, **rmmod** extracts certain values from the LKM's working storage and stores them in a file. When you next insert the LKM with **insmod**, **insmod** reads the persistent data from the file and inserts it into the LKM.

See the `--persist` option on **insmod** and **rmmod**.

Persistent data was introduced in November 2000.

10. Technical Details

10.1. How They Work

insmod makes an `init_module` system call to load the LKM into kernel memory. Loading it is the easy part, though. How does the kernel know to use it? The answer is that the `init_module` system call invokes the LKM's initialization routine right after it loads the LKM. **insmod** passes to `init_module` the address of the subroutine in the LKM named `init_module` as its initialization routine.

(This is confusing –– every LKM has a subroutine named `init_module`, and the base kernel has a system call by that same name, which is accessible via a subroutine in the standard C library also named `init_module`).

The LKM author set up `init_module` to call a kernel function that registers the subroutines that the LKM contains. For example, a character device driver's `init_module` subroutine might call the kernel's `register_chrdev` subroutine, passing the major and minor number of the device it intends to drive and the address of its own "open" routine among the arguments. `register_chrdev` records in base kernel tables that when the kernel wants to open that particular device, it should call the open routine in our LKM.

But the astute reader will now ask how the LKM's `init_module` subroutine knew the address of the base kernel's `register_chrdev` subroutine. This is not a system call, but an ordinary subroutine bound into the base kernel. Calling it means branching to its address. So how does our LKM, which was not compiled anywhere near the base kernel, know that address? The answer to this is the relocation that happens at **insmod** time.

How that relocation happens is fundamentally different between Linux 2.4 and Linux 2.6. In 2.6, **insmod** pretty much just passes the verbatim contents of the LKM file (.ko file) to the kernel and the kernel does the relocation. In 2.4, **insmod** does the relocation and passes a fully linked module image, ready to stuff into kernel memory, to the kernel. *The description below covers the 2.4 case.*

insmod functions as a relocating linker/loader. The LKM object file contains an external reference to the symbol `register_chrdev`. **insmod** does a `query_module` system call to find out the addresses of various symbols that the existing kernel exports. `register_chrdev` is among these. `query_module` returns the address for which `register_chrdev` stands and **insmod** patches that into the LKM where the LKM refers to `register_chrdev`.

If you want to see the kind of information **insmod** can get from a `query_module` system call, look at the contents of `/proc/ksyms`.

Note that some LKMs call subroutines in other LKMs. They can do this because of the `__ksymtab` and `.kstrtab` sections in the LKM object files. These sections together list the external symbols within the LKM object file that are supposed to be accessible by other LKMs inserted in the future. **insmod** looks at `__ksymtab` and `.kstrtab` and tells the kernel to add those symbols to its exported kernel symbols table.

To see this for yourself, insert the LKM `msdos.o` and then notice in `/proc/ksyms` the symbol `fat_add_cluster` (which is the name of a subroutine in the `fat.o` LKM). Any subsequently inserted LKM can branch to `fat_add_cluster`, and in fact `msdos.o` does just that.

10.2. The .modinfo Section

An ELF object file consists of various named sections. Some of them are basic parts of an object file, for example the `.text` section contains executable code that a loader loads. But you can make up any section you want and have it used by special programs. For the purposes of Linux LKMs, there is the `.modinfo` section. An LKM doesn't have to have a section named `.modinfo` to work, but the macros you're supposed to use to code an LKM cause one to be generated, so they generally do.

To see the sections of an object file, including the `.modinfo` section if it exists, use the **objdump** program. For example:

To see all the sections in the object file for the msdos LKM:

```
objdump msdos.o --section-headers
```

To see the contents of the `.modinfo` section:

```
objdump msdos.o --full-contents --section=.modinfo
```

You can use the **modinfo** program to interpret the contents of the `.modinfo` section.

So what is in the `.modinfo` section and who uses it? **insmod** uses the `.modinfo` section for the following:

- It contains the kernel release number for which the module was built. I.e. of the kernel source tree whose header files were used in compiling the module.

  insmod uses that information as explained in Section 6.
- It describes the form of the LKM's parameters. **insmod** uses this information to format the parameters you supply on the **insmod** command line into data structure initial values, which **insmod** inserts into the LKM as it loads it.

10.3. The __ksymtab And .kstrtab Sections

Two other sections you often find in an LKM object file are named `__ksymtab` and `.kstrtab`. Together, they list symbols in the LKM that should be accessible (exported) to other parts of the kernel. A symbol is just a text name for an address in the LKM. LKM A's object file can refer to an address in LKM B by name (say, `getBinfo`"). When you insert LKM A, after having inserted LKM B, **insmod** can insert into LKM A the actual address within LKM B where the data/subroutine named `getBinfo` is loaded.

See Section 10.1 for more mind−numbing details of symbol binding.

10.4. Ksymoops Symbols

insmod adds a bunch of exported symbols to the LKM as it loads it. These symbols are all intended to help **ksymoops** do its job. **ksymoops** is a program that interprets and "oops" display. And "oops" display is stuff that the Linux kernel displays when it detects an internal kernel error (and consequently terminates a process). This information contains a bunch of addresses in the kernel, in hexadecimal.

ksymoops looks at the hexadecimal addresses, looks them up in the kernel symbol table (which you see in `/proc/ksyms` (Linux 2.4) or `/proc/kallsyms` (Linux 2.6), and translates the addresses in the oops message to symbolic addresses, which you might be able to look up in an assembler listing.

So lets say you have an LKM crash on you. The oops message contains the address of the instruction that choked, and what you want **ksymoops** to tell you is 1) in what LKM is that instruction, and 2) where is the instruction relative to an assembler listing of that LKM? Similar questions arise for the data addresses in the oops message.

To answer those questions, **ksymoops** must be able to get the loadpoints and lengths of the various sections of the LKM from the kernel symbol table.

Well, in Linux 2.4, **insmod** knows those addresses, so it just creates symbols for them and includes them in the symbols it loads with the LKM.

In particular, those symbols are named (and you can see this for yourself by looking at `/proc/ksyms`):

```
__insmod_name_Ssectionname_Llength
```

`name` is the LKM name (as you would see in `/proc/modules`.

`sectionname` is the section name, e.g. `.text` (don't forget the leading period).

`length` is the length of the section, in decimal.

The value of the symbol is, of course, the address of the section.

Insmod also adds a pretty useful symbol that tells from what file the LKM was loaded. That symbol's name is

```
__insmod_name_Ofilespec_Mmtime_Vversion
```

`name` is the LKM name, as above.

`filespec` is the file specification that was used to identify the file containing the LKM when it was loaded. Note that it isn't necessarily still under that name, and there are multiple file specifications that might have been used to refer to the same file. For example, `../dir1/mylkm.o` and `/lib/dir1/mylkm.o`.

`mtime` is the modification time of that file, in the standard Unix representation (seconds since 1969), in hexadecimal.

`version` tells the kernel version level for which the LKM was built (same as in the `.modinfo` section). It is the value of the macro `LINUX_VERSION_CODE` in Linux's `linux/version.h` file. For example, `132101`.

The value of this symbol is meaningless.

In Linux 2.6, it works differently. (I haven't figured out how yet).

10.5. Other Symbols

insmod adds another symbol, similar to the **ksymoops** symbols. This one tells where the persistent data lives in the LKM, which **rmmod** needs to know in order to save the persistent data.

```
__insmod_name_Plength
```

10.6. Debugging Symbols

There is another kind of symbol that relates to an LKM: kallsyms symbols. These are not exported symbols; they do not show up in `proc/ksyms`. They refer to addresses in the kernel that are nobody's business except the module they are in, and are not meant to be referenced by anything except a debugger. Kdb, the kernel debugger that comes with the Linux kernel, is one user of kallsyms symbols.

The kallsyms facility works for both the base kernel and LKMs. For the base kernel, you select it when you build the base kernel, with the CONFIG_KALLSYMS configuration option. When you do that, the kernel contains a kallsyms symbol for all the symbols in the base kernel object files. You know your base kernel is participating in the kallsyms facility if you see the symbol `__start___kallsyms` in `/proc/ksyms`.

For an LKM, you decide at load time whether it will contain kallsyms symbols. You include the kallsyms definitions in the data you pass to the `init_module` system call to load the LKM. **insmod** does this if either 1) you specify the `--kallsyms` option, or 2) **insmod** determines, by looking at `/proc/ksyms`, that the base kernel is participating in the kallsyms facility. The kallsyms that **insmod** defines are all the symbols in the LKM object file. To wit, those are the symbols you see when you run **nm** on the LKM object file.

Each loaded LKM that is participating in kallsyms has its own kallsyms symbol table. When the base kernel is participating in the kallsyms facility, the individual LKM kallysms symbol tables are linked into a master symbol table so that a debugger can look up a symbol anywhere in the kernel. When the base kernel is not participating in kallsyms, a debugger must look explicitly at a particular LKM to find symbols for that LKM. Kdb, for one, cannot do this. So the basic rule is: If you're going to do any kernel debugging, use CONFIG_KALLSYMS.

Note that the `__kallsyms` section has nothing to do with LKMs. That's a section in the base kernel object module. The base kernel doesn't have the luxury of something as high−level as sophisticated as **insmod** to load it, so it needs that extra object file section to facilitate its participation in kallsyms.

Similarly, the program **kallsyms** has nothing to do with LKMs. It is what creates the `__kallsyms` section.

There is another kind of debugging symbol −− the kind that **gcc** creates with its `-g` option. These are unrelated to the kallsyms facility. They do not get loaded into kernel memory. Kdb does not use them. But Kgdb (which gets information both from kernel memory and source and object files) does.

10.7. Memory Allocation For Loading

This section is about how Linux allocates memory in which to load an LKM. It is not about how an LKM dynamically allocates memory, which is the same as for any other part of the kernel.

The memory where an LKM resides is a little different from that where the base kernel resides. The base kernel is always loaded into one big contiguous area of real memory, whose real addresses are equal to its virtual addresses. That's possible because the base kernel is the first thing ever to get loaded (besides the loader) −− it has a wide open empty space in which to load. And since the Linux kernel is not pageable, it stays in its homestead forever.

By the time you load an LKM, real memory is all fragmented −− you can't simply add the LKM to the end of the base kernel. But the LKM needs to be in contiguous virtual memory, so Linux uses `vmalloc` to allocate a contiguous area of virtual memory (in the kernel address space), which is probably not contiguous in real memory. But *the memory is still not pageable*. The LKM gets loaded into real page frames from the start, and stays in those real page frames until it gets unloaded.

This design is based on the principle that it's much easier to get a large contiguous area of virtual memory than to get a large contiguous area of real memory because there is orders of magnitude more virtual address space than real memory. In the early days of virtual memory, that was certainly true –– a 32 bit machine had a 4GiB virtual address space and rarely more than 64 MiB of real memory. Now, however, it's quite often just the reverse, with virtual address space being the constrained resource. So this LKM loading strategy may not make sense on your system.

Some CPUs can take advantage of the properties of the base kernel to effect faster access to base kernel memory. For example, on one machine, the entire base kernel is covered by one page table entry and consequently one entry in the translation lookaside buffer (TLB). Naturally, that TLB entry is virtually always present. For LKMs on this machine, there is a page table entry for each memory page into which the LKM is loaded. Much more often, the entry for a page is not in the TLB when the CPU goes to access it, which means a slower access.

This effect is probably trivial.

It is also said that PowerPC Linux does something with its address translation so that transferring between accessing base kernel memory to accessing LKM memory is costly. I don't know anything solid about that.

The base kernel contains within its prized contiguous domain a large expanse of reusable memory –– the kmalloc pool. In some versions of Linux, the module loader tries first to get contiguous memory from that pool into which to load an LKM and only if a large enough space was not available, go to the vmalloc space. Andi Kleen submitted code to do that in Linux 2.5 in October 2002. He claims the difference is in the several per cent range.

10.8. Linux internals

If you're interested in the internal workings of the Linux kernel with respect to LKMs, this section can get you started. You should not need to know any of this in order to develop, build, and use LKMs.

