Creating custom kernels with Debian's kernel-package system

Kevin McKinley

<kjmck@users.sourceforge.net>

Revision History

Revision v0.95	15 April 2003	Revised by: kjm

Made new section for "Checking Minimal Requirements", added use of ver_linux script to check minimal requirements, changed to use "dpkg -i" to check program versions instead of checking Debian.changelog, moved "Create symlink" section, added new section "Adding third-party modules to an existing kernel", numerous minor stylistic changes.

Revision v0.9	21 March 2003	Revised by: kjm

Numerous additions and revisions to bring up to date with Woody and kernel 2.4.18, revised list of packages to install, added general comments on configuration, expanded comments on xconfig, added section on menuconfig, added section on --append-to-version, revised section on --revision, deleted section on --flavours, added section on kernel-image file names, added use of fakeroot, added Advanced topics page.

Revision v0.3	23 May 2001	Revised by: jwg

Updated section on symlinks, program version checks, how to compile as a regular user rather than root, filled in the checklist, added troubleshooting section(needs work).

Revision v0.2	22 April 2001	Revised by: jwg

Quick fix on grammatical and spelling errors.

Revision v0.1	21 April 2001	Revised by: jwg

Initial release.

Thanks to Jesse Goerz for taking the biggest step and writing the first three versions of this document, and for graciously allowing me to carry it forward.

Table of Contents

1. Introduction

I used to compile kernels manually, and it involved a series of steps to be taken in order; kernel-package was written to take all the required steps (it has grown beyond that now, but essentially, that is what it does). This is especially important to novices: make-kpkg takes all the steps required to compile a kernel, and installation of kernels is a snap. — Manoj Srivastava, author of kernel-package

If you have ever compiled a kernel you will be amazed at how easy it can be. This tutorial was designed to enable you to use Debian's kernel-package system, which simplifies building and installing custom kernels. Learn to create Debian packages of kernels and their modules, and manage the resulting kernel images with dpkg.

> *You can make your system unbootable if you're not careful. I highly recommend you read this entire document at least once, then work through it as you build your kernel.*

Please remember that inevitably, some people will use this document and screw up their systems. Don't be one of them. If you don't understand something, get more information before proceeding. If you see something here that you know is a mistake, please let me know at <newbiedoc-discuss@lists.sourceforge.net>.

---

2. Preparations

2.1. Making a backup boot diskette

Before you make a new kernel the first thing to do is make a boot disk for the kernel you're running. This way, if anything gets screwed up you'll be able to boot the machine and fix it. Grab a floppy disk you don't mind erasing and do this (as the root user):

```
bash:~# mkboot path_to_kernel
```

Replace `path_to_kernel` with the appropriate information, such as "/boot/vmlinuz-2.4.18-bf2.4" (if you installed Woody with the 2.4 kernel).

2.2. What to install

We will be working with Debian 3.0r1 (Woody revision 1); you will need the following packages:

1. gcc
2. kernel-package
3. kernel-source-2.4.18 (or whatever kernel sources you will be using)
4. libc6-dev
5. tk8.0 or tk8.1 or tk8.3
6. libncurses5-dev
7. fakeroot
8. bin86 (for building 2.2.x kernels on PCs)

You can install these packages by doing (as the root user):

```
bash:~# apt-get install gcc kernel-package kernel-source-2.4.18 libc6-dev tk8.3 libncurses5-dev
```

Installing these packages will cause several other packages to be installed to satisfy dependencies.

> *If you're compiling a 2.2 series kernel for the PC architecture (machines with AMD processors are PCs, but not Macs or Alphas) you'll need the bin86 package.*

You can use any version of the kernel sources you wish; I chose 2.4.18 because it's the most current in Woody. Debian maintainers have done an excellent job packaging kernel sources into .deb files, and I recommend you use those rather than other source packages. The only reason I can see to use other kernel source packages is to get hardware support not available in Debian.

---

3. Setting up the source tree

"Source tree" is just a term for the directory which contains the source code you will be compiling.

