LPI certification 102 (release 2) exam prep, Part 2

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

Welcome to "Configuring and compiling the Linux kernel," the second of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll show you how to compile the Linux kernel from sources. Along the way, we'll cover various important kernel configuration options and provide more in-depth information about PCI and USB support in the kernel.

This tutorial is ideal for those who want to learn about or improve their Linux kernel compilation and configuration skills. This tutorial is especially appropriate for those who will be setting up Linux servers or desktops. For many readers, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of rounding out their important Linux kernel skills. If you are new to Linux, we recommend you start with *Part 1* and work through the series from there.

By the end of this series of tutorials (eight in all; this is part six), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

For those who have taken the *release 1 version* of this tutorial for reasons other than LPI exam preparation, you probably don't need to take this one. However, if you do plan to take the exams, you should strongly consider reading this revised tutorial.

The LPI logo is a trademark of the *Linux Professional Institute*.

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org*.

Daniel lives in Albuquerque, New Mexico, and is the Chief Architect of Gentoo Technologies, Inc., the creator of Gentoo Linux, an advanced Linux for the PC, and the Portage system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and their daughter, Hadassah.

Section 2. Introducing the kernel

And the kernel is... Linux!

Typically, the word "Linux" is used to refer to a complete Linux distribution and all the cooperating programs that make the distribution work. However, you may be surprised to find out that, technically, Linux is a kernel, and a kernel only. While the other parts of what we commonly call "Linux" (such as a shell and compiler) are essential parts of a complete operating environment, they are technically separate from Linux (the kernel). Still, people will continue to use the word "Linux" to mean "Linux-based distribution." Nevertheless, everyone can at least agree that the Linux kernel is the **heart** of every "Linux OS."

Interfacing with hardware

The primary role of the Linux kernel is to interface directly with the hardware in your system. The kernel provides a *layer of abstraction* between the raw hardware and application programs. This way, the programs themselves do not need to know the details of your specific motherboard chipset or disk controller -- they can instead operate at the higher level of reading and writing files to disk, for example.

CPU abstraction

The Linux kernel also provides a level of abstraction on top of the processor(s) in your system -- allowing for multiple programs to appear to run simultaneously. The kernel takes care of giving each process a fair and timely share of the processors' computing resources.

If you're running Linux right now, then the kernel that you are using now is either UP (uniprocessor) or SMP (symmetric multiprocessor) aware. If you happen to have an SMP motherboard, but you're using a UP kernel, Linux won't "see" your extra processors! To fix this, you'll want to compile a special SMP kernel for your hardware. Currently, SMP kernels will also work on uniprocessor systems, but at a slight performance hit.

Abstracting IO

The kernel also handles the much-needed task of abstracting all forms of file IO. Imagine what would happen if every program had to interface with your specific disk hardware directly -- if you changed disk controllers, all your programs would stop working! Fortunately, the Linux kernel follows the UNIX model of providing a simple data storage and access abstraction that all programs can use. That way, your favorite database doesn't need to be concerned whether it is storing data on an IDE disk, on a SCSI RAID array, or on a network-mounted file system.

Networking central

One of Linux's main claims to fame is its robust networking, especially TCP/IP support. And, if you guessed that the TCP/IP stack is in the Linux kernel, you're right! The kernel provides a standards-compliant, high-level interface for programs that want to send data over the

network. Behind the scenes, the Linux kernel interfaces directly with your particular Ethernet card or `pppd` daemon and handles the low-level Internet communication details. Note that the next tutorial in this series, Part 7, will deal with TCP/IP and networking.

Networking goodies

One of the greatest things about Linux is the wealth of useful optional features that are available in the kernel, especially related to networking. For example, you can configure a kernel that will allow your entire home network to access the Internet by way of your Linux modem -- this is called IP Masquerading, or IP NAT.

Additionally, the Linux kernel can be configured to export or mount network-based NFS file systems, allowing for other UNIX machines on your LAN to easily share data with your Linux system. There are a lot of goodies in the kernel, as you'll learn once you begin exploring the Linux kernel's many configuration options.

