The Mike Saunders SCHOOL OF LINUX

Part 1: Looking to get a job in Linux? Take a seat at the front of the class and amass the information you need to get LPI certified. This issue: hardware.

Mike

Linux Professional Institute

Our expert

Mike Saunders has been writing about Linux for over a decade, and has installed more distros than he's had hot dinners.

How do you prove just how good your Linux knowledge is? To your friends and family, it's not so hard – set someone up with a Linux box, let them see you doing a few intricate operations at the command line and they'll soon be pretty convinced that you've got the nous. For companies, however, it's not so easy. They can't necessarily tell from a CV, covering letter or interview whether your brain is swelling with useful information on package management, the Linux boot process and so forth. That's something that has to be confirmed somewhere else.

Certified and bonafide

Fortunately, here in the Linux world we have an excellent scheme to do just that: LPI Certification. LPI stands for the Linux Professional Institute, a non-profit group that provides exams and qualifications for those seeking to work with Linux systems. There are three levels of certification available, the first of which covers general system administration, including configuring hardware, working at the command line, package management and handling processes.

In this series, we're going to set you up with all you need to know for the LPI 101 exam, thereby proving you have what it takes to look after Linux boxes in a business setting. If you've been using Linux for a while, you might find this installment somewhat familiar, but it's worth going through anyway in case there's a snippet of knowledge that you're missing.

Live and learn

At this point, it's also worth noting that LPI training materials tend to be based around conservative, long-life distros that don't change drastically every six months. Red Hat Enterprise Linux (RHEL) is a good example, but it's not cheap, so CentOS – a free rebuild of RHEL that's bootable from your **LXFDVD** – is an excellent base for your training. Another good choice is Debian, which tries to adhere to standards and remains stable for years at a time. Here, we'll use CentOS 5.5.

So, without further ado, let's get started! In this tutorial we'll be focusing on hardware, so we'll walk you through the process of finding out what devices are in your system, enabling/disabling drivers and more.

Section 1: Listing hardware

PCs are complicated beasts at the best of times, with designs that desperately try to look and feel modern, but remnants from the 1980s still lurking beneath the surface. Fortunately, open systems such as Linux provide all the tools we need to get lots of information about devices and peripherals. The starting points for this are the **/proc** and **/sys** directories. These are not real, 'tangible' folders in the same sense as your **home** directory, but rather virtual directories created by the kernel, which contains information about running processes and hardware devices. What are they for? Well, **/proc** is largely focused on supplying information about processes (read: running programs on the system), whereas **/sys** primarily covers hardware devices. However, there is a bit of overlap between the two.

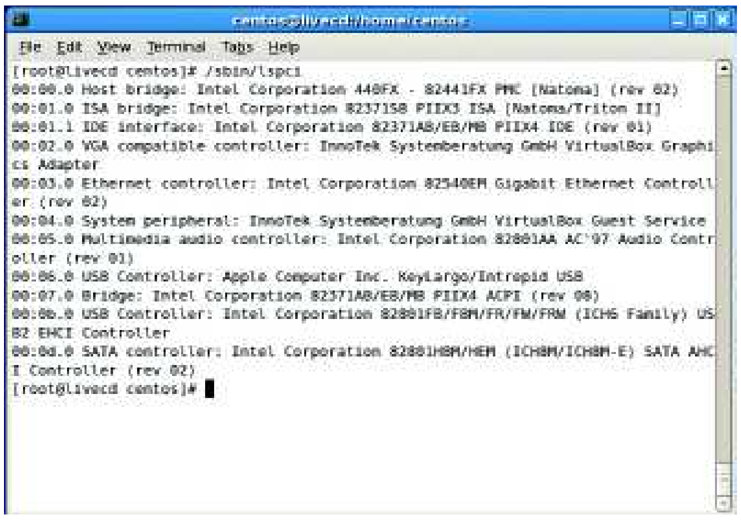
Sloppy lscpi

Most internal devices in your PC sit on a data transfer system called the PCI bus. On older distros, you can obtain information about devices using the command **cat /proc/pci**, but this file doesn't exist in newer distros. Instead, you could look inside **/sys/bus/pci/devices** – although you should be aware that this information isn't meant to be read by us mere mortals. Instead, the command we will use is:

```
/sbin/lspci
```