3.1. Adding yourself to the src group

When a kernel source package is installed, the source tarball is placed in the `/usr/src` directory. To work in this directory, you must be the either the root user or a member of the src group. To avoid doing things as root whenever possible, add yourself to src. Only the root user can add users to groups, so become root and do the following:

```
bash:~# adduser my_username src
Adding user my_username to group src...
Done.
```

Of course, substitute your own user name for "my_username". Your group memberships are checked during login, so you need to logout and login again to be recognized as a member of src. Here are three ways to do so:

1. If you're at a terminal window inside X, log out of X and logout from the console. Then log in at the console and start another X session.
2. If you're using a graphical login manager like xdm, gdm, or kdm, log out of X and back in again.
3. If you're at a console just log out and log back in.

To make sure you were actually added to the src group, issue this command (once you've done the logout/login dance):

```
bash:~$ groups
my_username src
```

You should see "src" among the groups listed. (By default, you also belong to the one-user group "my_username".)

3.2. Expanding the source tarball

As a normal user, change to the directory `/usr/src` and list its contents; you should see the bzipped tar file you installed:

```
bash:~$ cd /usr/src
bash:/usr/src$ ls
kernel-source-2.4.18.tar.bz2
```

If you also see a symbolic link called `linux` remove it:

```
bash:/usr/src$ rm linux
```

> *If you have that symlink and don't remove it before expanding your new source tarball, you may get a mess of old and new source files which won't compile correctly.*

Debian kernel source packages expand to a directory named `kernel-source-x.y.z/`. "Official" kernel sources for kernels up to 2.4.18 expand to a directory called `linux/`. (We recommend moving that directory to `linux-x.y.z/`.) "Official" sources for kernels 2.4.19 and later expand to `linux-x.y.z/`.

To expand the source tarball do this as a regular user:

```
bash:/usr/src$ tar -jxf kernel-source-2.4.18.tar.bz2
```

> *Past versions of tar used the `-I` option for bzip2 compression.*

> *If you have gzipped kernel sources (the filename ends in `gz` instead of `bz2`) use `z` instead of `j` as an option to `tar`.*

This should create a directory called `kernel-source-2.4.18/` owned by you and the `src` group.

3.3. Setting up the symlink

There should not be a symlink pointing to any kernel source tree; it's time to create that link now.

```
bash:/usr/src$ ln -s kernel-source-2.4.18 linux
```

Substitute the name of the directory created when you expanded your kernel source for "kernel-source-2.4.18".

It's a good idea to double-check symbolic links, so do "ls -l". It should look like the example below.

Example 1. Symlink

```
bash:/usr/src$ ls -l
total 66908
-rw-r--r--    1 root     root     23833519 Mar 20 16:18 kernel-source-2.4.18.tar.bz2
drwxr-sr-x   15 kjmck  ❶ src          4096 Mar 20 16:30 kernel-source-2.4.18
lrwxrwxrwx    1 kjmck    src            20 Mar 20 16:36 linux -> kernel-source-2.4.18
                                                                        ❷
```

❶ The group for both the kernel source tree and the link is src.

❷ The arrow shows that the link "linux" points to the kernel source tree.

3.4. Checking Current Minimal Requirements

Inside the `linux` directory you should find another called `Documentation`. Change into that directory and look for the file `Changes`. Open it with your favorite text editor and look for the section "Current Minimal Requirements". Here's an excerpt from mine:

Example 2. Partial contents of /usr/src/kernel-source-2.4.18/Documentation /Changes

```
Current Minimal Requirements
============================

Upgrade to at *least* these software revisions before thinking you've
encountered a bug!  If you're unsure what version you're currently
running, the suggested command should tell you.

o  Gnu C                  2.95.3                  # gcc --version
o  Gnu make               3.77                    # make --version
o  binutils               2.9.1.0.25              # ld -v
o  util-linux             2.10o                   # fdformat --version
o  modutils               2.4.2                   # insmod -V
```

```
o  e2fsprogs                1.25                    # tune2fs
o  reiserfsprogs            3.x.0j                  # reiserfsck 2&>1|grep reiserfsprogs
o  pcmcia-cs                3.1.21                  # cardmgr -V
o  PPP                      2.4.0                   # pppd --version
o  isdn4k-utils             3.1pre1                 # isdnctrl 2>&1|grep version
```

The `Changes` document lists a required program in the first column, the required version in the 2nd, and a command to check the version in the 3rd. You may not need all the programs/packages listed. (For example you don't need pcmcia-cs if you don't use PC cards.)