Booting review

Now would be a good time for a quick refresher of the Linux boot process. When you turn on your Linux-based system, the kernel image (stored in a single binary file) is loaded from disk to memory by a boot loader, such as LILO or GRUB. At this point, the kernel takes control of your system. One of the first things it does is detect and initialize all the hardware that it finds and has been configured to support. Once the hardware has been initialized properly, the kernel is ready to start normal user-space programs (also known as "processes").

The first process run by the kernel is /sbin/init. It, in turn, starts additional processes as specified in /etc/inittab. Within seconds, your Linux system is up and running, ready for you to use. Although you never interact with the kernel directly, the Linux kernel is always running "above" all normal processes, providing the necessary virtualization and abstractions that your various programs and libraries require to function.

Introducing... modules!

All recent Linux kernels support kernel modules. Kernel modules are really neat things -- they're pieces of the kernel that reside in relatively small binary files on disk. As soon as the kernel needs the functionality of a particular module, the kernel can load that specific module from disk and automatically integrate it into itself, thus dynamically extending its capabilities.

If the features of a loaded kernel module haven't been used for several minutes, the kernel can voluntarily disassociate it from the rest of the kernel and unload it from memory -- something that's called *autocleaning*. Without kernel modules, you'd need to ensure that your running kernel (which exists on disk as a single binary file) contains absolutely all the functionality you could possibly need. Without modules, you'd need to build a completely **new** kernel to add important new functionality to it.

Typically, users build a single kernel image that contains all **essential** functionality, and then build a bunch of modules that correspond to features that they **may** need in the future. If and when that time comes, the appropriate module can be loaded into the kernel as needed. This also helps to conserve RAM, since a module uses RAM only when it has been loaded into

the running kernel. When a module is removed from the kernel, that memory can be freed and used for other purposes.

Where modules live

Kernel modules typically live in /lib/modules/x.y.z (where x.y.z is the kernel version with which the modules are compatible); each module has ".o" at the end of its name, which identifies it as a binary file containing machine instructions. As you may guess, each individual module represents a particular component of kernel functionality. One module may provide FAT file system support, while another may support a particular ISA Ethernet card.

While the kernel modules for the 2.4 and earlier kernels end in ".o", kernel modules for the 2.5 and 2.6 kernels end in ".ko".

Modules -- not for every process!

It's worth mentioning that you can't put **everything** in a module. Because modules are stored on disk, your bootable kernel image needs to have compiled-in support for your disk controller, drives, and your root file system. If you don't have these essential components compiled into your kernel image -- that is, if you try to compile them as modules instead -- then your kernel won't have the necessary ability to load these modules from disk, creating a rather ugly chicken-and-egg problem that will result in a kernel that can't boot your system!

Section 3. Locating and downloading sources

Kernel version history

At the time this tutorial was written, the most recent kernel available was 2.4.20. The 2.4.20 kernel is part of the 2.4 stable kernel series. This series of kernel releases is intended for production systems.

There are also several 2.5 series kernels available, but you should not use them on production systems. The "5" in "2.5" is an odd number, indicating that these kernels are experimental in nature and intended for kernel developers. When the "2.5" kernels are ready for production use, a "2.6" (even second number) series will begin.

Getting new kernel sources

If you simply want to compile a new version of your currently-installed kernel (for example, to enable SMP support), then the best way to proceed is to install your distribution's kernel source package. After doing so, you should find a bunch of new files in /usr/src/linux.

However, there may be times where you want to install a *new* kernel. Generally, the best approach is to simply install a new or updated version of your distribution's kernel source package. This package will contain a kernel sources that have been patched and tweaked to run optimally on your Linux system. The latest versions of Red Hat Linux require a special "Red Hat" kernel in order to function. While other distributions don't specifically require that you use their patched kernel sources, doing so is still recommended.

Getting new kernel sources, continued

If you have a bit of an adventurous streak, you can grab a "mainline" kernel source tarball from *http://www.kernel.org/pub/linux/kernel* instead. In this directory, you'll find the official kernel sources, as released by Linus or Marcelo. They may not have all the features found in your distribution's kernel source package, so it's generally best to not use a mainline kernel until you feel that you know what you're doing :) Again, note that while "mainline" sources will work for most distributions, the latest versions of Red Hat need a specially-patched kernel source tree that you should get from Red Hat.