Open a terminal with Applications > Accessories > Terminal and switch to the root (admin) user by entering **su**. Then run the command above and you'll get a list of all hardware devices on the PCI bus in your machine, as shown in the image on the right. You should be able to see your video card, Ethernet adaptor and other devices. You can generate much more detail by adding **-vv** (dash v v) to the command, which will show information about interrupts and I/O (input/output) ports. If you're new to the world of PC hardware, then interrupts are effectively ways for a device to tell the CPU that it needs servicing – for instance, a soundcard telling the system that it has completed an operation. Meanwhile, I/O ports are for transmitting data to and from the device.

A PC has a finite number of interrupts (aka IRQs). While that was fine back in the days when most systems just had a monitor and keyboard to their name, today it's rather limiting. Consequently, IRQs can be shared across devices. You won't have to fiddle with the IRQs and I/O ports that a device needs – those days are long gone, thankfully – but you can always get a detailed list of hardware resources in use with the aforementioned **lspci -vv** command. More options to **lspci** are available, and you can find out more about these in the manual page (**man lspci**).



```
[root@Livecd centos]# /sbin/lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox Graphi
cs Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controll
er (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Contr
oller (rev 01)
00:06.0 USB Controller: Apple Computer Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0b.0 USB Controller: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) US
B2 EHCI Controller
00:0d.0 SATA controller: Intel Corporation 82801HBM/HEM (ICH8M/ICH8M-E) SATA AHC
I Controller (rev 02)
[root@Livecd centos]# █
```

› **The output of lspci, showing the hardware devices inside the machine.**

Section 2: Driver modules

What if you want to disable a device? Well, first of all we need to identify what enables a device in the first place: the hardware driver. In Linux, drivers can be enabled in two ways when compiling the kernel. The distro maker can either compile them directly into the kernel itself, or as standalone module files that the kernel loads when necessary. The latter approach is the norm, since it makes the kernel smaller, speeds up booting and makes the OS much more flexible too.

You can find your modules in **/lib/modules/<kernel version>/kernel**. These are KO files, and you'll see that they're sorted into directories for sound, filesystems (fs) and so on. If you go into the **Drivers** subdirectory, you'll see more categories of modules. It's important to make a distinction here between block and char (character) devices. The former is for hardware where data is transmitted in large blocks, such as hard drivers, whereas character devices stream data a byte or so at a time – for instance, mice and serial ports.

The Linux kernel is clever, and can load modules when it detects certain pieces of hardware. Indeed, it can load modules on demand when USB devices are plugged in – but we'll come to USB in a moment. In the meantime, let's look at how to manage modules. To get a list of all the modules that the kernel has currently loaded, enter this command as root:

```
lsmod
```

Note that in some distros, such as CentOS, you may need to prefix that with **/sbin/ – ie /sbin/lsmod**. You'll see a list, the exact contents of which will vary from system to system, depending on the hardware that you have installed.

You know my name

Now, the names of some of these modules will be immediately obvious to you, such as *cdrom* and *battery*. For certain modules, you'll see a list of Used By modules in the right-hand column. These are like dependencies in the package management world, and show which modules need other ones to be loaded beforehand.

What about those modules with cryptic names, though? What do they do exactly? Here's where the **modinfo** »

Quick tip

A cold-pluggable device needs to be plugged in when the machine is off. Adding and removing cold-pluggable devices (such as PS/2 mice and keyboards) when the machine is on can potentially damage chips on the motherboard.

What is /dev?

One of the core philosophies of Unix, and therefore Linux, is that everything is a file. Not just your documents and images, but hardware too. That sounds strange at first – how can a hardware device be represented as a file? Well, it makes sense at a fundamental level. A file is something which you can read information from and write it to. The same's true for a physical device, such as a hard drive: you can read bytes of data from it and write bytes of data to it. However, there are some devices (such as random number generators) that normally only work one way – for instance, they can be read, but you have no ability to send anything back.