The code to handle LKMs is in the source files `kernel/module.c` in the Linux source tree.

The kernel module loader (see Section 5.4) lives in `kernel/kmod.c`.

(Ok, that wasn't much of a start, but at least I have a framework here for adding this information in the future).

11. Writing Your Own Loadable Kernel Module

The Linux Kernel Module Programming Guide by Peter J Salzman, Michael Burian, and Ori Pomerantz is a complete explanation of writing your own LKM. This book is also available in print. There are two versions of it: one for Linux 2.4, and another for 2.6.

At one time, the Linux 2.4 version of this document was rather out of date and contained an error or two.

Here are a few things about writing an LKM that, at least at one time, weren't in there. Let the author of the LKM HOWTO know if it's still true. If not, he can remove this section from the LKM HOWTO.

11.1. Simpler hello.c

Lkmpg gives an example of the world's simplest LKM, hello−1.c. But it is not as simple as it could be and depends on your having kernel messaging set up a certain way on your system to see it work. Finally, the program requires you to include *−D* options on your compile command to work, because it does not define some macros in the source code, where the definitions belong.

Here is an improved world's simplest LKM, `hello.c`.

```
/* hello.c
 *
 * "Hello, world" - the loadable kernel module version.
 *
 * Compile this with
 *
 *          gcc -c hello.c -Wall
 */

/* Declare what kind of code we want from the header files */
#define __KERNEL__         /* We're part of the kernel */
#define MODULE             /* Not a permanent part, though. */

/* Standard headers for LKMs */
#include <linux/modversions.h>
#include <linux/module.h>

#include <linux/tty.h>      /* console_print() interface */

/* Initialize the LKM */
int init_module()
{
  console_print("Hello, world - this is the kernel speaking\n");
  /* More normal is printk(), but there's less that can go wrong with
     console_print(), so let's start simple.
  */

  /* If we return a non zero value, it means that
   * init_module failed and the LKM can't be loaded
   */
  return 0;
}


/* Cleanup - undo whatever init_module did */
void cleanup_module()
```

```
{
  console_print("Short is the life of an LKM\n");
}
```

Compile this with the simple command

```
$ gcc -c -Wall -nostdinc -I /usr/src/linux/include hello.c
```

The *-I* above assumes that you have the source code from which your base kernel (the base kernel of the kernel into which you hope to load `hello.c`) was built in the conventional spot, `/usr/src/linux`. If you're masochistic enough to be using symbol versioning in your base kernel, then you better have run 'make dep' on that kernel source too, because that's what builds the .ver files that change the names of all your symbols.

But note that it's reasonably common *not* to have the kernel headers installed there, and often, the *wrong* headers are installed there. When you use a kernel that you loaded from a distribution CD, you often have to separately load the headers for it. To be safe, if you're playing with compiling LKMs, you really should compile your own kernel, so you know exactly what you're working with and can be absolutely sure you're working with matching header files.

The *-nostdinc* option isn't strictly necessary, but is the right thing to do. It will keep you out of trouble and also remind you that the services of the standard C library, which you may have melded in your mind with C itself, are not available to kernel code. *-nostdinc* says not to include "standard" directories in the include file search path. This means, most notably, `/usr/include`.

The *-c* option says you just want to create an object (.o) file, as opposed to gcc's default which is to create the object file, then link it with a few other standard object files to create something suitable for exec'ing in a user process. As you will not be exec'ing this module but rather adding it to the kernel, that link phase would be entirely inappropriate.

-Wall (which makes the compiler warn you about lots of kinds of questionable code) is obviously not necessary, but this program should not generate any warnings. If it does, you need to fix something.

11.2. Using the Kernel Build System

Lkmpg contains fine instructions for building (compiling) an LKM (except that the __KERNEL__ macro and usually the MODULE macro should be defined in the source code instead of with *-D* compiler options as Lkmpg suggests). But it deserves mention that some Linux kernel programmers believe that the only right way to build an LKM is to add it to a copy of the complete Linux source tree and build it with the existing Linux make files just like the LKMs that are part of Linux.

There are advantages to this. The biggest one is that when Linux programmers change the way LKMs interface with the rest of the kernel in a way that affects how you build an LKM, you're covered.

On the other hand, you will probably find from a code management point of view that you really have to keep your own code and Linux separate, and from a coding point of view, you really need to understand all the intricacies of how your code gets compiled, especially when it changes.

11.3. Rubini et al: Linux Device Drivers

The most popular book on writing device drivers is O'Reilly's *Linux Device Drivers* by Alessandro Rubini, Jonathan Corbet, and Greg Kroah−Hartman.

Even if you're writing an LKM that isn't a device driver, you can learn a lot from this book that will help you.

The first edition of this book covers Linux 2.0, with notes about differences in 2.2. The second edition (June 2001) covers Linux 2.4. The third edition (April 2005) covers Linux 2.6. Of course, if you know anything about Linux, you know that a book like this doesn't perfectly cover any release, because Linux changes frequently. Linux 2.6 as was current a month after the Third Edition was released had significant differences from the Linux 2.6 about which the book was written.

The second edition of this book is available under the FDL. You can read it at http://www.xml.com/ldd/chapter/book/. The third edition is available under the terms of the Creative Commons Attribution−ShareAlike license, and you'll find it at http://lwn.net/Kernel/LDD3/.

This book is also available in print in any decent technical book store.

11.4. Module Use Counts

It is essential that the kernel not try to reference the code of a module after it has been unloaded; i.e. you must not unload a module while it is in use. An example of in use is a device driver for which a device special file is open. Because there is an open file descriptor for it, a user might do a read of the device and to execute that read, the kernel would want to call a function that is in the device driver. You can see that there would be a problem if you unloaded that device driver module before the read −− the kernel would reuse the memory that used to contain the read subroutine and there's no telling what instructions the kernel would branch to when it thinks it's calling the read subroutine.

In the original design, the LKM increments and decrements its use count to tell the module manager whether it is OK to unload it. For example, if it's a filesystem driver, it would increment the use count when someone mounts a filesystem of the type it drives, and decrement it at unmount time.

Later a more flexible alternative was added. Your LKM can register a function that the module manager will call whenever it wants to know if it is OK to unload the module. If the function returns a `true` value, that means the LKM is busy and cannot be unloaded. If it returns a `false` value, the LKM is idle and can be unloaded. The module manager holds the big kernel lock from before calling the module−busy function until after its cleanup subroutine returns or sleeps, and unless you've done something odd, that should mean that your LKM cannot become busy between the time that you report "not busy" and the time you clean up.

So how do you register the module−busy function? By putting its address in the unfortunately named `can_unload` field in the module descriptor ("struct module"). The name is truly unfortunate because the boolean value it returns is the exact opposite of what "can unload" means: true if the module manager *cannot* unload the LKM.

The module manager ensures that it does not attempt to unload the module before its initialization subroutine has returned or sleeps, so you are safe in setting the `can_unload` field anywhere in the initialization subroutine except after a sleep.

`can_unload` is little known and rarely used. Starting with Linux 2.6, it no longer exists.

Whether you use traditional use counts of `can_unload`, there are cases where you cannot be sure that your module doesn't get unloaded while it is still in use. If your LKM creates a kernel thread that executes LKM code, it is just about impossible to be absolutely sure that thread is gone before the LKM gets unloaded. There are various other kernel services that you can give addresses within your LKM that won't properly let you know when they have forgotten them.

The problem used to be worse than it is now. For example, it used to be that if your LKM created a proc filesystem file, you couldn't stop the LKM from getting unloaded while some process was executing your read and write routines for the file. This and other instances of the problem have been fixed by having code *outside* the LKM understand that the address it's using might be in an LKM, and therefore increment and decrement the use count as necessary. Where this function is implemented, you often see a structure member named "owner" which is a handle for the LKM (i.e. a struct module address).

These problems may be fixed in future version of Linux. Until then, you can just cross your fingers. Some people believe these types of problems are so hard to fix that the proper design for Linux is just to make it impossible ever to unload an LKM. Starting with Linux 2.6, the CONFIG_MODULE_UNLOAD kernel build configuration option determines whether module unloading is allowed or not.

---

12. Differences Between Versions Of Linux

One thing that deserves mention in this section is the variety of Linux versions that exist in the world and what we call them. Unlike a proprietary software product where one company carefully controls the name and creates a small number of well defined releases, variations of Linux are developed by lots of different independent people and all of them are called Linux.

The most basic Linux releases are controlled by Linus Torvalds and distributed by kernel.org as the main Linux releases. They are the only releases that can properly by called "Linux 2.4," "Linux 2.6.6," etc.

But hardly anybody uses those releases. Instead, people start with those releases and make modifications. People often sloppily refer to a Linux based on Linux 2.6.6 as Linux 2.6.6 itself. But to be correct, you have to add something −− usually a hyphen and a suffix. Red Hat versions of Linux, which you see a lot, unfortunately use just a plain number for that suffix, e.g. Linux 2.6.6−12. (It would be better if they used something more explicitly Red Hat, such as Linux 2.6.6−rh12).

Remember that in this document, "Linux" means the kernel; when we consider the operating systems called "Linux", the situation gets even more complicated.

12.1. Linux 2.4 − Linux 2.6

12.1.1. Linking Done In Kernel

The biggest change to LKMs between Linux 2.4 and Linux 2.6 is an internal one: LKMs get loaded much differently. Most people won't see any difference except that the suffix on a file containing an LKM has changed, because they use high level tools to manage the LKMs and the interface to those tools hasn't changed.

Before Linux 2.6, a user space program would interpret the ELF object (.o) file and do all the work of linking it to the running kernel, generating a finished binary image. The program would pass that image to the kernel and the kernel would do little more than stick it in memory. In Linux 2.6, the kernel does the linking. A user space program passes the contents of the ELF object file directly to the kernel. For this to work, the ELF object image must contain additional information. To identify this particular kind of ELF object file, we name the file with suffix ".ko" ("kernel object") instead of ".o" For example, the serial device driver that in Linux 2.4 lived in the file `serial.o` in Linux 2.6 lives in the file `serial.ko`.

So there is a whole new modutils package for use with Linux 2.6. In it, **insmod** is a trivial program, as compared to the full blown linker of the Linux 2.4 version.

Also, the procedure to build an LKM is somewhat harder. To make a .ko file, you start with a regular .o file. You run the program **modpost** (which comes with the Linux source code) on it to create a C source file that describes the additional sections the .ko file needs. We'll call this the .mod file because you conventionally include ".mod" in the file name.

You compile the .mod file and link the result with the original .o file to make a .ko file.

The .mod object file contains the name that the LKM instance will have when you load the LKM. You set that name with a −D compile option (when you compile the .mod file) that sets the KBUILD_MODNAME macro.

This change means some things are decidedly harder –– choosing the name for the LKM instance, for example. In Linux 2.4, the name was one of the inputs to the kernel. **insmod** decided on the name and passed it to the kernel. **insmod**'s −o option told it explicitly what to use for the LKM instance name. But in 2.6, there is no such parameter on the system call and hence no −o option on **insmod**. The name is part of the ELF object (.o file) that you pass to the kernel. The default name is built into the ELF object, but if you want to load it with some other name, you must edit the ELF image before passing it to **insmod**.

12.1.2. No Module Busy Function

In Linux 2.6 `can_unload` (see Section 11.4) is gone.

12.1.3. CONFIG_MODULE_UNLOAD

You can configure the kernel build to build a kernel that does not allow unloading of modules at all, thus sidestepping any problems with modules that get unloaded while still in use. See Section 11.4.

12.1.4. Reference Counting

The interface that the code of an LKM uses to manipulate its reference count has been replaced.

13. Copyright Considerations With LKMs

A perennial question about LKMs is whether the terms of GPL apply to one, considering that the Linux kernel is distributed under GPL. For example, is it OK to ship an LKM in binary only form? This section covers the copyright issues surrounding LKMs, which are as interesting as they are complex. Fair warning: we don't reach a conclusion. The law is unsettled and there are plenty of good but conflicting arguments.