At kernel.org, you'll find the kernel sources organized into several different directories, based on kernel version (v2.2, v2.4, etc.) Inside each directory, you'll find files labeled "linux-x.y.z.tar.gz" and "linux-x.y.z.tar.bz2". These are the Linux kernel source tarballs. You'll also see files labeled "patch-x.y.z.gz" and "patch-x.y.z.bz2". These files are patches that can be used to update the previous version of complete kernel sources. If you want to compile a new kernel release, you'll need to download one of the "linux" files.

Unpacking the kernel

If you downloaded a new kernel from kernel.org, now it's time to unpack it. To do so, cd into /usr/src. If there is an existing "linux" directory there, move it to "linux.old" (`mv linux linux.old`, as root.)

Now, it's time to extract the new kernel. While still in /usr/src, type `tar xzvf /path/to/my/kernel-x.y.z.tar.gz` or `cat /path/to/my/kernel-x.y.z.tar.bz2 | bzip2 -d | tar xvf -`, depending on whether your sources are compressed with gzip or bzip2. After typing this, your new kernel sources will be extracted into a new "linux" directory. Beware -- the full kernel sources typically occupy more than 50 MB on disk!

Section 4. Configuring the kernel

Let's talk configuration

Before you compile your kernel, you need to configure it. Configuration is your opportunity to control exactly what kernel features are enabled (and disabled) in your new kernel. You'll also be in control of what parts get compiled into the kernel binary image (which gets loaded at boot-time), and what parts get compiled into load-on-demand kernel module files.

The old-fashioned way of configuring a kernel was a tremendous pain, and involved entering /usr/src/linux and typing `make config`. While make config still works, please don't try to use this method to configure your kernel -- unless you like answering hundreds (yes, hundreds!) of yes/no questions on the command-line.

The new way to configure

Instead of typing "make config," we modern folks type either "make menuconfig" or "make xconfig" to configure our kernels. If you type "make menuconfig," you'll get a nice console-based color menu system that you can use to configure the kernel. If you type "make xconfig," you'll get a very nice X-based GUI that can be used to configure various kernel options.

When using "make menuconfig," options that have a "< >" to their left can be compiled as a module. When the option is highlighted, hit the space bar to toggle whether the option is deselected ("< >"), selected to be compiled into the kernel image ("<*>"), or selected to be compiled as a module ("<M>"). You can also hit "y" to enable an option, "n" to disable it, or "m" to select it to be compiled as a module if possible. Fortunately, most kernel configuration options have verbose help that you can view by typing `h`.

Configuration tips

Unfortunately, there are so many kernel configuration options that we simply don't have room to cover them all here (but you can check the *options(4)* man page for a more complete list of options, if you're curious).

In the following panels, I'll give you an overview of the important categories you'll find when you do a "make menuconfig" or "make xconfig," pointing out essential or important kernel configuration options along the way.

Code maturity level options

Now, let's take a look at the various kernel configuration option categories. I'll include a brief overview of each category below. I encourage you to follow along by typing "make menuconfig" or "make xconfig" in /usr/src/linux.

Code maturity level options: This configuration category contains a single option: "Prompt for development and/or incomplete code/drivers." If enabled, many options that are considered experimental (such as ReiserFS, devfs, and others) will be visible under other

category menus. If this option isn't selected, the only options that will be visible will be those that are considered "stable." Generally, it's a good idea to enable this option so that you can see all that the kernel has to offer. Some of the "experimental" options are truly experimental, but many are experimental in name only and are in wide production use.

Modules and CPU-related options

Loadable module support: Under this configuration category are three options related to the kernel's support for modules. In general, all three options should be enabled.

Processor type and features: This section includes various CPU-specific configuration options. Of particular importance is the "Symmetric multiprocessing support option", which should be enabled if your system has more than one CPU. Otherwise, only the first CPU in your system will be utilized. The "MTRR Support" option should generally be enabled, since it will result in better performance in X on modern systems.

General and parallel port options

General setup: In this section, Networking and PCI support options should generally always be enabled, as should "Kernel support for ELF binaries" (build it into the kernel, not as a module). The a.out and MISC binary options are recommended, but generally make more sense as kernel modules. Also be sure to enable "System V IPC" and "Sysctl support." See the built-in help for more information on these options.