The **/dev** directory contains hardware device nodes – files representing the devices. For example, **/dev/dvd** is your DVD-ROM drive. With a disc in the drive, you could enter **cat /dev/dvd** and it would spew out the binary data to your terminal. Device nodes are created automatically by the kernel, and some are placed in subdirectories such as **snd** (sound cards/chips), **input** (mice) and so on. There's a **/dev/null** device that simply eats data and destroys it, which you can use when you want to redirect output of a command so it doesn't show on the screen. There's also **/dev/mem**, a device for the machine's RAM. Running **strings /dev/mem | less** is a fascinating way to see what text your RAM chips are currently holding.

» command comes into play. For instance, it isn't at all clear what the *dm_mod* module does, but by running:

```
/sbin/modinfo dm_mod
```

We get a bunch of information. This is largely technical, but comes with a handy Description line that provides a smidgen of information about what the module does. Unfortunately, not every module has anything useful in this field, but it's worth trying if you're stumped about one's purpose.

As mentioned, many modules are loaded by the kernel automatically. You can also force one to be loaded with the **modprobe** command. This small utility is responsible for both loading and removing modules from the kernel, and is a very handy way to disable and enable kernel functionality on the fly. For instance, in our module list we see that there's *lp*, *parport* and *parport_pc*. These are for printers hooked up to the parallel port, which hardly anyone uses these days, so we can disable this functionality to free up a bit of RAM with:

```
/sbin/modprobe -r lp parport_pc parport
```

How do we know the right order to enter these? We can work it out using the Used By field mentioned before, placing the module that the first two depend on at the end of the command. So we remove the *lp* printer module, the *parport_pc* PC-specific parallel port driver and finally the generic parallel port driver.

Quick tip

You can find out more about LPI certification at the organisation's website: **www.lpi.org**. There you can find detailed lists of objectives covered in the various qualifications and, best of all, sample exam questions.

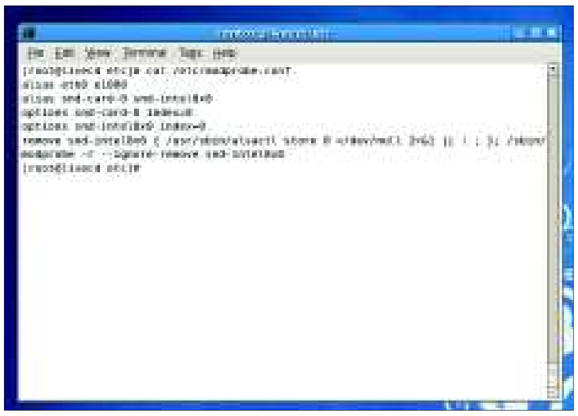
Probe deeper

Similarly, we can enable these modules again by using a plain **modprobe** command (without the **-r** remove flag). Because of the dependencies system, we need only specify the first in the list, and **modprobe** will work out what else it needs:

```
/sbin/modprobe lp
```

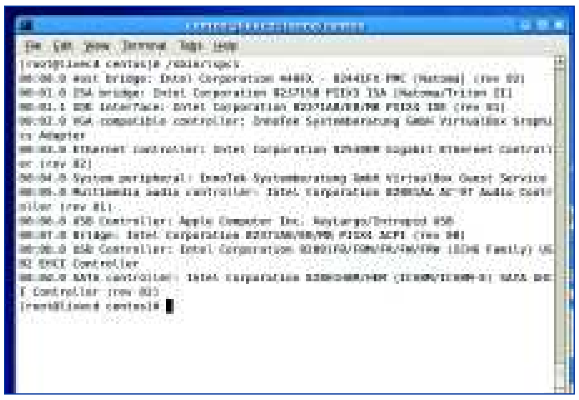
This also loads up *parport_pc* and *parport*, which we can confirm with a quick **lsmod** command.

While Linux typically handles modules automatically and with great aplomb, sometimes it's useful to have a bit of manual input in the process. We can do this via the **/etc/**



```
[root@[illegible] etc]# cat /etc/modprobe.conf
alias eth0 e1000
alias snd-card-0 snd-intel8x0
options snd-card-0 index=0
options snd-intel8x0 index=0
remove snd-intel8x0 { /usr/sbin/alsactl store 0 >/dev/null 2>&1 || : ; }; /sbin/
modprobe -r --ignore-remove snd-intel8x0
[root@[illegible] etc]#
```