Remember that countries have varying copyright laws, so even if you figure out the answer to a question in one country, it might be different in another. But copyright laws are actually remarkably similar, and this section won't go into the matter in enough detail that it will really make a difference.

Let's start at the beginning and look at what a copyright is.

Primarily, a copyright is a person's right to stop other people from copying something. It is a legal right, not a moral one. That means it is created by law to bring about some practical effect, rather than something that people believe people are naturally entitled to.

The primary effect that copyright seeks to bring about is that an author gets paid for what he created. Some people believe that is a valid goal itself, because an author has a natural right to the value that he creates. But historically, that isn't the real goal of copyright. The real goal is secondary to making the author get paid for his work: It causes the author to create in the first place. An author is more likely to spend time and money writing if he will get paid for it.

The actual operation of copyright law achieves that goal only approximately. We've all seen cases where the law is used to transfer wealth in ways that don't contribute to that goal at all. For example, a music publisher denies a person permission to copy a song that he wouldn't have paid for anyway. Legislators think of that as collateral damage –– what copyright law does to cause things to get written makes up for the senseless copy restrictions in other cases.

But when it comes to LKMs, there is another, much more complex area of copyright law that matters: derivative works. Copyright law gives an author the right to stop someone from creating a derivative work. A derivative work is *not* a copy of anything the author wrote. So what is it?

The definition of derivative work is elusive, but here are some examples: When you translate a book from English into French, the French version is a derivative work of the English work. If you write a new chapter for a novel, your new chapter is a derivative work of the novel. If you write a whole new book for the Harry Potter series, with the same characters and settings, that is a derivative work of all the Harry Potter books. If you draw a Dilbert cartoon you thought of on a birthday card, that is a derivative work of all the Dilbert strips, books, etc.

And this brings us to LKMs, because many people believe that an LKM is a derivative work of the Linux kernel. The Free Software Foundation has said so. Linus Torvalds has said it is sometimes. No court of law has ever ruled one way or the other.

Now let's turn to the copyright on Linux. For the sake of discussion, when we say "Linux," we will mean the contents of the tarball you download from kernel.org as the regular Linux kernel.

Who owns the copyright on Linux? Lots of people. Nearly everyone who has contributed code to Linux reserved his own copyright. Some of that work was done for hire and therefore the employing corporations own the copyright. Linus Torvalds is the most visible copyright owner, but he holds copyright on a very small part of Linux.

How do these people prosecute their copyright? Well, nobody makes deals with them to purchase copying rights. It wouldn't be practical. However, they all offer a copyright license −− the same one −− to the public. The license is documented and offered in a file in the tarball, and is known as the General Public License (GPL). GPL is not specific to Linux. It was developed by the Free Software Foundation before Linux existed, and was simply chosen by Linus back when he was the only copyright owner. And Linus doesn't put any code into the tarball without the author offering the same license.

A copyright license is permission by a copyright owner to do something that he has a legal right to stop you from doing, such as make a copy or a derivative work. What it means to be a public license is that it is offered to the public, as opposed to the copyright owner offering it to particular persons he knows about.

The GPL lets you do almost anything with the code −− it's almost like the copyright owners waiving all their copyright rights. But not quite: there are strings attached. The license has conditions. In order to have the permission, you have to meet the conditions. The condition we care about here is one of two things, or both, depending on how you read the license document (the document is ambiguous). Either a) if you distribute a derivative work of Linux, you have to supply source code for the whole derivative work; or b) if you distribute a work containing Linux, you have to supply source code for the entire work.

Now we come to the real question: what sort of rights do you have to distribute an LKM you wrote?

Certainly, many people believe that you can distribute an LKM under any terms you like, including binary−only and the owners of the Linux copyright have nothing to say about it. You're distributing nothing but your own code. And it's not a derivative work of anything. Many people do distribute binary−only LKMs and believe that a principle benefit of LKMs is the ability to do that.

Others think differently: The LKM, while an original work, is a derivative work of Linux. The Linux copyright owners have the right to control distribution. The only permission they gave you to distribute it is GPL, and that permission is only under the condition that you supply source code.

Let's look a little closer at the issue of whether an LKM is a derivative work of Linux. The argument in favor goes that writing an LKM is like writing a supplemental chapter for a novel. We know the latter is creating a derivative work. The LKM is like the chapter because it's specifically designed to be part of a whole with the existing Linux kernel. It has no use in any other context, and when deployed, is tightly woven into the rest of Linux. The fact that you typically use a bunch of Linux header files to compile it is proof that it's just an extension of Linux (don't confuse this with another argument −− that the #include actually means you're distributing that header file inside your object code). And note that a kernel module loaded at run time is essentially the same program you would statically bind into the base kernel if you chose to go that route. If writing a module for the Linux source tree is creating a derivative work, then so must be writing an LKM. LKMs often have to be updated to conform to updates the base kernel.

Wasabi Systems, a company that sells kernels with freer licensing than Linux, has published a paper that argues in more detail that LKMs are derivative works of Linux and you should therefore be wary of publishing binary−only LKMs for it.

The argument against says that an LKM is something that interacts with Linux, not something that is part of Linux. It likens the LKM to a user space program, communicating with the kernel via system calls, or an FTP client program (which would not be a derivative work of any FTP server program).

The issue also runs into one of those areas where books and computer programs aren't analogous because a computer program does something, whereas a book just communicates an idea to a person. Loading an LKM might be more like plugging an attachment into your vacuum cleaner than like inserting pages into your book. And we know that the blueprints for a vacuum cleaner attachment are not a derivative work of the blueprints for a vacuum cleaner.

So that's as far as I can go. There appears to be a matter of degree here and a judge will have to draw a line somewhere.

But if you need to make a practical decision, consider that there have been well−known binary LKMs (drivers for Nvidia video adapters seem to be the most famous) for years and no one has sued for copyright infringement. Also, Linus Torvalds, influential for reasons other than legal, has said binary−only LKMs are OK with him.

What about GPL−ONLY symbols? Kernel developers have selected some symbols that one uses in interfacing an LKM to the base kernel as GPL−only. These symbols have "GPL_ONLY" in their names, making this intent obvious. Furthermore, in order for the Linux module loader to let your LKM use them, you must include some code in your LKM that supposedly certifies you license your module under GPL.

These probably have no legal significance. If the LKM is not a derivative work of Linux, then the kernel developers simply have no legal way to block you from putting the GPL annotation in your code and distributing it binary−only anyhow. If the LKM *is* a derivative work of Linux, then the absence of the GPL−only classification of the symbol is probably not enough to give permission to use it in a binary−only LKM. The license document does not mention them. At best, you can use GPL−only symbols as a gentleman's promise not to sue you for use of all the other symbols in a binary−only LKM.

---

14. Related Documentation

For modules that are part of Linux (i.e. distributed with the base kernel), you can sometimes find documentation in the `Documentation` subdirectory of the Linux source code.

Many LKMs can be alternatively bound into the base kernel. If you do that, you will pass parameters to them via the kernel "command line," which in its most basic form means via a prompt at boot time. *The BootPrompt HOWTO* by Paul Gortmaker `<Paul.Gortmaker@anu.edu.au>` will help you with that. It is available from the Linux Documentation Project.

Don't forget that the source code of Linux and any LKM is always the documentation of last resort, and the most trustworthy.

---

15. Individual Modules

In this chapter, I document individual LKMs. Where possible, I do this by reference to more authoritative documentation for the particular LKM (probably maintained by the same person who maintains the LKM code).

15.1. Executable Interpreters

You must have at least one executable interpreter bound into the base kernel, because in order to load an executable interpreter LKM, you have to run an executable and something has to interpret that executable.

That one bound−in executable interpreter is almost certainly the ELF interpreter, since virtually all executables in a Linux system are ELF.

Historical note: Before ELF existed on Linux (c. 1995), the normal executable format was a.out. For a while, part ELF/part a.out systems were common. Some still exist.

15.1.1. binfmt_aout: executable interpreter for a.out format

a.out is the venerable executable format that was common in Unix's early history and originally Linux's only executable format. To this day, the default name of the executable output file of the GNU compiler is `a.out` (regardless of what it's format is).

If you try to run an a.out executable without this, your `exec` system call fails with a "cannot execute binary file" error.

There are no LKM parameters.

Example:

```
modprobe binfmt_aout
```

15.1.2. binfmt_elf: executable interpreter for ELF format

ELF is the normal executable format on Linux systems.

It's almost inconceivable that you wouldn't have this executable interpreter bound into the base kernel (if for no other reason that your **insmod** is probably an ELF executable). However, it is conceptually possible to leave it out of the base kernel and insert it as an LKM.

There are no LKM parameters.

Example:

```
modprobe binfmt_elf
```

15.1.3. binfmt_java: executable interpreter for Java bytecode

Java is a relatively modern object oriented programming language. Java programs are traditionally compiled into "Java bytecode" which is meant to be interpreted by a Java bytecode interpreter. The point of this new object language is that the bytecode object files are portable: Although different systems require different object formats, as long as each system has a bytecode interpreter, it can run bytecode object files. (This only works for a while, of course. If portability were that easy, all systems today would use the same object format anyway).

While the intent was that the bytecode interpreter would run as a user space program, with this LKM you can make the Linux kernel interpret Java bytecode like any other executable format. So you can run a program compiled from Java the same as you would run a program compiled from C (e.g. type its name at a command shell prompt).

In practice, the advantages of the intermediate bytecode language have not been proven and it is quite common to compile Java directly to a more traditional executable format, such as ELF. If you do that, you don't need **binfmt_java**.

There are no LKM parameters.

Example:

```
modprobe binfmt_java
```

15.2. Block Device Drivers

15.2.1. floppy: floppy disk driver

This is the device driver for floppy disks. You need this in order to access a floppy disk in any way.

This LKM is documented in the file `README.fd` in the `linux/drivers/block directory` of the Linux source tree. For detailed up to date information refer directly to this file.

Note that if you boot (or might boot) from a floppy disk or with a root filesystem on a floppy disk, you must have this driver bound into the base kernel, because your system will need it before it has a chance to insert the LKM.

Example:

```
  modprobe floppy 'floppy="daring two_fdc 0,thinkpad 0x8,fifo_depth"'
```

There is only one LKM parameter: *`floppy`*. But it contains many subparameters. The reason for this unusual parameter format is to be consistent with the way you would specify the same things in the kernel boot parameters if the driver were bound into the base kernel.

The value of *`floppy`* is a sequence of blank−delimited words. Each of those words is one of the following sequences of comma−delimited words:

asus_pci
: Sets the bit mask of allowed drives to allow only units 0 and 1. Obsolete, as this is the default setting

anyways

daring
: Tells the floppy driver that you have a well behaved floppy controller. This allows more efficient and smoother operation, but may fail on certain controllers. This may speed up certain operations.

0,daring
: Tells the floppy driver that your floppy controller should be used with caution.

one_fdc
: Tells the floppy driver that you have only floppy controller (default).

`address`,two_fdc
: Tells the floppy driver that you have two floppy controllers. The second floppy controller is assumed to be at `address`. This option is not needed if the second controller is at address 0x370, and if you use the 'cmos' option

two_fdc
: Like above, but with default address

thinkpad
: Tells the floppy driver that you have an IBM Thinkpad model notebook computer. Thinkpads use an inverted convention for the disk change line.

0,thinkpad
: Tells the floppy driver that you don't have a Thinkpad.

nodma
: Tells the floppy driver not to use DMA for data transfers. This is needed on HP Omnibooks, which don't have a workable DMA channel for the floppy driver. This option is also useful if you frequently get "Unable to allocate DMA memory" messages. Indeed, DMA memory needs to be continuous in physical memory, and is thus harder to find, whereas non−DMA buffers may be allocated in virtual memory. However, I advise against this if you have an FDC without a FIFO (8272A or 82072). 82072A and later are OK). You also need at least a 486 to use nodma. If you use nodma mode, I suggest you also set the FIFO threshold to 10 or lower, in order to limit the number of data transfer interrupts.