Parallel port support option: The **Parallel port support**section should be of interest to anyone with parallel port devices, including printers. Note that in order to have full printer support, you must also enable "Parallel printer support" under the "Character devices" section in addition to the appropriate parallel port support here.

RAID and LVM

Multi-device support (RAID and LVM): This contain options relating to Linux software RAID and logical volume management. Software RAID allows you to use your disks in a redundant fashion in order to increase availability. You can find more information on software RAID in the developerWorks software RAID series (see the final section of this tutorial, "Resources", for links).

Networking and related devices

Networking options: This contains options related to -- you guessed it-- networking! If you're planning to attach your Linux system to a typical network, you should be sure to enable "Packet socket," "Unix domain sockets" and "TCP/IP networking." There are various other options you may be interested in, including "Network packet filtering" that allows you to use the `iptables` command to set up your own stateful firewall. For information on doing this, see the *developerWorks* tutorial *Linux 2.4 stateful firewall design*.

Network device support: The second requirement for getting Linux networking to work is to

compile in support for your particular networking hardware. You should select support for the card(s) that you'd like your kernel to support. The options you want are most likely hiding under the "Ethernet (10 or 100Mbit)" sub-category.

IDE support

ATA/IDE/MFM/RLL support: This section contains important options for those using IDE drives, CD-ROMs, DVD-ROMs, and other peripherals. If your system has IDE disks, be sure to enable "Enhanced IDE/MFM/RLLdisk/cdrom/tape/floppy support," "Include IDE/ATA-2 DISK support", and the chipset appropriate to your particular motherboard (built-in to the kernel, not as modules -- so your system can boot!). If you have an IDE CD-ROM, be sure to enable "Include IDE/ATAPI CD-ROM support" as well. Note: without specific chipset support, IDE will still work but may not take advantage of all the performance-enhancing features of your particular motherboard.

Also note that the "Enable PCI DMA by default if available" is a highly recommended option for nearly all systems. Without DMA (direct memory access) enabled by default, your IDE peripherals will run in PIO mode and may perform up to **15 times** slower than normal! You can verify that DMA is enabled on a particular disk by typing `hdparm -d 1 /dev/hdx` at the shell prompt as root, where /dev/hdx is the block special device corresponding to the disk on which you'd like to enable DMA.

SCSI support

SCSI support: This contains all the options related to SCSI disks and peripherals. If you have a SCSI-based system, be sure to enable "SCSI support," "SCSI disk support," "SCSI CD-ROM support," and "SCSI tape support" as necessary. If you are booting from a SCSI disk, ensure that both "SCSI support" and "SCSI disk support" are compiled-in to your kernel and are not selected to be compiled as loadable modules. For SCSI to work correctly, you need to perform one additional step: head over to the "SCSI low-level drivers" sub-category and ensure that support for your particular SCSI card is enabled and configured to be compiled directly into the kernel.

Miscellaneous character devices

Character devices: This section contains a potpourri of miscellaneous kernel drivers. Be sure to enable "Virtual terminal" and "Support for console on virtual terminal"; these are needed for the standard text-based console that greets you after the kernel boots. You'll most likely need to enable "Unix98 PTY support" as well. If you want to use a parallel printer, remember to enable "Parallel printer support" too. Everything else is typically optional. "Enhanced real-time clock support" is recommended; "/dev/agpgart (AGP support)" and "Direct Rendering Manager" are typically required to take advantage of free Linux 3D acceleration under X (particularly if you have a Voodoo3+, ATI Rage 128, ATI Radeon, or Matrox card). Getting X to work in accelerated mode requires additional configuration steps besides simply enabling these options.

File systems and console drivers

File systems: This contain options related to file system drivers, as you might guess. You'll need to ensure that the file system used for "/" (the root directory) is compiled into your kernel. Typically, this is ext2, but it may also be ext3, JFS, XFS, or ReiserFS. Be sure to also enable the "/proc file system support" option, as most distributions require it. Typically, you should also enable "/dev/pts file system support for Unix98 PTYs," unless you're planning to use "/dev file system support," (also known as "devfs") in which case you should leave the "/dev/pts" option disabled, since devfs contains a superset of this capability.