Instead of checking for each required program individually, you can run the script `ver_linux`, which is found in the `scripts` directory of the kernel tree. You'll need to make it executable first. Note that the script may not check all the requirements listed in `Changes`, and that it checks for some programs *not* listed.

Example 3. Using ver_linux to Check Minimal Requirements

```
bash:/usr/src/linux/Documentation$ cd ../scripts
bash:/usr/src/linux/scripts$ chmod +x ver_linux
bash:/usr/src/linux/scripts$ ./ver_linux
If some fields are empty or look unusual you may have an old version.
Compare to the current minimal requirements in Documentation/Changes.

Linux sirius 2.4.18.030309 #1 Sun Mar 9 22:15:39 EST 2003 i586 unknown unknown GNU/Linux

  Gnu C                  2.95.4
  Gnu make               3.79.1
  util-linux             2.11n
  mount                  2.11n
  modutils               2.4.15
  e2fsprogs              1.27
  pcmcia-cs              3.1.33
  Linux C Library        2.3.1
  Dynamic linker (ldd)   2.3.1
  Procps                 2.0.7
  Net-tools              1.60
  Console-tools          0.2.3
  Sh-utils               2.0.11
  Modules Loaded         ds i82365 3c589_cs pcmcia_core ext3 jbd rtc
```

If you can't determine program versions on your Debian system using the above methods, try using dpkg.

Example 4. Using dpkg to Check Minimal Requirements

```
bash:/usr/src/linux/scripts$ dpkg -l make
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-installed
|/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: uppercase=bad)
||/ Name           Version        Description
+++-==============-==============-============================================
ii  make           3.79.1-14      The GNU version of the "make" utility.
```

If you're using sources from kernel.org or from the Debian testing or unstable branches, you may need to update some of your tools to meet the minimal requirements. If you're using sources from the Debian stable branch the above should have been a mere formality. Now that you have the source tree set up you can configure the kernel.

---

4. Configuring the kernel

4.1. General Notes

An important principle of kernel configuration is TANSTAAFL (There Ain't No Such Thing As A

Free Lunch). Any features you add to the kernel increase its size (and the time to build it), even if you choose to add them as modules.

To make their kernels useable by most people on most hardware, Linux distributors generally include support for most hardware and functions. Debian is no different; pre-compiled kernels include support for hardware you'll never have and languages you'll never read. My general rule for kernel configuration is, "If in doubt leave it out." If you test your new kernel and find that some of your hardware doesn't work, it's easy to tweak the configuration and build another kernel.

> *Kernel size depends on configuration. My kernel 2.4.18 was 645 KB; the kernel I built using Debian's 2.4.18-bf2.4 configuration was 1228 KB.*

Apart from larger and larger sizes, another feature of recent kernel images is their use of initrd. A special dictionary I keep next to my computer defines initrd as "one more thing to worry about". You can avoid the need to worry about initrd by ensuring that you compile directly into the kernel (not as modules) support for your boot hardware and root filesystem. If the hard drive you boot from is IDE, compile IDE support into the kernel. If your root filesystem is Reiserfs be sure Reiserfs support is built not as modules but directly into the kernel.

> *Be sure (!) that you build floppy disk support into the kernel. If you compile it as modules you won't be able to boot with your new kernel from a floppy disk.*

Unfortunately a detailed explanation of kernel options is well beyond the scope of this document. To find out more about configuring the kernel, please see The Linux Documentation Project's "Kernel-HOWTO", the files in `/usr/src/linux/Documentation`, and your hardware documentation. Search the archives of the debian-user mailing list, and ask questions there if you don't find answers in the archives. Remember that "Google is your friend."