  If you have a FIFO−able FDC, the floppy driver automatically falls back on non DMA mode if it can't find any DMA−able memory. If you want to avoid this, explicitly specify "yesdma".

omnibook
: Same as *`nodma`*.

yesdma
: Tells the floppy driver that a workable DMA channel is available (the default).

nofifo
: Disables the FIFO entirely. This is needed if you get "Bus master arbitration error" messages from your Ethernet card (or from other devices) while accessing the floppy.

fifo
: Enables the FIFO (default)

`threshold`,fifo_depth
: Sets the FIFO threshold. This is mostly relevant in DMA mode. If this is higher, the floppy driver tolerates more interrupt latency, but it triggers more interrupts (i.e. it imposes more load on the rest of the system). If this is lower, the interrupt latency should be lower too (faster processor). The benefit of a lower threshold is fewer interrupts.

  To tune the fifo threshold, switch on over/underrun messages using 'floppycontrol −−messages'. Then access a floppy disk. If you get a huge amount of "Over/Underrun − retrying" messages, then the fifo threshold is too low. Try with a higher value, until you only get an occasional Over/Underrun.

  The value must be between 0 and 0xf, inclusive.

As you insert and remove the LKM to try different values, remember to redo the 'floppycontrol −−messages' every time you insert the LKM. You shouldn't normally have to tune the fifo, because the default (0xa) is reasonable.

`drive`,`type`,cmos
: Sets the CMOS type of `drive` to `type`. This is mandatory if you have more than two floppy drives (only two can be described in the physical CMOS), or if your BIOS uses non−standard CMOS types. The CMOS types are:

    0
    : Use the value of the physical CMOS

    1
    : 5 1/4 DD

    2
    : 5 1/4 HD

    3
    : 3 1/2 DD

    4
    : 3 1/2 HD

    5
    : 3 1/2 ED

    6
    : 3 1/2 ED

    16
    : unknown or not installed

    (Note: there are two valid types for ED drives. This is because 5 was initially chosen to represent floppy *tapes*, and 6 for ED drives. AMI ignored this, and used 5 for ED drives. That's why the floppy driver handles both)

unexpected_interrupts
: Print a warning message when an unexpected interrupt is received. (default behavior)

no_unexpected_interrupts
: Don't print a message when an unexpected interrupt is received. This is needed on IBM L40SX laptops in certain video modes. (There seems to be an interaction between video and floppy. The unexpected interrupts only affect performance, and can safely be ignored.)

L40SX
: Same as *`no_unexpected_interrupts`*.

broken_dcl
: Don't use the disk change line, but assume that the disk was changed whenever the device node is reopened. Needed on some boxes where the disk change line is broken or unsupported. This should be regarded as a stopgap measure, indeed it makes floppy operation less efficient due to unneeded cache flushings, and slightly more unreliable. Please verify your cable, connection and jumper settings if you have any DCL problems. However, some older drives, and also some laptops are known not to have a DCL.

debug
: Print debugging messages

messages
: Print informational messages for some operations (disk change notifications, warnings about over and underruns, and about autodetection)

silent_dcl_clear
: Uses a less noisy way to clear the disk change line (which doesn't involve seeks). Implied by daring.

`nr`,irq
: Tells the driver to expect interrupts on IRQ `nr` instead of the conventional IRQ 6.

`nr`,dma
: Tells the driver to use DMA channel `nr` instead of the conventional DMA channel 2.

slow
: Use PS/2 stepping rate: PS/2 floppies have much slower step rates than regular floppies. It's been recommended that take about 1/4 of the default speed in some more extreme cases.

`mask`,allowed_drive_mask
: Sets the bitmask of allowed drives to `mask`. By default, only units 0 and 1 of each floppy controller are allowed. This is done because certain non−standard hardware (ASUS PCI motherboards) mess up the keyboard when accessing units 2 or 3. This option is somewhat obsoleted by the 'cmos' option.

all_drives
: Sets the bitmask of allowed drives to all drives. Use this if you have more than two drives connected to a floppy controller.

15.2.2. loop: loop device driver

This module lets you mount a filesystem that is stored in a regular file (in another filesystem). That other file is called the backing file.

One use of this is to test an ISO 9660 filesystem before irreversibly burning it onto a CD. You build the filesystem in a 650 MB regular file. That file will be the input to the CD burning program. But you can define a loopback device based on that file as backing file and then mount the filesystem right from the backing file.

It can also give you a handy way to transmit collections of files over a network. It's like a tar file, only you don't have to pack and unpack it −− you just mount the original file.

Some people use loop devices on a machine that sometimes runs Windows and sometimes runs Linux to allow them to maintain the Linux system via the Windows system: put a Linux root filesystem in a file in a FAT filesystem that Windows can access, then mount the Linux root filesystem via a loop device when Linux is running.

You can keep the filesystem encrypted or compressed, or encoded in any arbitrary way, in the backing file. The loop device encodes (e.g. encrypts) as you write to it, and decodes (e.g. decrypts) as you read. (An alternative more popular strategy for encrypting and compressing a filesystem is to use an encrypted or compressed filesystem type, either a native one or one backed by a normal filesystem. Cfs, Tcfs, and Stegfs are examples of such filesystem types).

An encoding system is based on a "transfer function". There are two tranfer functions built into the `loop` module: the identify transfer function (which is for the normal no−encoding case −− What you see in the loop device is exactly what is in the backing file) and a simple XOR encryption function. A separate kernel module can add any transfer function by calling the `loop` module's exported `loop_register_transfer()` function.

There appear to be various modules floating around that provide transfer functions to do compression and encryption (DES, IDEA, Fish, etc.). Some of them appear to be part of current Linux kernel distributions. In addition, there appear to be various alternative loop device drivers, many of them also called `loop`, that have such transfer functions built in.

Do not confuse these loop devices with the "loopback device" used for network connections from the machine to itself. That isn't actually a device at all – it's a network interface.

This module is a block device driver. You set up a loop device by issuing an ioctl to it to bind a file to it. The typical program to issue this ioctl is **losetup**. See the documentation of **losetup** for more details. There are also options on the normal 'mount' command to do loop device setup under the covers, but because that confuses the logically separate operations of setting up a loop device and mounting a filesystem, for the sake of clarity you're probably better off using **losetup**.

Example:

```
modprobe loop
```

Module Parameters:

max_loop
: Number of loop devices that will exist. Contrary to what its name suggests, the number you specify is the number of loop devices that always exist. An existing device is not necessarily configured (bound to a backing file), though, so this number can be thought of as the maximum number of loop devices that you can configure.

  The minor numbers for these loop devices are consecutive starting at 0.

There is more information on loop devices in the Loopback Encrypted Filesystem HOWTO and the Loopback Root Filesystem HOWTO and the manual for **losetup**.

15.2.3. linear: linear (non−RAID) disk array device driver

This driver lets you combine several disk partitions into one logical block device.

If you use this, then your multiple devices driver will be able to use the so−called linear mode, i.e. it will combine the disk partitions by simply appending one to the other.

See *Software−RAID−HOWTO*.

Example:

```
modprobe linear
```

There are no module parameters.

15.2.4. raid0: RAID−0 device driver

This driver lets you combine several disk partitions into one logical block device.

If you use this, then your multiple devices driver will be able to use the so−called raid0 mode, i.e. it will combine the disk partitions into one logical device in such a fashion as to fill them up evenly, one chunk here and one chunk there. This will increase the throughput rate if the partitions reside on distinct disks.

See *Software−RAID−HOWTO*.

Example:

```
modprobe raid0
```

There are no module parameters.

15.2.5. rd: ramdisk device driver

A ramdisk is a block device whose storage is composed of system memory (real memory; not virtual). You can use it like a very fast disk device and also in circumstances where you need a device, but don't have traditional hardware devices to play with.

A common example of the latter is for a rescue system –– a system you use to diagnose and repair your real system. Since you don't want to mess with your real disks, you run off ramdisks. You might load data into these ramdisks from external media such as floppy disks.

Sometimes, you have your boot loader (e.g. **lilo**) create a ramdisk and load it with data (perhaps from a floppy disk). Of course, if you do this, you cannot use the LKM version of the ramdisk driver because the driver will have to be in the kernel at boot time.

A ramdisk is actually conceptually simple in Linux. Disk devices operate through memory because of the buffer cache. The only difference with a ramdisk is that you never actually get past the buffer cache to a real device. This is because with a ramdisk, 1) when you first access a particular block, Linux just assumes it is all zeroes; and 2) the device's buffer cache blocks are never written to the device, ergo never stolen for use with other devices. This means reads and writes are always to the buffer cache and never reach the device.

There is additional information about ramdisks in the file `Documentation/ramdisk.txt` in the Linux source tree.

Example:

```
  modprobe rd
```

There are no module parameters that you can supply to the LKM, but if you bind the module into the base kernel, there are kernel parameters you can pass to it. See *BootPrompt−HOWTO*.

15.2.6. xd: XT disk device driver

Very old 8 bit hard disk controllers used in the IBM XT computer. No, the existence of XT disk support does NOT mean that you can run Linux on an IBM XT :).

Example:

```
modprobe xd
```

There are no module parameters.

15.3. SCSI Drivers

Detailed information about SCSI drivers is in *SCSI−2.4−HOWTO*.

Linux's SCSI function is implemented in three layers, and there are LKMs for all of them.

In the middle is the mid−level driver or SCSI core. This consists of the **scsi_mod** LKM. It does all those things that are common among SCSI devices regardless of what SCSI adapter you use and what class of device (disk, scanner, CD−ROM drive, etc.) it is.

There is a low−level driver for each kind of SCSI adapter −− typically, a different driver for each brand. For example, the low−level driver for Advansys adapters (made by the company which is now Connect.com) is named **advansys**. (If you are comparing ATA (aka IDE) and SCSI disk devices, this is a major difference −− ATA is simple and standard enough that one driver works with all adapters from all companies. SCSI is less standard and as a result you should have less confidence in any particular adapter being perfectly compatible with your system).

High−level drivers present to the rest of the kernel an interface appropriate to a certain class of devices. The SCSI high−level driver for tape devices, **st**, for example, has ioctls to rewind. The high−level SCSI driver for CD−ROM drives, **sr**, does not.

Note that you rarely need a high−level driver specific to a certain brand of device. At this level, there is little room for one brand to be distinguishable from another.

One SCSI high−level driver that deserves special mention is **sg**. This driver, called the "SCSI generic" driver, is a fairly thin layer that presents a rather raw representation of the SCSI mid−level driver to the rest of the kernel. User space programs that operate through the SCSI generic driver (because they access device special files whose major number is the one registered by **sg** (to wit, 21)) have a detailed understanding of SCSI protocols, whereas user space programs that operate through other SCSI high−level drivers typically don't even know what SCSI is. *SCSI−Programming−HOWTO* has complete documentation of the SCSI generic driver.

The layering order of the SCSI modules belies the way the LKMs depend upon each other and the order in which they must be loaded. You always load the mid−level driver first and unload it last. The low−level and high−level drivers can be loaded and unloaded in any order after that, and they hook themselves into and establish dependency on the mid−level driver at both ends. If you don't have a complete set, you will get a "device not found" error when you try to access a device.

Most SCSI low−level (adapter) drivers don't have LKM parameters; they do generally autoprobe for card settings. If your card responds to some unconventional port address you must bind the driver into the base kernel and use kernel "command line" options. See *BootPrompt−HOWTO*. Or you can twiddle The Source and recompile.

Many SCSI low−level drivers have documentation in the `drivers/scsi` directory in the Linux source tree, in files called `README.*`.

15.3.1. scsi_mod: SCSI mid−level driver

Example:

```
modprobe scsi_mod
```

There are no module parameters.

15.3.2. sd_mod: SCSI high−level driver for disk devices

Example:

```
modprobe sd_mod
```

There are no module parameters.