Console drivers: Typically, most people will enable "VGA text console" (normally required on x86 systems) and optionally "Video mode selection support." It's also possible to use "Frame-buffer support," which will cause your text console to be rendered on a graphics rather than a text screen. Some of these drivers **can** negatively interact with X, so it's best to stick with the VGA text console, at least in the beginning.

Section 5. Compiling and installing the kernel

make dep

Once your kernel is configured, it's time to get it compiled. But before we can compile it, we need to generate dependency information. Do this by typing `make dep` while in /usr/src/linux.

make bzImage

Now it's time to compile the actual binary kernel image. Type `make bzImage`. After several minutes, compilation will complete and you'll find the bzImage file in /usr/src/linux/arch/i386/boot (for an x86 PC kernel). You'll see how to install the new kernel image in a bit, but now it's time for the modules.

Compiling modules

Now that the bzImage is done, it's time to compile the modules. Even if you didn't enable any modules when you configured the kernel, don't skip this step -- it's good to get into the habit of compiling modules immediately after a bzImage. And, if you really have **no** modules enabled for compilation -- this step will go really quickly for you. Type `make modules && make modules_install`. This will cause the modules to be compiled and then installed into/usr/lib/<kernelversion>.

Congratulations! Your kernel is now fully compiled, and your modules are all compiled and installed. Now it's time to reconfigure LILO so that you can boot the new kernel.

Section 6. Boot configuration

Intro to LILO

It's finally time to reconfigure LILO so that it loads the new kernel. LILO is the most popular Linux boot loader, and is used by all popular Linux distributions. The first thing you'll want to do is take a look at your /etc/lilo.conf file. It will contain a line that says something like "image=/vmlinuz" This line tells LILO where it should look for the kernel.

Configuring LILO

To configure LILO to boot the new kernel, you have two options. The first is to overwrite your existing kernel -- this is risky unless you have some kind of emergency boot method, such a boot disk with this particular kernel on it.

The safer option is to configure LILO so that it can boot either the new or the old kernel. LILO can be configured to boot the new kernel by default, but still provide a way for you to select your older kernel if you happen to run into problems. This is the recommended option, and the one we'll show you how to perform.

LILO code

Your lilo.conf may look like this:

```
boot=/dev/hdadelay=20vga=normalroot=/dev/hda1read-only
image=/vmlinuzlabel=linux
```

To add a new boot entry to your lilo.conf, do the following. First, copy /usr/src/linux/arch/i386/boot/bzImage to a file on your root partition, such as /vmlinuz2. Once it's there, duplicate the last three lines of your lilo.conf and add them again to the end of the file... we're almost there...

Tweaking LILO

Now, your lilo.conf should look like this:

```
boot=/dev/hda delay=20 vga=normal root=/dev/hda1 read-only
image=/vmlinuz label=linux
image=/vmlinuzlabel=linux
```

Now, change the first "image=" line to read `image=/vmlinuz2`" Next, change the **second** "label=" line to read `label=oldlinux`. Also, make sure there is a "delay=20" line near the top of the file -- if not, add one. If there is, make sure the number is at least twenty.

The final lilo.conf

Your **final** lilo.conf file will look something like this:

```
boot=/dev/hda  delay=20  vga=normal  root=/dev/hda1  read-only
image=/vmlinuz2  label=linux
image=/vmlinuz label=oldlinux
```

After doing all this, you'll need to run "lilo" as root. This is very important! If you don't do this, the booting process won't work. Running "lilo" will give it an opportunity to update its boot map.

The whys and wherefores of LILO configuration

Now for an explanation of our changes. This lilo.conf file was set up to allow you to boot two different kernels. It'll allow you to boot your original kernel, located at /vmlinuz. It'll also allow you to boot your new kernel, located at /vmlinuz2. By default, it will try to boot your new kernel (because the image/label lines for the new kernel appear first in the configuration file).

If, for some reason, you need to boot the old kernel, simply reboot your computer and hold down the shift key. LILO will detect this, and allow you to type in the label of the image you'd like to boot. To boot your old kernel, you'd type `oldlinux`, and hit Enter. To see a list of possible labels, you'd hit TAB.