4.2. make xconfig

There are several ways to configure the kernel. The first one we will cover is called "xconfig". Once you've changed to the `/usr/src/linux` directory, start it like this:

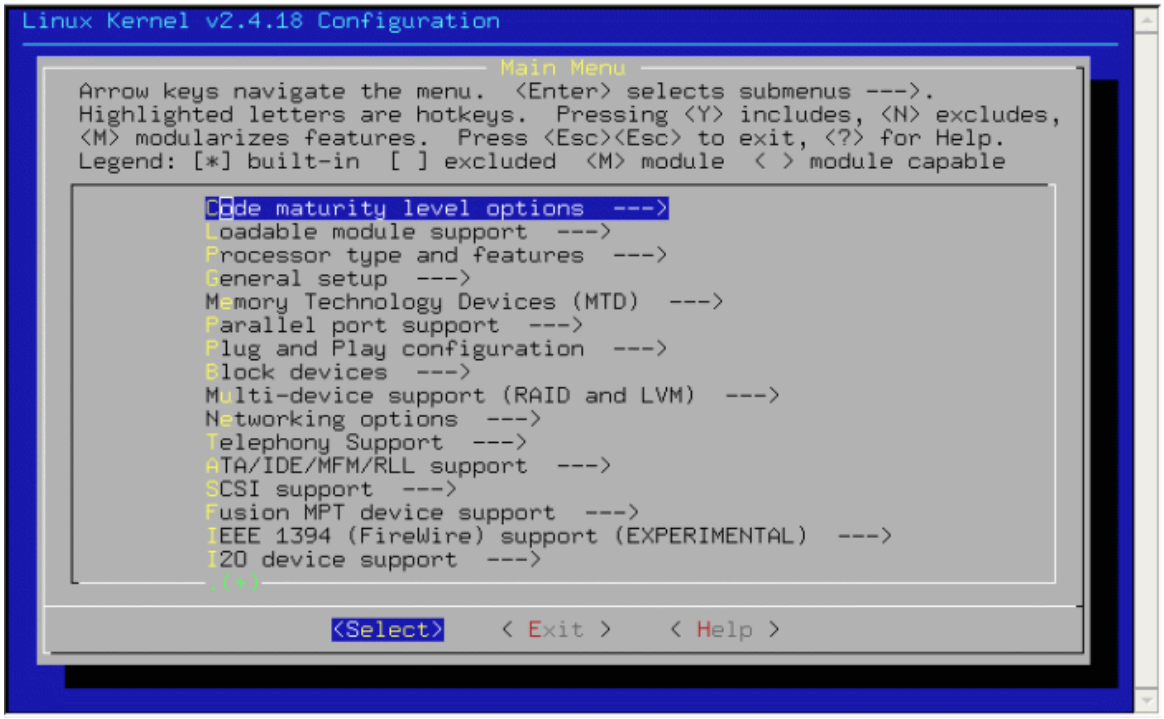
```
bash:/usr/src/linux$ make xconfig
```

Notice that the "IEEE 1394 ..." and "Bluetooth support" buttons are greyed out. They are not available because they depend on another option which has not been enabled. Click on "Code maturity level options" and change "Prompt for development and/or incomplete code/drivers" to "y". Now click the "Main menu" button, and notice how the IEEE and Bluetooth buttons can be selected just like the others. Anywhere you see an option that's greyed out, it depends on another option that has been disabled somewhere else.

4.3. make menuconfig

Perhaps you aren't using X. Perhaps you're getting better at this configuration business, and your trusty keyboard is faster than a mouse. (Maybe your cat is a little *too* efficient.) Menuconfig may be just the tool for you.

```
bash:/usr/src/linux$ make menuconfig
```



If you start menuconfig in a longer terminal window, you'll be able to see more categories without scrolling. Menuconfig opens to the size of the terminal you start it from.