› **An example /etc/modprobe.conf file in CentOS 5.5. Note the ability to expand upon remove commands in the last line.**



```
[root@[illegible] centos]# /sbin/lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox Graphi
cs Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controll
er (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Contr
oller (rev 01)
00:06.0 USB Controller: Apple Computer Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0b.0 USB Controller: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) US
B2 EHCI Controller
00:0d.0 SATA controller: Intel Corporation 82801HBM/HEM (ICH8M/ICH8M-E) SATA AHC
I Controller (rev 02)
[root@[illegible] centos]#
```

› **Listing driver modules with the lsmod command.**

modprobe.conf file. First up is aliases, a way to provide a shorthand term for a list of modules. For instance, you might want to be able to disable and enable your soundcard manually, but you can't always remember the specific module that it uses. You can add an **alias** line like this:

```
alias sound snd-ens1371
```

Now you can just enter **modprobe sound** and have your card working without having to remember the specific driver. Using this system, you can unify the commands you use across different machines. Then there's **options**, which enables you to pass settings to a module to configure the way it works. To find out which options are available for a particular module, use the **modinfo** command as described previously, looking for **parm** sections in the output.

For instance, when running **modinfo snd-intel8x0** we can see a list of **parm** sections that show options available for this sound chip module. One is called **index**. Our CentOS on VirtualBox **/etc/modprobe.conf** shows this in action with:

```
options snd-intel8x0 index=0
```

Custom commands

Lastly, we have the **install** and **remove** facilities. These are really powerful: they enable you to replace commands with different ones. For instance, in CentOS on *VirtualBox* we see:

```
remove snd-intel8x0 { /usr/bin/alsactl store 0...
```

The full line is much longer, but essentially it says: 'When the user or system runs **modprobe -r snd-intel8x0**, execute this command instead, beginning with **alsactl** – a volume control utility.' In this way, you can perform clean up and logging operations before the module removal takes place.

To prevent a module from loading entirely, simply alias it to **off** in **/etc/modprobe.conf**:

```
alias parport off
```

This will stop the module from ever being loaded, and therefore usually stop the hardware from being activated.

What are HAL, udev, D-Bus?

Desktop environments, such as Gnome and KDE, are abstracted from the nitty-gritty of hardware management. After all, Gnome hackers working on a photo management app don't want to write code to poke bytes down a USB cable to a camera – they want the OS to handle it. This makes sense and enables Gnome to run on other OSes. The HAL (hardware abstraction layer) daemon once provided this abstraction, but it's been replaced by *udev*, a background process that creates device nodes in **/dev** and interfaces with hardware.

How do programs interact with *udev*? They do this primarily via D-Bus, an inter-process communication (IPC) system which helps programs send messages to one another. For instance, a desktop environment can ask D-Bus to inform it if a new device is plugged in. D-Bus gets this information from *udev* when the user plugs in hardware and then informs the desktop so that it can pop up a dialog or launch an app.

Section 3: USB peripherals

If this article were written in the mid 1990s, we'd have to include long sections on the various ports sitting around in the back of a PC case. PS/2, AUX, serial, parallel… almost every device required its own connector and things were extremely messy. Thankfully, the situation is much simpler today with USB (Universal Serial Bus) – virtually every mainstream computer made in the last decade includes at least one USB port. The specification hasn't stayed still either: we've had USB 2.0 and 3.0 ramping up speeds to compete with other connectivity systems, such as FireWire.

When you plug in a USB device, the kernel initially probes it to find out what class it belongs to. USB devices are organised into these classes to facilitate driver development. There are classes for audio devices, printers, webcams, human interface devices (mice, keyboards, joysticks) and more. There's even a vendor-specific class for very specialised devices that don't fit into the normal categories and therefore require specific drivers to be installed.

ls + usb = lsusb

Linux's USB support is excellent, and there are many tools available for administrators to find out what's going on behind the scenes. First of all, as the USB controller typically sits on the USB bus, we can use our trusty **lspci** command to find out information about what type of USB controller we have:

```
/sbin/lspci | grep -i usb
```

Here, we're taking the output of **lspci** and piping it through to the **grep** utility, to search for all instances of the word USB in upper or lower-case. Don't worry if piping and **grep** are unfamiliar: we'll cover the command line in a later installment. Currently all you need to know is that this command filters the **lspci** output to just lines containing USB information.