Section 7. PCI devices

PCI devices 101

This section will take a closer look at the finer details of dealing with PCI devices under Linux. Enabling support for PCI devices under Linux is pretty straightforward. Simply ensure that you have "PCI support" enabled under the "General Setup" kernel configuration category. The "PCI device name database" option is also recommended, as it will allow you to view the actual English names of the PCI devices that Linux can see (instead of just their official PCI device ID numbers). Other than ensuring that the above options are enabled, Linux is ready to support PCI.

The only additional step required is to enable the specific driver for the type of card you're installing into your system. For example, you'd enable "Creative SBLive!" support (under the "Sound" category) if you were installing a SoundBlaster Live!" card, and you'd enable "3c590/3c900 series (592/595/597) "Vortex/Boomerang" support" under the "Network device support/Ethernet (10 or 100Mbit)" category/subcategory if you were installing a 3Com 3c905c Fast Ethernet card.

Inspecting your PCI devices

To view information about installed PCI devices, you can type `cat /proc/pci` to view bare-bones (and somewhat cryptic) information -- or type `lspci -v` for more verbose and understandable output. The "lspci" is part of the pciutils package, whose sources are available from *http://atrey.karlin.mff.cuni.cz/~mj/pciutils.html*. Generally, using the version of pciutils that comes with your particular distribution is sufficient. When you type `lspci -v`, you may see many PCI devices that you never knew even existed. More often than not, such a device is one of the many PCI-based peripheral devices that has been built-in to your computer's motherboard. These devices can be disabled (and enabled if they aren't currently visible) in your computer's BIOS, typically under the "Integrated peripherals" section. You can normally access your computer's BIOS by pressing the Delete key or the F2 key as your system boots.

The pciutils package also contains a program called "setpci" that can be used to change various PCI device settings, including PCI device latency. To learn more about PCI device latency and the effects it can have on your system, see the *developerWorks* article *Linux hardware stability guide, Part 2*.

PCI device resources

In order to do their thing, the PCI devices in your system need to take advantage of various specific hardware resources in your system, such as interrupts. Many PCI devices take advantage of hardware interrupts to signal the processor when they have some data ready for processing. To see what interrupts are being used by your various hardware devices, you can view the /proc/interrupts file by typing `cat /proc/interrupts`. You'll see output that looks something like this:

```
           CPU0       0:    3493317          XT-PIC  timer  1:      86405          XT
```

The first column lists an IRQ number; the second column displays how many interrupts have been processed by the kernel for this particular IRQ; and the last column identifies the "short name" of the hardware device(s) associated with the IRQ. As you can see, multiple devices are capable of sharing the same IRQ if necessary.

/proc also contains additional information about your hardware contained in the following files:

Section 8. Linux USB

Introducing Linux USB

When configuring the kernel, you probably noticed a "USB support" section containing options pertaining to USB, also known as the Universal Serial Bus. USB is a relatively new way of connecting peripheral devices to PCs. These days, there are USB mice, keyboards, game controllers, printers, modems, and more. Because Linux USB support is relatively new, many Linux users have never used USB devices on their Linux systems, or may not be fully up-to-speed on how Linux USB support works. The following panels will give you a quick introduction to Linux USB to help you get started.

This 2nd edition of the LPI 102 tutorials also includes a more thorough look at USB in Part 4, which will guide you through the steps to enable Linux support for several popular USB devices.

Enabling USB

To enable Linux USB support, first go inside the "USB support" section and enable the "Support for USB" option. While that step is fairly obvious, the following Linux USB setup steps can be confusing. In particular, you now need to select the proper USB host controller driver for your system. Your options are "EHCI," "UHCI," "UHCI (alternate driver)," and "OHCI." This is where a lot of people start getting confused about USB for Linux.

UHCI, OHCI, EHCI -- oh my!

To understand what "EHCI" and friends are, you need to first know that every motherboard or PCI card that includes support for plugging in USB devices needs to have a USB host controller chipset on it. This particular chipset interfaces with the USB devices that you plug in to your system and takes care of all the low-level details necessary to allow your USB devices to communicate with the rest of the system.