15.3.3. st: SCSI high−level driver for tape devices

Example:

```
modprobe st
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

15.3.4. sr_mod: SCSI high−level driver for CD−ROM drives

Example:

```
modprobe sr_mod
```

There are no module parameters.

15.3.5. sg: SCSI high−level driver for generic SCSI devices

See the explanation of this special high−level driver above.

Example:

```
modprobe sg
```

There are no module parameters.

15.3.6. wd7000: SCSI low−level driver for 7000FASST

Example:

```
modprobe wd7000
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver atoprobes the card and requires installed BIOS.

15.3.7. aha152x: SCSI low−level driver for Adaptec AHA152X/2825

Example:

```
modprobe aha152x
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver atoprobes the card and requires installed BIOS.

15.3.8. aha1542: SCSI low−level driver for Adaptec AHA1542

Example:

```
modprobe aha1542
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card at 0x330 and 0x334 only.

15.3.9. aha1740: SCSI low−level driver for Adaptec AHA1740 EISA

Example:

```
modprobe aha1740
```

There are no module parameters.

This driver autoprobes the card.

15.3.10. aic7xxx: SCSI low−level driver for Adaptec AHA274X/284X/294X

Example:

```
modprobe aic7xxx
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card and BIOS must be enabled.

15.3.11. advansys: SCSI low−level driver for AdvanSys/Connect.com

Example:

```
modprobe advansys asc_iopflag=1 asc_ioport=0x110,0x330 asc_dbglvl=1
```

Module Parameters:

asc_iopflag

  1
    enable port scanning
  0
    disable port scanning

asc_ioport
  I/O port addresses to scan for Advansys SCSI adapters

asc_dbglvl
  debugging level:

  0
    Errors only

1
: High level tracing

2−N
: Verbose tracing

If you bind this driver into the base kernel, you can pass parameters to it via the kernel boot parameters. See *BootPrompt−HOWTO*.

15.3.12. in2000: SCSI low−level driver for Always IN2000

Example:

```
modprobe in2000
```

There are no module parameters.

This driver autoprobes the card. No BIOS is required.

15.3.13. BusLogic: SCSI low−level driver for BusLogic

The list of BusLogic cards this driver can drive is long. Read file `drivers/scsi/README.BusLogic` in the Linux source tree to get the total picture.

Example:

```
modprobe BusLogic
```

There are no module parameters.

If you bind this driver into the base kernel, you can pass parameters to it via the kernel boot parameters. See *BootPrompt−HOWTO*.

15.3.14. dtc: SCSI low−level driver for DTC3180/3280

Example:

```
modprobe dtc
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card.

15.3.15. eata: SCSI low−level driver for EATA ISA/EISA

This driver handles DPT PM2011/021/012/022/122/322.

Example:

```
modprobe eata
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

15.3.16. eata_dma: SCSI low−level driver for EATA−DMA

This driver handles DPT, NEC, AT&T, SNI, AST, Olivetti, and Alphatronix.

This driver handles DPT Smartcache, Smartcache III and SmartRAID.

Example:

```
modprobe eata_dma
```

There are no module parameters.

Autoprobe works in all configurations.

15.3.17. eata_pio: SCSI low−level driver for EATA−PIO

This driver handles old DPT PM2001, PM2012A.

Example:

```
  modprobe eata_pio
```

There are no module parameters.

15.3.18. fdomain: SCSI low−level driver for Future Domain 16xx

Example:

```
modprobe fdomain
```

There are no module parameters.

This driver autoprobes the card and requires installed BIOS.

15.3.19. NCR5380: SCSI low−level driver for NCR5380/53c400

Example:

```
modprobe NCR5380 ncr_irq=xx ncr_addr=xx ncr_dma=xx ncr_5380=1   \
                 ncr_53c400=1
```

for a port mapped NCR5380 board:

```
modprobe g_NCR5380 ncr_irq=5 ncr_addr=0x350 ncr_5380=1
```

for a memory mapped NCR53C400 board with interrupts disabled:

```
modprobe g_NCR5380 ncr_irq=255 ncr_addr=0xc8000 ncr_53c400=1
```

Parameters:

ncr_irq
: the irq the driver is to service. 255 means no or DMA interrupt. 254 to autoprobe for an IRQ line if overridden on the command line.

ncr_addr

the I/O port address or memory mapped I/O address, whichever is appropriate, that the driver is to drive

ncr_dma
: the DMA channel the driver is to use

ncr_5380
: 1 = set up for a NCR5380 board

ncr_53c400
: 1 = set up for a NCR53C400 board

If you bind this driver into the base kernel, you can pass parameters to it via the kernel boot parameters. See *BootPrompt−HOWTO*.

15.3.20. NCR53c406a: SCSI low−level driver for NCR53c406a

Example:

```
modprobe NCR53c406a
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

15.3.21. 53c7,8xx.o: SCSI low−level driver for NCR53c7,8xx

Example:

```
modprobe 53c7,8xx
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card and requires installed BIOS.

15.3.22. ncr53c8xx: SCSI low−level driver for PCI−SCS NCR538xx family

Example:

```
modprobe ncr53c8xx
```

There are no module parameters.

15.3.23. ppa: low−level SCSI driver for IOMEGA parallel port ZIP drive

See the file `drivers/scsi/README.ppa` in the Linux source tree for details.

Example:

```
modprobe ppa ppa_base=0x378 ppa_nybble=1
```

Parameters:

ppa_base
: Base address of the PPA's I/O port. Default 0x378.

ppa_speed_high

Delay used in data transfers, in microseconds. Default is 1.

ppa_speed_low
: Delay used in other operations, in microseconds. Default is 6.

ppa_nybble
: 1 = Use 4−bit mode. 0 = don't. Default is 0.

15.3.24. pas16: SCSI low−level driver for PAS16

Example:

```
modprobe pas16
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card. No BIOS is required.

15.3.25. qlogicfas: SCSI low−level driver for Qlogic FAS

Example:

```
modprobe qlogicfas
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

15.3.26. qlogicisp: SCSI low−level driver for Qlogic ISP

Example:

```
modprobe qlogicisp
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

Requires firmware.

15.3.27. seagate: SCSI low−level driver for Seagate, Future Domain

This driver is for Seagate ST−02 and Future Domain TMC−8xx.

Example:

```
  modprobe seagate
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes for address only. The IRQ is fixed at 5. The driver requires installed BIOS.

15.3.28. t128: SCSI low−level driver for Trantor T128/T128F/T228

Example:

```
modprobe t128
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card. The driver requires installed BIOS.

15.3.29. u14−34f: SCSI low−level driver for UltraStor 14F/34F

Example:

```
modprobe u14-34f
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

This driver autoprobes the card, but *not* the 0x310 port. No BIOS is required.

15.3.30. ultrastor: low−level SCSI driver for UltraStor

Example:

```
modprobe ultrastor
```

There are no module parameters for the LKM, but if you bind this module into the base kernel, you can pass some parameters via the Linux boot parameters. See *BootPrompt−HOWTO*.

15.4. Network Device Drivers

15.4.1. bsd_comp: optional BSD compressor for PPP

Example:

```
modprobe bsd_comp
```

There are no module parameters.

This module depends on module **ppp**.

15.4.2. slhc: SLHC compressor for PPP

This module contains routines to compress and uncompress tcp packets (for transmission over low speed serial lines).

These routines are required by PPP (also ISDN−PP) and SLIP protocols, and are used by the LKMs that implement those protocols.

Example:

```
modprobe slhc
```

There are no module parameters.

15.4.3. dummy: Dummy network interface driver

This is said to be a bit−bucket device (i.e. traffic you send to this device is consigned into oblivion) with a configurable IP address. It is most commonly used in order to make your currently inactive SLIP address seem like a real address for local programs.

However, it also functions as a sort of loopback device. You configure it for a particular IP address and any packet you send to that IP address via this interface comes back and appears as a packet received by that interface for that IP address. This is especially handy for an IP address that would normally be reflected by another interface (a PPP interface, perhaps), but that interface is down right now.

You can have multiple dummy interfaces. They are named `dummy0`, `dummy1`, etc.

Example:

```
modprobe dummy
```

There are no module parameters.

15.4.4. eql: serial line load balancer

If you have two serial connections to some other computer (this usually requires two modems and two telephone lines) and you use PPP (a protocol for sending internet traffic over telephone lines) or SLIP (an older alternative to PPP) on them, you can make them behave like one double speed connection using this driver.

Example:

```
modprobe eql
```

There are no module parameters.

15.4.5. dlci: frame relay DLCI driver

This implements the frame relay protocol; frame relay is a fast low−cost way to connect to a remote internet access provider or to form a private wide area network. The one physical line from your box to the local "switch" (i.e. the entry point to the frame relay network) can carry several logical point−to−point connections to other computers connected to the frame relay network. To use frame relay, you need supporting hardware (FRAD) and certain programs from the net− tools package as explained in `Documentation/networking/framerelay.txt` in the Linux source tree.

Example:

```
modprobe dlci
```

There are no module parameters.

15.4.6. sdla: Sangoma S502A FRAD driver

This is a driver for the Sangoma S502A, S502E and S508 Frame Relay Access Devices. These are multi−protocol cards, but this driver can drive only frame relay right now. Please read `Documentation/networking/framerelay.txt` in the Linux source tree.

Example:

```
modprobe sdla
```

There are no module parameters.

This module depends on module **dlci**.

15.4.7. plip: PLIP network interface driver

PLIP (Parallel Line Internet Protocol) is used to create a mini network consisting of two (or, rarely, more) local machines. The parallel ports (the connectors virtually all ISA−descendant computers have that are normally used to attach printers) are connected using "null printer" or "Turbo Laplink" cables which can transmit 4 bits at a time or using special PLIP cables, to be used on bidirectional parallel ports only, which can transmit 8 bits at a time. The cables can be up to 15 meters long. This works also if one of the machines runs DOS/Windows and has some PLIP software installed, e.g. the Crynwr PLIP packet driver and winsock or NCSA's **telnet**.

See *PLIP−Install−HOWTO*.

Example:

```
modprobe plip io=0x378 irq=7
```

Parameters:

io
: Port address of parallel port driver is to drive.

irq
: IRQ number of IRQ driver is to service. Default is IRQ 5 for port at 0x3bc, IRQ 7 for port at 0x378, and IRQ 9 for port at 0x278.

If you don't specify the *`io`* parameter, the driver probes addresses 0x278, 0x378, and 0x3bc.

15.4.8. ppp: PPP network protocol driver

PPP (Point to Point Protocol) is the most common protocol to use over a serial port (with or without a modem attached) to create an IP network link between two computers.

Along with this kernel driver, you need the user space program **pppd** running.

See *PPP−HOWTO*.

Example:

```
modprobe ppp
```

There are no module parameters.

This module depends on module **slhc**.

The module also accesses serial devices, which are driven by the **serial** module, so it depends on that module too. This dependency is not detected by **depmod**, so you either have to declare it manually or load **serial** explicitly.

15.4.9. slip: SLIP network protocol driver

SLIP (Serial Line Internet Protocol) is like PPP, only older and simpler.

Example:

```
modprobe slip slip_maxdev=1
```

Parameters:

slip_maxdev
: Maximum number of devices the driver may use at one time. Default is 256.

This module depends on module **slhc**.

The module also accesses serial devices, which are driven by the **serial** module, so it depends on that module too. This dependency is not detected by **depmod**, so you either have to declare it manually or load **serial** explicitly.

15.4.10. baycom: BAYCOM AX.25 amateur radio driver

This is a driver for Baycom style simple amateur radio modems that connect to either a serial interface or a parallel interface. The driver works with the ser12 and par96 designs.

For more information, see http://www.baycom.org/~tom.

Example:

```
modprobe baycom modem=1 iobase=0x3f8 irq=4 options=1
```

Parameters:

major
: major number the driver should use; default 60

modem
: modem type of the first channel (minor 0):

    1
    : ser12

    2
    : par96/par97

iobase

base address of the port the driver is to drive. Common values are for ser12 0x3f8, 0x2f8, 0x3e8, 0x2e8 and for par96/par97 0x378, 0x278, 0x3bc.

irq
: IRQ the driver is to service. Common values are 3 and 4 for ser12 and 7 for for par96/par97.

options
: 0
  : use hardware DCD

  1
  : use software DCD

15.4.11. strip: STRIP (Metricom starmode radio IP) driver

STRIP is a radio protocol developed for the MosquitoNet project to send Internet traffic using Metricom radios. Metricom radios are small, battery powered, 100kbit/sec packet radio transceivers, about the size and weight of a wireless telephone. (You may also have heard them called "Metricom modems" but we avoid the term "modem" because it misleads many people into thinking that you can plug a Metricom modem into a phone line and use it as a modem.) You can use STRIP on any Linux machine with a serial port, although it is obviously most useful for people with laptop computers.

Example:

```
modprobe strip
```

There are no module parameters.

15.4.12. wavelan: WaveLAN driver

WaveLAN card are for wireless ethernet−like networking. This driver drives AT&T GIS and NCR WaveLAN cards.

Example:

```
modprobe wavelan io=0x390 irq=0
```

Parameters:

io
: Address of I/O port on the card. Default is 0x390. You can set a different address on the card, but it is not recommended.

irq
: IRQ the driver is to service. Default is 0. Any other value is ignored and the card still services IRQ 0.

15.4.13. wic: WIC Radio IP bridge driver

This is a driver for the WIC parallel port radio bridge.

Example:

```
modprobe wic
```

It appears that devices `wic0`, `wic1` and `wic2` are directly related to corresponding `lpN` ports.

15.4.14. scc: Z8530 SCC kiss emulation driver

These cards are used to connect your Linux box to an amateur radio in order to communicate with other computers. If you want to use this, read `Documentation/networking/z8530drv.txt` in the Linux kernel source tree and *HAM−HOWTO*.

Example:

```
modprobe scc
```

There are no module parameters.

15.4.15. 8390: General NS8390 Ethernet driver core

This is driver code for the 8390 Ethernet chip on which many Ethernet adapters are based. This is not a complete interface driver; the routines in this module are used by drivers for particular Ethernet adapters, such as **ne** and **3c503**.

Example:

```
modprobe 8390
```

There are no module parameters.

15.4.16. ne: NE2000/NE1000 driver

This is a driver for the venerable NE2000 Ethernet adapter, its NE1000 forerunner, and all the generic Ethernet adapters that emulate this de facto standard card. This is an ISA bus card. For the PCI version, see the ne2k−pci module.

Example:

```
modprobe ne io=0x300 irq=11
```

Parameters:

io
: Address of I/O port on the card. This parameter is mandatory, but you may specify 0x000 to have the driver autoprobe 0x300, 0x280, 0x320, 0x340, and 0x360.

irq
: IRQ the driver is to service. If you don't specify this, the driver determines it by autoIRQ probing.

bad
: The value 0xBAD means to assume the card is poorly designed in that it does not acknowledge a reset or does not have a valid 0x57,0x57 signature. If you have such a card and do not specify this option, the driver will not recognize it.

  With any other value, the option has no effect.

You can repeat the options to specify additional cards. The `n`th occurence of an option applies to the `n`th card.

This module depends on module **8390**.

15.4.17. ne2k−pci: NE2000 PCI Driver

This is a driver for the PCI version of the venerable NE2000 Ethernet adapter, and all the generic Ethernet adapters that emulate this de facto standard card.

Example:

```
modprobe ne io=0x300 irq=11
```

Parameters:

debug
: Level of debug messages. 0 means no messages. 1 is the default. Higher numbers mean more debugging messages.

options
: The value of this option determines what options are set in the network adapter. Each bit of the value, expressed as a binary number, controls one option. The only option defined is full duplex, which is the 6th least significant bit. It is much easier to use the *full_duplex* option instead.

full_duplex
: A "1" value sets the adapter in full duplex mode. A "0" value sets it in half duplex mode. If you include the full duplex flag in the flags you specify with the *options* parameter, the *full_duplex* has no effect.

You may repeat the *options* and *full_duplex* parameters once per network adapter, for up to 8 network adapter.

This driver can drive the following chipsets:

- RealTek RTL−8029
- Winbond 89C940
- Winbond W89C940F
- KTI ET32P2
- NetVin NV5000SC
- Via 86C926
- SureCom NE34
- Holtek HT80232
- Holtek HT80229
- Compex RL2000

This module depends on module **8390**.

15.4.18. 3c501: 3COM 3c501 Ethernet driver

This is a driver for 3COM's 3c501 Ethernet adapter.

Example: modprobe 3c501 io=0x280 irq=5

Parameters:

io

Address of I/O port on the card.

irq
: IRQ the driver is to service. Default is 5.

If you don't specify an I/O port, the driver probes addresses 0x280 and 0x300.

15.4.19. 3c503: 3COM 3c503 driver

This is a driver for 3COM's 3c503 Ethernet adapter.

Example:

```
modprobe 3c503 io=0x300 irq=5 xcvr=0
```

Parameters:

io
: Address of I/O port on the card.

irq
: IRQ the driver is to service.

xcvr
: Determines whether to use external tranceiver.

  0
  : no

  1
  : yes

If you don't specify an I/O port, the driver probes addresses 0x300, 0x310, 0x330, 0x350, 0x250, 0x280, 0x2A0, and 0x2E0.

This module depends on module **8390**.

15.4.20. 3c505: 3COM 3c505 driver

This is a driver for 3COM's 3c505 Ethernet adapter.

Example:

```
modprobe 3c503 io=0x300 irq=5 xcvr=0
```

Parameters:

io
: Address of I/O port on the card.

irq
: IRQ the driver is to service.

If you don't specify an I/O port, the driver probes addresses 0x300, 0x280, and 0x310.

This module depends on module **8390**.

15.4.21. 3c507: 3COM 3c507 driver

This is a driver for 3COM's 3c507 Ethernet adapter.

Example:

```
modprobe 3c503 io=0x300 irq=5 xcvr=0
```

Parameters:

io
: Address of I/O port on the card.

irq
: IRQ the driver is to service.

If you don't specify an I/O port, the driver probes addresses 0x300, 0x320, 0x340, and 0x280.

This module depends on module **8390**.

15.4.22. 3c509: 3COM 3c509/3c579 driver

This is a driver for 3COM's 3c507 and 3c579 Ethernet adapters.

Example:

```
modprobe 3c503 io=0x300 irq=5 xcvr=0
```

Parameters:

io
: Address of I/O port on the card.

irq
: IRQ the driver is to service.

Module load−time probing Works reliably only on EISA, ISA ID−PROBE IS NOT RELIABLE! Bind this driver into the base kernel for now, if you need it auto−probing on an ISA−bus machine.

15.4.23. 3c59x: 3COM 3c590 series "Vortex" driver

This is a driver for the following 3COM Ethernet adapters:

- 3c590 Vortex 10Mbps.
- 3c595 Vortex 100baseTX.
- 3c595 Vortex 100baseT4.
- 3c595 Vortex 100base−MII.
- EISA Vortex 3c597.

Example:

```
modprobe 3c59x debug=1 options=0,,12
```

Parameters:

debug
: A number selecting the level of debug messages.

options
: This is a string of options numbers separated by commas. There is one option number for each adapter that the driver drives (for the case that you have multiple Ethernet adapters in the system of types driven by this driver). The order of the option numbers is the order of the cards assigned by the PCI BIOS.

  Each number represents a binary value. In that value, the lower 3 bits is the media type:

  0
  : 10baseT

  1
  : 10Mbs AUI

  2
  : undefined

  3
  : 10base2 (BNC)

  4
  : 100base−TX

  5
  : 100base−FX

  6
  : MII (not yet available)

  7
  : Use default setting

  The next bit (the "8" bit) is on for full duplex, off for half.

  The next bit (the "16" bit) is on to enable bus−master, which is for experimental use only.

Details of the device driver implementation are at the top of the source file.

15.4.24. wd: Western Digital/SMC WD80*3 driver

This is a driver for the Western Digital WD80*3 Ethernet adapters.

Example:

```
modprobe wd io=0x300 irq=5 mem=0x0D0000 mem_end=0x0D8000
```

Parameters:

io
: Address of I/O port on the card.

irq
: IRQ the driver is to service.

mem
: Shared memory address

mem_end
: End of shared memory (address of next byte after it).

If you don't specify an I/O port, the driver probes 0x300, 0x280, 0x380, and 0x240.

If you don't specify an IRQ, the driver reads it from the adapter's EEPROM and with ancient cards that don't have it, the driver uses autoIRQ.

The driver depends on module **8390**.

15.4.25. smc−ultra: SMC Ultra/EtherEZ driver

This is a driver for the SMC Ultra/EtherEZ Ethernet adapters.

Example:

```
modprobe smc−ultra io=0x200 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x200, 0x220, 0x240, 0x280, 0x300, 0x340, and 0x380.

irq
: IRQ the driver is to service. Default is the value read from the adapter's EEPROM.

This driver depends on module **8390**.

15.4.26. smc9194: SMC 9194 driver

This is a driver for SMC's 9000 series of Ethernet cards.

Example:

```
modprobe smc9194 io=0x200 irq=5 ifport=0
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x200, 0x220, etc. up through 0x3E0.

irq
: IRQ the driver is to service.

ifport
: Type of Ethernet.

  0
  : autodetect

  1
  : TP

  2
  : AUI (or 10base2)

The debug level is settable in the source code.

15.4.27. at1700: AT1700 driver

This is a driver for the AT1700 Ethernet adapter.

Example:

```
modprobe at1700 io=0x260 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x260, 0x280, 0x2A0, 0x240, 0x340, 0x320, 0x380, and 0x300.

irq
: IRQ the driver is to service.

15.4.28. e2100: Cabletron E21xx driver

Example:

```
modprobe e2100 io=0x300 irq=5 mem=0xd0000 xcvr=0
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x300, 0x280, 0x380, and 0x220.

irq
: IRQ the card is to generate and the driver is to service. (The driver sets this value in the card).

mem
: shared memory address. Default is 0xd0000.

xcvr
: 0
  : Don't select external transceiver

  1
  : Select external transceiver

This module depends on module **8390**.

15.4.29. depca: DEPCA, DE10x, DE200, DE201, DE202, DE422 driver

This is a driver for the DEPCA, DE10x, DE200, DE201, DE202, and DE422 Ethernet adapters.

Example:

```
modprobe depca io=0x200 irq=7
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x300, and 0x200 on an

ISA machine or 0x0c00 on an EISA machine.

irq
: IRQ the driver is to service. Default is 7.

15.4.30. ewrk3: EtherWORKS 3 (DE203, DE204, DE205) driver

This is a driver for the EtherWORKS 3 (DE203, D3204, and DE205) Ethernet adapters.

Example:

```
modprobe ewrk3 io=0x300 irq=5
```

io
: Address of I/O port on the card. Default is 0x300.

irq
: IRQ the driver is to service. Default is 5.

On an EISA bus, this driver does EISA probing.

On an ISA bus, this driver does no autoprobing when loaded as an LKM. However, if you bind it into the base kernel, it probes addresses 0x100, 0x120, etc. up through 0x3C0 except 0x1E0 and 0x320.

15.4.31. eexpress: EtherExpress 16 driver

This is a driver for the EtherExpress 16 Ethernet adapter.

Example:

```
modprobe eexpress io=0x300 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x300, 0x270, 0x320, and 0x340. 1

irq
: IRQ the driver is to service. The default is the value read from the adapter's EEPROM.

15.4.32. eepro: EtherExpressPro driver

This is a driver for the EtherExpressPro Ethernet adapter.

Example:

```
modprobe eepro io=0x200 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x200, 0x240, 0x280, 0x2C0, 0x300, 0x320, 0x340, and 0x360.

irq
: IRQ the driver is to service.

15.4.33. fmv18k: Fujitsu FMV−181/182/183/184 driver

This is a driver for the Fujitsu FMV−181, FMV−182, FMV−183, FMV−183, and FMV−184 Ethernet adapters.