The options are the same as you'd see in xconfig, with one crucial difference -- there's no way to "grey out" an option in menuconfig. If you don't select "Prompt for development or incomplete code/drivers", you won't even see the options for Firewire or Bluetooth.

For this reason many of us encourage kernel newbies to use xconfig, and try menuconfig later when they're more familiar with kernel configuration.

4.4. make config

Kernel 0.01 had no configuration at all, because there *were* no options. Kernel 1.0 had 49 different options, so a configuration mechanism was needed. Enter "make config".

Now exit "make config". You don't want to answer as many as ~1300 different questions one at a time, and there's very little chance you'd get a configuration you could live with anyway. Much better is to use either xconfig or menuconfig.

---

5. Building the kernel image

If you have an older machine with a slower processor it can take several hours to build a Debian kernel image. (It is possible to build your kernel on a faster machine and install it on the slower machine, but I won't cover that here.) I built a kernel 2.4.18 package on my high-speed low-drag 166 MHz Pentium and it took about 1 hour 40 minutes. That was using my configuration. Using Debian's kernel 2.4.18-bf2.4 configuration, it took 4 hours 45 minutes on the same machine. On a 1GHz Pentium III the same packages built in 9 and 23 ½ minutes, respectively.

5.1. make-kpkg

To build the kernel you'll invoke "make-kpkg", a script which automates and replaces the sequence "make dep; make clean; make bzImage; make modules". Take a few minutes and read over the manual page for make-kpkg.

> *make-kpkg is part of the "kernel-package" package. Its documentation is located in* `/usr/share/doc/kernel-package/`.

The make-kpkg command line can be complex and at first intimidating. Its basic syntax is

make-kpkg <options> <target>

Your target will be "kernel_image". Let's examine two of the more important and common options, "--append-to-version" and "--revision".

5.2. --append-to-version

The first option lets you specify an addition to the kernel version, which then becomes part of the kernel's name. You may use alphanumeric characters, "+" and "." (period or full stop); do *not* use underscore "_".

Here's the kernel I'm running now:

```
bash:/usr/src/linux$ uname -a
bash:/usr/src/linux$ Linux sirius 2.4.18.030309 #1 Sun Mar 9 22:15:39 EST 2003 i586 unknown unk
```

I used "--append-to-version=.030309" (note the leading ".") when I built the image for this kernel. It's shorthand for "2003 March 09", and it's my way of distinguishing this kernel from other 2.4.18 kernels (such as kernel 2.4.18.020724 which I built on 2002 July 24).

> *You should avoid using --append-to-version values such as "-686", "-K7", and "-sparc". They are commonly used for Debian pre-compiled kernels.*

Kernel modules live in subdirectories of `/lib/modules`; each kernel has its own subdirectory. Every time you install a kernel image with a new name, the package installer creates a new subdirectory of `/lib/modules` to hold its modules.

This means that by using a new value for --append-to-version each time you build a kernel image, you can ensure that the new kernel will have a new name, and that its modules won't conflict with those of other kernels.

> *If you install a kernel with the same name (the same version and --append-to-version) as an already-installed kernel, installing the new kernel package will overwrite the old kernel and its modules. You will be warned and offered the chance to abort. Take it. Use another value for --append-to-version and rebuild.*

5.3. --revision

Another make-kpkg option is "--revision", which affects the name of the Debian package itself but not the kernel name. As with --append-to-version, you may use only alphanumeric characters, "+" and ".". Do *not* use underscores "_". If you do not supply a value for --revision, make-kpkg will use "10.00.Custom".

Using different values of --revision will *not* prevent conflicts between kernels with the same name.

You will not be using --revision in this tutorial.

5.4. Kernel package names

Debian kernel-image *file* names have the form

kernel-image-(kernel-version)(--append-to-version)_(--revision)_(architecture).deb

The *package* name is everything before the first underscore. Here is a partial list of my kernel packages:

```
bash:/usr/src$ ls
kernel-image-2.4.18.030309_1.0_i386.deb
kernel-image-2.4.18.030320_1.0_i386.deb
kernel-image-2.4.18.030320_10.00.Custom_i386.deb
```

Now you can see why underscores are not allowed in make-kpkg options — they separate the

elements of package names.