The Linux USB drivers have three different USB host controller options because there are three different types of USB chips found on motherboards and PCI cards. The "EHCI" driver is designed to provide support for chips that implement the new high-speed USB 2.0 protocol. The "OHCI" driver is used to provide support for USB chips found on non-PC systems, as well as those on PC motherboards with SiS and ALi chipsets. The "UHCI" driver is used to provide support for the USB implementations you'll find on most other PC motherboards, including those from Intel and Via. You simply need to select the "?HCI" driver that corresponds to the type of USB support you'd like to enable. If in doubt, you can enable "ECHI," "UHCI" (pick either of the two, there's no significant difference between them), and "OHCI" just to be safe.

The last few steps

Once you've enabled "USB support" and the proper "?HCI" USB host controller drivers, there are just a few more steps involved in getting USB up and running. You should enable the "Preliminary USB device file system," and then ensure that you enable any drivers specific to

the actual USB peripherals that you will be using with Linux. For example, in order to enable support for my USB game controller, I enabled the "USB Human Interface Device (full HID) support". I also enabled "Input core support" and "Joystick support" under the main "Input core support" section.

Mounting usbdevfs

Once you've rebooted with your new USB-enabled kernel, you should mount the USB device file system to /proc/bus/usb by typing the following command:

```
# mount -t usbdevfs none /proc/bus/usb
```

In order to have the USB device file system mounted automatically when your system boots, add the following line to /etc/fstab after the /proc mount line:

```
none            /proc/bus/usb            usbdevfs        defaults   0   0
```

These steps are unnecessary for many Linux distributions, which will auto-detect if usbdevfs support is enabled in your kernel at boot time and automatically mount the usbdevfs file system if possible. For more information about USB, visit the USB sites I've listed in "Resources," which follow.

Section 9. Summary and resources

Summary

Congratulations, you've reached the end of this tutorial! We hope it has helped solidify your knowledge of compiling the kernel. Please join us in the next tutorial in the series (Part 7) on networking, where we show you TCP/IP and Ethernet networking fundamentals, show you how to use `inetd` and `xinetd`, and introduce you to security and print serving. By continuing in this tutorial series, you'll soon be ready to attain your LPIC Level 1 Certification from the Linux Professional Institute.

On the next page, you'll find a number of resources that will help you learn more about the topics in this tutorial.

Resources

The *Linux Kernel HOWTO* is another good resource for kernel compilation instructions.

The *LILO, Linux Crash Rescue HOW-TO* shows you how to create an emergency Linux boot disk.

www.kernel.org hosts the Linux Kernel archives.

Don't forget *The Linux Documentation Project*. You'll find its collection of guides, HOWTOs, FAQs, and man-pages to be invaluable. Be sure to check out *Linux Gazette* and *LinuxFocus* as well.

The Linux System Administrators guide, available from *Linuxdoc.org's "Guides" section*, is a good complement to this series of tutorials -- give it a read! You may also find Eric S. Raymond's *Unix and Internet Fundamentals HOWTO* to be helpful.

In the *Bash by example* article series on *developerWorks*, Daniel shows you how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly Parts 1 and 2) will be excellent additional preparation for the LPI exam:

- *Bash by example, Part 1: Fundamental programming in the Bourne-again shell*
- *Bash by example, Part 2: More bash programming fundamentals*
- *Bash by example, Part 3: Exploring the ebuild system*

We highly recommend the *Technical FAQ for Linux users* by Mark Chapman, a 50-page in-depth list of frequently asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out. We also recommend *Linux glossary for Linux users*, also from Mark.

If you're not familiar with the `vi` editor, we strongly recommend that you check out Daniel's *Vi intro -- the cheat sheet method tutorial* . This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use `vi`.

You can find more information on software RAID in Daniel's *developerWorks* software RAID series: *Part 1* and *Part 2*. Logical volume management adds an additional storage management layer to the kernel that allows you to easily grow, shrink, and span file systems across multiple disks. To learn more about LVM, see Daniel's articles on the subject: *Part 1* and *Part 2*. Both software RAID and LVM require additional user-land tools and setup.

For more information on using the `iptables` command to set up your own stateful firewall, see the *developerWorks* tutorial *Linux 2.4 stateful firewall design*.

For more information about USB, visit *linux-usb.org* . For additional USB setup and configuration instructions, be sure to read the *Linux-USB guide* .

For more information on the Linux Professional Institute, visit the *LPI home page*.

Feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the author, Daniel Robbins, directly at *drobbins@gentoo.org*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.