Example:

```
modprobe fmv18x io=0x220 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x220, 0x240, 0x260, 0x280, 0x2a0, 0x2c0, 0x300, and 0x340.

irq
: IRQ the driver is to service.

15.4.34. hp−plus: HP PCLAN+ (27247B and 27252A) driver

This is a driver for HP's PCLAN+ (27247B and 27252A) Ethernet adapters.

Example:

```
modprobe hp-plus io=0x200 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x200, 0x240, 0x280, 0x2C0, 0x300, 0x320, and 0x340.

irq
: IRQ the driver is to service. The default is the value the driver reads from the adapter's configuration register.

This module depends on module **8390**.

15.4.35. hp: HP PCLAN (27245, 27xxx) driver

This is a driver for HP's PCLAN (27245 and other 27xxx series) Ethernet adapters.

Example:

```
modprobe hp io=0x300 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes 0x300, 0x320, 0x340, 0x280, 0x2C0, 0x200, and 0x240.

irq
: IRQ the driver is to service. If you don't specify this, the driver determines it by autoIRQ probing.

This module depends on module **8390**.

15.4.36. hp100: HP 10/100VG PCLAN (ISA, EISA, PCI) driver

This is a driver for HP's 10/100VG PCLAN Ethernet adapters. It works with the ISA, EISA, and PCI versions.

Example:

```
modprobe hp100 hp100_port=0x100
```

Parameters:

hp100_port
: Base address of I/O ports on the card. If you don't specify this, the driver autoprobes 0x100, 0x120, etc. up through 0x3E0 on an ISA bus. It does EISA probing on an EISA bus.

15.4.37. eth16i: ICL EtherTeam 16i/32 driver

This is a driver for ICL's EtherTeam 16i (eth16i) and 32i (eth32i) Ethernet adapters.

Example:

```
modprobe eth16i io=0x2a0 irq=5
```

Parameters:

io
: Address of I/O port on the card. If you don't specify this, the adapter probes the following adddresses. For the eth16i adapter: 0x260, 0x280, 0x2A0, 0x340, 0x320, 0x380, and 0x300. For the eth32i: 0x1000, 0x2000, 0x3000, 0x4000, 0x5000, 0x6000, 0x7000, 0x8000, 0x9000, 0xA000, 0xB000, 0xC000, 0xD000, 0xE000, and 0xF000.

irq
: IRQ the driver is to service. If you don't specify this, the driver determines it by autoIRQ probing.

15.4.38. ni52: NI5210 driver

This is a driver for the NI5210 Ethernet adapter.

Example:

```
modprobe ni52 io=0x360 irq=9 memstart=0xd0000 memend=0xd4000
```

15.4.39. ac3200: Ansel Communications EISA 3200 driver

This is a driver for the Ansel Communications EISA 3200 Ethernet adapter.

Example:

```
modprobe ac3200
```

This module depends on module **8390**.

15.4.40. apricot: Apricot Xen−II on board ethernet driver

Example:

```
modprobe apricot io=0x300 irq=10
```

Parameters:

io
: address of base I/O port on card.

irq
: IRQ that driver is to service.

15.4.41. de4x5: DE425, DE434, DE435, DE450, DE500 driver

This is a driver for the DE425, DE434, DE435, DE450, and DE500 Ethernet adapters.

Example:

```
modprobe de4x5 io=0x000b irq=10 is_not_dec=0
```

Parameters:

io
: address of base I/O port.

irq
: IRQ the driver is to service.

is_not_dec
: For a non−DEC card using the DEC 21040, 21041, or 21140 chip, set this to 1.

15.4.42. tulip: DECchip Tulip (dc21x4x) PCI driver

Example:

```
modprobe tulip
```

Read `Documentation/networking/tulip.txt` in the Linux source tree.

15.4.43. dgrs: Digi Intl RightSwitch SE−X driver

This is a driver for the Digi International RightSwitch SE−X EISA and PCI boards. These boards have a 4 (EISA) or 6 (PCI) port Ethernet switch and a NIC combined into a single board.

There is a tool for setting up input and output packet filters on each port, called **dgrsfilt**.

The management tool lets you watch the performance graphically, as well as set the SNMP agent IP and IPX addresses, IEEE Spanning Tree, and Aging time. These can also be set from the command line when the driver is loaded.

There is also a companion management tool, called **xrightswitch**.

Examples:

```
modprobe dgrs debug=1 dma=0 spantree=0 hashexpire=300 ipaddr=199,86,8,221
modprobe ipxnet=111
```

Parameters:

debug
: Level of debugging messages to print

dma
: 0
  : Disable DMA on PCI card
  1
  : Enable DMA on PCI card

spantree
: 0
  : Disable IEEE spanning tree
  1
  : Enable IEEE spanning tree

hashexpire
: Change address aging time, in seconds. Defaults is 300.

ipaddr
: SNMP agent IP address. Value is IP address in dotted decimal notation, except with commas instead of periods.

ipxnet
: SNMP agent IPX network number

15.4.44. de600: D−Link DE600 pocket adapter driver

This is a driver for the D−Link DE600 pocket Ethernet adapter.

Example:

```
modprobe de600 de600_debug=0
```

Parameters:

de600_debug
: The driver expects the adapter to be at port 0x378 and generate IRQ 7. This is the same as the DOS `lpt1` device. These are compile time options.

15.4.45. de620: D−Link DE620 pocket adapter driver

This is a driver for the D−Link DE620 pocket Ethernet adapter.

Example:

```
modprobe de620 bnc=0 utp=0 io=0x378 irq=7
```

Parameters:

bnc

1
: Network is 10Base2

0
: Network is not 10Base2

utp

1
: Network is 10BaseT

0
: Network is not 10BaseT

io
: I/O port address of port driver is to drive. Default is 0x378.

irq
: IRQ driver is to service. Default is 7.

You can't specify both *bnc=1* and *utp=1*.

15.4.46. ibmtr: Tropic chipset based token ring adapter driver

Example:

```
modprobe ibmtr io=0xa20 irq=5
```

Parameters:

io
: I/O port address of port driver is to drive. Default is 0xa20.

irq
: IRQ driver is to service. By default, the driver determines the IRQ by autoIRQ probing.

15.4.47. arcnet: ARCnet driver

Read The Fine Information in `Documentation/networking/arcnet.txt` in the Linux source tree. Also Arcnet hardware information `arcnet-hardware.txt` is found in same place.

Example:

```
modprobe arcnet io=0x300 irq=2 shmem=0xd0000 device=arc1
```

Parameters:

io
: I/O port address of port driver is to drive. If you don't specify this, the driver probes addresses 0x300, 0x2E0, 0x2F0, 0x2D0, 0x200, 0x210, 0x220, 0x230, 0x240, 0x250, 0x260, 0x270, 0x280, 0x290, 0x2A0, 0x2B0, 0x2C0, 0x310, 0x320, 0x330, 0x340, 0x350, 0x360, 0x370, 0x380, 0x390, 0x3A0, 0x3E0, and 0x3F0.

irq
: IRQ driver is to service. By default, the driver determines the IRQ by autoIRQ probing.

device
: device name.

15.4.48. isdn: basic ISDN functions

This module provides ISDN functions used by ISDN adapter drivers.

Setting up ISDN networking is a complicated task. Read documentation found in `Documentation/isdn` in the Linux source tree.

Example:

```
modprobe isdn
```

There are no module parameters.

This module depends on module **slhc**.

15.4.49. icn: ICN 2B and 4B driver

This is a driver for the ICN 2B and ICN 4B ISDN adapters.

Example:

```
modprobe icn portbase=0x320 membase=0xd0000 icn_id=idstring icn_id2=idstring2
```

Parameters:

portbase
: Address of the base I/O port on the adapter. Defaults is 0x320.

membase
: Address of shared memory. Default is 0xd0000.

icn_id
: idstring for the first adapter. Must start with a character! This parameter is required.

icn_id2
: idstring for the second adapter. Must start with a character! This parameter is required with the double card.

This module depends on module **isdn**.

15.4.50. pcbit: PCBIT−D driver

This is a driver for the PCBIT−D ISDN adapter driver.

Example:

```
modprobe pcbit mem=0xd0000 irq=5
```

Parameters:

mem
: Shared memory address. Default is 0xd0000

irq
: IRQ the driver is to service. Default is 5.

This module depend on module **isdn**.

15.4.51. teles: Teles/NICCY1016PC/Creatix driver

This is a driver for the Teles/NICCY1016PC/Creatix ISDN adapter. It can drive up to 16 cards.

Example:

```
modprobe teles io=0xd0000,15,0xd80,2 teles_id=idstring
```

Parameters:

io
: This is a whole collection of parameters in one. It's syntax is *io*=`card1options` [,`card2options` ,...] where `card1options` is a set of options for the first card, etc.

  The syntax of `card1options`, etc. is `sharedmem, irq, portbase, dprotocol`

  sharedmem
  : Address of shared memory. Default 0xd0000

  irq
  : IRQ driver is to service.

  portbase
  : Address of base I/O port.

  dprotocol
  : D−channel protocol of the card

    1
    : 1TR6

    2
    : EDSS1. This is the default.

teles_id
: Driver ID for accessing with utilities and identification when using a line monitor. Value must start with a character! Default: none.

The driver determines the type of card from the port, irq and shared memory address:

- port == 0, shared memory != 0 −> Teles S0−8
- port != 0, shared memory != 0 −> Teles S0−16.0
- port != 0, shared memory == 0 −> Teles S0−16.3

This module depends on module **isdn**.

15.5. CDROM Device Drivers

15.5.1. aztcd: Aztech/Orchid/Okano/Wearnes/TXC/CDROM driver

This is a driver for the Aztech, Orchid, Okano, Wearnes, TXC, and CDROM devices (which have special non−SCSI non−ATA interfaces).

Example:

```
modprobe aztcd aztcd=0x340
```

Parameters:

aztcd
: address of base I/O port

Read `Documentation/cdrom/aztcd` in the Linux source tree for full information.

15.5.2. gscd: Goldstar R420 CDROM driver

This is a driver for the Goldstar R420 CDROM drive, which does not use either an ATA or SCSI interface.

Example:

```
modprobe gscd gscd=0x340
```

Parameters:

gscd
: address of base I/O port. Default is 0x340, which will work for most applications. You select the address of the drive with the PN801−1 through PN801−4 jumpers on the Goldstar Interface Card. Appropriate settings are: 0x300, 0x310, 0x320, 0x330, 0x340, 0x350, 0x360, 0x370, 0x380, 0x390, 0x3A0, 0x3B0, 0x3C0, 0x3D0, 0x3E0, and 0x3F0.

15.5.3. sbpcd: Sound Blaster CDROM driver

This is a driver for the Matsushita, Panasonic, Creative, Longshine, and TEAC CDROM drives that don't attach via ATA or SCSI.

Example:

```
modprobe sbpcd sbpcd=0x340
```

Parameters:

sbpcd
: address of base I/O port

An additional parameter is an SBPRO setting, as described in `Documentation/cdrom/sbpcd` in the Linux source tree.

15.5.4. mcd: Mitsumi CDROM driver

This is a driver for Mitsumi CDROM drives that don't attach via ATA or SCSI. It does not handle XA or multisession.

Example:

```
modprobe mcd mcd=0x300,11,0x304,5
```

Parameters:

mcd
: This is a comma separated list of i/o base addresses and IRQs, in pairs.

15.5.5. mcdx: Mitsumi XA/MultiSession driver

This driver is like **mcd**, only it has XA and multisession functions.

Example:

```
modprobe mcdx mcdx=0x300,11,0x304,5
```

15.5.6. optcd: Optics Storage DOLPHIN 8000AT CDROM driver

This is the driver for the so−called "dolphin" CDROM drive form Optics Storage, with the 34−pin Sony−compatible interface. For the ATA−compatible Optics Storage 8001 drive, you will want the ATAPI CDROM driver. The driver also seems to work with the Lasermate CR328A.

Example:

```
modprobe optcd optcd=0x340
```

Parameters:

optcd
: address of base I/O port

15.5.7. cm206: Philips/LMS CM206 CDROM driver

This is the driver for the Philips/LMS cm206 CDROM drive in combination with the cm260 host adapter card.

Example:

```
modprobe cm206 cm206=0x300,11
```

Parameters:

cm206
: The address of the base I/O port the driver is to drive and the IRQ the driver is to service, separated by a comma. It doesn't matter what order you put them in, and you may specify just one, in which case the other defaults.

15.5.8. sjcd: Sanyo CDR−H94A CDROM driver

Example:

```
modprobe sjcd sjcd_base=0x340
```

Parameters:

sjcd_base
: address of the base I/O port the driver is to drive. Default is 0x340.

The driver uses no IRQ and no DMA channel.

15.5.9. isp16: ISP16/MAD16/Mozart soft configurable cdrom driver

This is a driver for the ISP16 or MAD16 or Mozart soft configurable cdrom interface.

Example:

```
modprobe isp16 isp16_cdrom_base=0x340 isp16_cdrom_irq=3
         isp16_cdrom_dma=0 isp16_cdrom_type=Sanyo
```

Parameters:

isp16_cdrom_base
: address of base I/O port the driver is to drive. Valid values are 0x340, 0x320, 0x330, and 0x360.

isp16_cdrom_irq
: IRQ the driver is to service. Valid values are 0, 3, 5, 7, 9, 10, and 11.

isp16_cdrom_dma
: DMA channel the driver is to use with the device. Valid values are 0, 3, 5, 6, and 7.

isp16_cdrom_type
: Type of device being driven. Valid values are `noisp16`, `Sanyo`, `Panasonic`, `Sony` and `Mitsumi`. Note that these values are case sensitive.

15.5.10. cdu31a: Sony CDU31A/CDU33A CDROM driver

Example:

```
  modprobe cdu31a cdu31a_port=0x340 cdu31a_irq=5
```

Parameters:

cdu31a_port
: address of base I/O port the driver is to drive. This parameter is mandatory.

cdu31a_irq
: IRQ the driver is to service. If you don't specify this, the driver does not use interrupts.

15.5.11. sonycd535: Sony CDU535 CDROM driver

Example:

```
modprobe sonycd535 sonycd535=0x340
```

Parameters:

sonycd535
: address of the base I/O port the driver is to drive.

15.6. Filesystem Drivers

15.6.1. minix: Minix filesystem driver

Example:

```
modprobe minix
```

There are no module parameters.

15.6.2. ext: "Extended" filesystem driver

Example:

```
modprobe ext
```

There are no module parameters.

15.6.3. ext2: "Second extended" filessystem driver

Example:

```
modprobe ext2
```

There are no module parameters.

15.6.4. xiafs: xiafs filesystem driver

Example:

```
modprobe xiafs
```

There are no module parameters.

15.6.5. fat: DOS FAT filesystem functions

This module provides services for use by the MSDOS and VFAT filesystem drivers.

Example:

```
modprobe fat
```

There are no module parameters.

15.6.6. msdos: MSDOS filesystem driver

Example:

```
modprobe msdos
```

There are no module parameters.

This module depends on the module **fat**.

15.6.7. vfat: VFAT (Windows−95) filesystem driver

Example:

```
modprobe vfat
```

There are no module parameters.

This module depends on module **fat**.

15.6.8. umsdos: UMSDOS filesystem driver

This is a driver for the UMSDOS filesystem type, which is a unix style filesystem built on top of an MSDOS FAT filesystem.

Example:

```
modprobe vfat
```

There are no module parameters.

This module depends on the **fat** and **msdos** modules.

15.6.9. nfs: NFS filesystem driver

Example:

```
modprobe nfs
```

There are no module parameters.

15.6.10. smbfs: SMB filesystem driver

SMBFS is a filesystem type which has an SMB protocol interface. This is the protocol Windows for Workgroups, Windows NT or Lan Manager use to talk to each other. SMBFS was inspired by Samba, the program written by Andrew Tridgell that turns any unix host into a file server for DOS or Windows clients. See ftp://nimbus.anu.edu.au/pub/tridge/samba/ for this interesting program suite and lots of more information on SMB and NetBIOS over TCP/IP. There you also find explanation for concepts like netbios name or share.

To use SMBFS, you need a special mount program, which can be found in the ksmbfs package, found on ftp://ibiblio.org/pub/Linux/system/Filesystems/smbfs.

Example:

```
modprobe smbfs
```

There are no module parameters

15.6.11. ncpfs: NCP (Netware) filesystem driver

NCPFS is a filesystem type which has an NCP protocol interface, designed by the Novell Corporation for their NetWare product. NCP is functionally similar to the NFS used in the TCP/IP community. To mount a Netware filesystem, you need a special mount program, which can be found in the ncpfs package. Homesite for ncpfs is ftp.gwdg.de/pub/linux/misc/ncpfs, but Ibiblio and its many mirrors will have it as well.

Related products are Linware and Mars_nwe, which will give Linux partial NetWare Server functionality.

Mars_nwe can be found on ftp.gwdg.de/pub/linux/misc/ncpfs.

Example:

```
modprobe ncpfs
```

There are no module parameters.

This module depends on module **ipx**.

15.6.12. isofs: ISO 9660 (CDROM) filesystem driver

Example:

```
modprobe isofs
```

There are no module parameters.

15.6.13. hpfs: OS/2 HPFS filesystem driver

This filesystem driver for OS/2's HPFS filesystem provides only read−only access.

Example:

```
modprobe hpfs
```

There are no module parameters.

15.6.14. sysv: System V and Coherent filesystem driver

This is the implementation of the SystemV/Coherent filesystem type for Linux.

It implements all of

- Xenix FS
- SystemV/386 FS
- Coherent FS

Example:

```
modprobe sysv
```

There are no module parameters.

15.6.15. affs: Amiga FFS filesystem driver

Example:

```
modprobe affs
```

There are no module parameters.

15.6.16. ufs: UFS filesystem driver

Apparently for mounting disks with FreeBSD and/or Sun partitions. No documentation exists, apart from The Source.

This filesystem driver provides only read−only access.

Example:

```
modprobe ufs
```

There are no module parameters.

15.7. Miscellaneous Device Driver

15.7.1. misc: device driver for "miscellaneous" character devices

A whole bunch of device types that don't appear in large enough numbers on a system to deserve major numbers of their own share Major Number 10 and are collectively called "miscellaneous" character devices. This module provides the common interface to serve that major number, but there are individual drivers for the specific device types. Those drivers register themselves with this driver.

Example:

```
modprobe misc
```

There are no module parameters.

15.8. Serial Device Drivers

15.8.1. serial: serial communication port (UART) device driver

This driver drives conventional serial ports (UARTs), but not some of the specialized high performance multi−port devices.

NOTE: **serial** is required by other modules, such as **ppp** and **slip**. Also it is required by serial mice and accordingly by gpm. However this isn't the regular kind of dependency that is detected by module handling tools, so you must load **serial** manually.

Example:

```
modprobe serial
```

There are no module parameters.

15.8.2. cyclades: Cyclades async mux device driver

Example:

```
  modprobe cyclades
```

There are no module parameters.

The intelligent boards also need to have their firmware code downloaded to them. This is done via a user level application supplied in the driver package called stlload. Compile this program where ever you dropped the package files, by typing **make**. In its simplest form you can then type **stlload** in this directory and that will download firmware into board 0 (assuming board 0 is an EasyConnection 8/64 board). To download to an ONboard, Brumby or Stallion do:

Read the information in the file `Documentation/stallion.txt` in the Linux source tree.

15.8.3. stallion: Stallion EasyIO or EC8/32 device driver

The intelligent boards also need to have their firmware code downloaded to them. This is done via a user level application supplied in the driver package called stlload.

Read the information in the file `Documentation/stallion.txt` in the Linux source tree.

Example:

```
  modprobe stallion
```

There are no module parameters.

15.8.4. istallion: Stallion EC8/64, ONboard, Brumby device driver

The intelligent boards also need to have their firmware code downloaded to them. This is done via a user level application supplied in the driver package called stlload.

Read the information at /usr/src/linux/drivers/char/README.stallion.

Example:

```
modprobe istallion
```

There are no module parameters.

15.8.5. riscom8: SDL RISCom/8 card device driver

Example:

```
modprobe riscom8 iobase=0xXXX iobase1=0xXXX iobase2=...
```

This driver can drive up to 4 boards at time.

15.9. Parallel Device Drivers

15.9.1. lp: Parallel printer device driver

Example:

```
  modprobe lp.o io=0x378 irq=0
```

This driver probes ports 0x278, 0x378, and 0x3bc.

Note: loading **lp** without any parameters will grab all parallel ports.

15.10. Bus Mouse Device Drivers

15.10.1. atixlmouse: ATIXL busmouse driver

Example:

```
modprobe atixlmouse
```

There are no parameters.

This module depends on module **misc**.

15.10.2. busmouse: Logitech busmouse driver

Example:

```
modprobe busmouse
```

There are no module parameters.

This module depends on module **misc**.

15.10.3. msbusmouse: Microsoft busmouse driver

Example:

```
modprobe msbusmouse
```

There are no module parameters.

This module depends on module **misc**.

15.10.4. psaux: PS/2 mouse (aka "auxiliary device") driver

Example:

```
modprobe psaux
```

There are no module parameters.

This module depends on module **misc**.

15.11. Tape Device Drivers

For SCSI tape device drivers, see Section 15.3. There are no LKMs for QIC−02 tape devices, but there is a device driver you can bind into the base kernel.

15.11.1. ftape: floppy tape (QIC−80/Travan) device driver

Example:

```
modprobe ftape tracing=3
```

Optional parameter *tracing* can take following values

0
: bugs

1
: + errors

2
: + warnings

3
: + information

4
: + more information

5
: + program flow

6
: + fdc/dma info

7
: + data flow

8
: + everything else

The default is 3.

15.12. Watchdog Timers

15.12.1. WDT: WDT Watchdog timer device driver

Example:

```
modprobe wdt
```

There are no module parameters.

The device address is hardcoded as 0x240. The IRQ is hardcoded as 14.

This module depends on module **misc**.

15.12.2. softdog: Software Watchdog Timer

Example:

```
  modprobe softdog
```

There are no module parameters.

This module depends on module **misc**.

15.12.3. pcwd: Berkshire Products PC Watchdog Driver

Example:

```
modprobe pcwd
```

There are no module parameters.

This module depends on module **misc**.

15.13. Sound Device Drivers

Configuring sound is a complex task. Read the files in directory `Documention/sound` in the Linux source tree.

Example:

```
modprobe sound
```

Option: dma_buffsize=32768

16. Maintenance Of This Document

This HOWTO is enthusiastically maintained by Bryan Henderson `<bryanh@giraffe-data.com>`. If you find something incorrect or incomplete or can't understand something, Bryan wants to know so maybe the next reader can be saved the trouble you had.

The source for this document is DocBook SGML, and is available from the Linux Documentation Project.

17. History

I have derived this (in 2001) from the HOWTO of the same name by Laurie Tischler, dated 1997. While I have kept all of the information from that original document (where it is still useful), I have rewritten the presentation entirely and have added a lot of other information. The original HOWTO's primary purpose was to document LKM parameters.

The original HOWTO was first released (Release 1.0) June 20, 1996, with a second release (1.1) October 20, 1996.

The first release of Bryan's rewrite was in June 2001.

18. Copyright

Here is Lauri Tischler's copyright notice from the original document from which this is derived:

Notes

[1] For the pedantic, see Section 10.7.

[2] You probably know this type of disk as "IDE". Strictly speaking, IDE is an incorrect appelation. IDE refers to the "Integrated Drive Electronics" technology which all modern disk drives, notably including all SCSI disk drives, use. The first IDE drives in common usage were ATA, and the names kind of got confused. ATA, like SCSI, is a precise specification of electrical signals, commands, etc.