I have two kernel *names*: 2.4.18.030309 and 2.4.18.030320. I can install both without worrying about conflicts.

I have two *revisions* of kernel 2.4.18.030320: revision 1.0 and revision 10.00.Custom. If I install either of these kernels and later try to install the other I will get a warning about conflicting kernels/modules. If I continue I will overwrite the already-installed kernel and its modules.

I recommend using a different --append-to-version value for each kernel you compile, and letting make-kpkg assign the default revision. Date-based values work for me, but you are free to invent your own scheme.

> *Please read `/usr/share/doc/kernel-package/README.gz` if --append-to-version or --revision is unclear, or if you plan on using options different from the ones I suggest. (One way to do this is "zcat README.gz | most".) Ignore the discussions of flavours and epochs until you are more familiar with make-kpkg and with Debian packages generally; they are not likely to be useful to you now.*

5.5. fakeroot

When you added yourself to the src group you made it possible to do most kernel-package work with normal user privilege. However the files that are part of a kernel package (like the kernel and kernel modules) will be owned by root and run with root privilege; they must be created with root privilege.

Using fakeroot you can start make-kpkg as a normal user, and most of the job will be run with normal permissions. Near the end of the job, fakeroot will simulate a root environment to create the kernel-image package.

The manual page for make-kpkg describes one way to use fakeroot; we will use the simpler method of putting "fakeroot" at the beginning of the make-kpkg command line, like this:

fakeroot make-kpkg <options> <target>

5.6. Making the kernel image

Finally! The time has arrived — you're ready to make the kernel image.

```
bash:/usr/src/linux$ fakeroot make-kpkg clean
```

Then do:

```
bash:/usr/src/linux$ fakeroot make-kpkg --append-to-version=.030320 kernel_image
```

Now all you can do is wait for the computer to finish. Take a few minutes off and enjoy yourself while the computer does its fair share of the work.

---

6. Installing the kernel-image package

Once the kernel image is built you will install the kernel-image package, which includes the kernel and its modules. First check to make sure it was built successfully by changing to the `/usr/src` directory and listing its contents:

```
bash:/usr/src/linux$ cd ..
bash:/usr/src$ ls
kernel-source-2.4.18
kernel-source-2.4.18.tar.bz2
❶kernel-image-2.4.18.030320_10.00.Custom_i386.deb
linux
```

❶ There it is! Notice how the --append-to-version value became part of the kernel name.

To install the kernel and all its modules become root and do:

```
bash:/usr/src# dpkg -i kernel-image-2.4.18.030320_10.00.Custom_i386.deb
```

To install a kernel-image package from disk (as opposed to installing from a Debian archive) use the full file name.

The post-install script should ask if you want to make a boot disk. If you made one earlier you can say "no" here. Otherwise grab a floppy you don't mind erasing and say "yes".

> *If you haven't made a boot disk yet then you need to do so. If the post-install process whacks lilo you will not be able to restart without a boot floppy.*

Be sure you say "no" when the script asks if you want to install a boot block using the existing `/etc/lilo.conf`. That would be trying to boot the new kernel with the old configuration, and it won't work. Next you'll be asked if you want to wipe out the old LILO configuration and make a new one. If you use LILO that would be "yes". Then you'll be asked whether you want to set up Linux to boot from the hard disk.

> *If you are using a bootloader other than lilo you should check the documentation in `/usr/share/doc/kernel-package` before answering this question. You probably will not want the post-install script to handle your bootloader.*

If you say "yes", you'll be prompted to install a partition boot record, install a master boot record, and make your root partition the active partition.

If you use grub see Kernel-package and grub.

If everything went well, all you need to do now is reboot. If your new kernel doesn't boot, insert your boot floppy and restart. Go back to Configuring the kernel, tweak your configuration, and try again.