After you've run the command, a few lines of information should appear, telling you the vendor and type of USB controller that you have. Slightly confusingly, there are two standard controller types for USB 1: UHCI and OHCI. USB 2.0 created EHCI, which layers on top of one of those. You don't need to worry about the differences between them, since the kernel handles this itself, but be aware that there's a bit of fragmentation in the USB world.

Like **lspci**, there's a command we can use to list all devices connected to our USB controller, and that's:

```
/sbin/lsusb
```

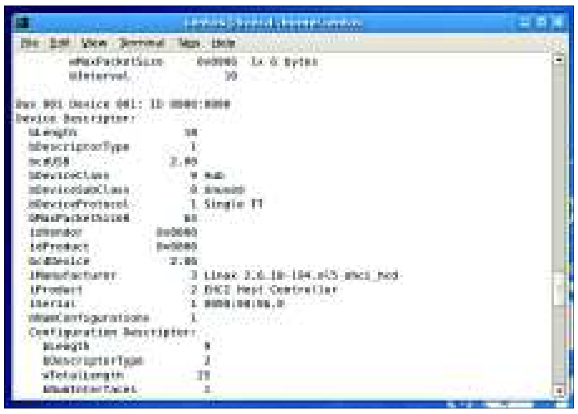
On its own, this command doesn't generate a great deal of information – just a list of device numbers and their positions on the USB bus. You can make it a bit more useful by adding **-t**, which shows the devices in a tree-like format and helps you see which are connected to which. However, by adding the **-v** flag we get much more verbose information, as shown in the screenshot on the left.

Look through the results and you can see information on both the USB controller you have and the devices connected to your box. If you're feeling particularly adventurous, go into **/sys/bus/usb/devices**, and there you'll see directories for each device, in which are files containing the manufacturer name, speed, maximum power usage and more.

As covered before, kernel modules are usually the method through which hardware devices are supported in Linux. This is also true for USB. Try this command, for instance:

```
/sbin/lsmod | grep hci
```

On our machine, it shows that the kernel has loaded a module for the OHCI controller, and also a module for EHCI USB 2.0 support along with that.



```
        wMaxPacketSize     0x0002  1x 2 bytes
        bInterval              10

Bus 001 Device 001: ID 0000:0000
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            9 Hub
  bDeviceSubClass         0 Unused
  bDeviceProtocol         1 Single TT
  bMaxPacketSize0        64
  idVendor           0x0000
  idProduct          0x0000
  bcdDevice            2.06
  iManufacturer           3 Linux 2.6.18-194.el5 ehci_hcd
  iProduct                2 EHCI Host Controller
  iSerial                 1 0000:00:0b.0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           25
    bNumInterfaces          1
```

› **The verbose output from the lsusb -v command.**

Dmesg in a bottle

A good way to determine how the kernel is recognising as USB device is with the **dmesg** command. This spits out a list of messages generated by the kernel since bootup. Run **dmesg**, then plug in a USB device, wait a few seconds for it to be recognised, and run **dmesg** again, noting the differences. Extra lines will be added to the bottom of the **dmesg** output, showing that the kernel has (hopefully) recognised the device and activated it. LXF

Hardware-less booting

We're all used to installing Linux on machines that have the essential peripherals: a keyboard, display and mouse. You can probably get away with ditching the mouse if you're familiar with the right kind of tab-space-enter combinations for your particular distro installer, but the other devices seem obligatory. Or are they? In a server environment, where your machine may be rack mounted and hard to access, you might have to install without hooking up these extra peripherals – and that's where network booting comes into play.

The magic to this method is PXE, Preboot Execution Environment. This is a bit of firmware on the computer that scans the network for an NBP (Network Bootstrap Program), which it then loads and executes. For this to work you'll need functioning DHCP and TFTP servers on your network, with the latter serving up the appropriate boot files for the distro. If your machine's BIOS doesn't support PXE, there's still another option – USB booting. You can boot a rudimentary Linux system from a USB key, which then goes on to load a more substantial setup from the network. You'll find a full tutorial on our sister magazine PC Plus's website at **http://tinyurl.com/linpxeboot**.

» **Next month** Understand the Linux boot process and runlevels.