Congratulations on compiling your first kernel with kernel-package. No sweat eh? There are still a few things you should do before you declare victory and fire up Quake III.

---

7. What now?

7.1. Hold that kernel!

If you used --append-to-version you shouldn't need to worry about apt-get trying to "upgrade" your kernel. If you're paranoid (it *is* out to get you) you can make sure by doing this (as root):

```
bash:~# echo "kernel-image-2.4.18.030320 hold" | dpkg --set-selections
```

Of course substitute the real name of your kernel. To refer to a package that dpkg knows about (an installed package or one in an archive's list) use the package name rather than the full file name. Also, the "|" character is made by typing **Shift**-****. Check that the package really is on hold; if it is you should see:

```
bash:~# dpkg --get-selections | grep kernel-image
kernel-image-2.4.18.030320      hold
```

7.2. Removing the symlink

Next, you should get rid of the symlink you created in the `/usr/src` directory. There are several reasons for this:

1. The next time you download a kernel it might not be from a Debian archive. When you expand the source tarball it could overwrite your old source tree right through the old symlink. Bummer.
2. The next time you download the kernel source from the Debian archive, you might expand the source into its own tree without problems. But since you already have a symlink called "linux" in the src directory, you might go ahead and compile (forgetting of course that it's linked to your old source tree.)
3. When you download patches or other source code into a specific source tree, you don't want anything else messing with it. By removing the symlink you might prevent #1 from happening.

To remove the link do this:

```
bash:~# cd /usr/src
bash:/usr/src# rm linux
```

7.3. Backing up your kernel

While not required, it's a good idea is to back up your custom kernel .deb. Copy it to a secure undisclosed location.

> *Once your kernel has been packaged with its modules, you can install it on any machine that has the hardware you specified in your kernel configuration. You could even reinstall it on your current machine after reinstalling your system.*

7.4. Making a new boot diskette

Create another boot diskette, this one for the kernel you just installed. Grab another floppy — it's not a good idea to overwrite your old boot disk; you haven't been using your new kernel long enough to be sure it works.

```
bash:/usr/src# cd /boot
bash:/boot# mkboot /boot/vmlinuz-2.4.18.030320
```

7.5. Removing old kernels

Now that you see how easy it is to build and install kernels, it won't be long before your `/boot` directory is crowded with kernels and configs and System.maps. Use dpkg to purge the ones you don't need:

```
bash:/boot# dpkg -P kernel-image-2.4.18.030309
```

If you get a message that "/lib/modules/2.4.18.030309 is not empty so wasn't removed", you probably have third-party modules installed there. They too can be removed by purging their packages. (See Third-party modules to find out how to make them with make-kpkg.)

```
bash:/boot# dpkg -P pcmcia-modules-2.4.18.030309
```

If the modules subdirectory still isn't empty, you probably have kernel modules that you didn't build with make-kpkg. You can remove the modules subdirectory once you're satisfied there's nothing there you need.

7.6. Building your next kernel

If you want to rebuild your kernel because you just bought a new sound card, or you want to enable a new feature you forgot to include the first time, all you need to do is reconfigure, do "fakeroot make-kpkg clean", and rebuild with a different value for --append-to-version. Your session should look something like this:

```
bash:~$ cd /usr/src
bash:/usr/src$ ln -s kernel-source-2.4.18 linux
bash:/usr/src$ cd linux
bash:/usr/src/linux$ make xconfig
                (reconfigure your kernel)
bash:/usr/src/linux$ fakeroot make-kpkg clean
                (lots of cleaning here)
bash:/usr/src/linux$ fakeroot make-kpkg --append-to-version=.030401 kernel_image
                (screens and screens of stuff)
```

8. Advanced topics

It may seem a bit strange to have a page for "Advanced topics" in a newbiedoc. We know that some of our readers aren't new to Linux or even to building kernels; they are trying to learn The Debian Way to administer their systems. Once you have successfully used kernel-package to build and install your own custom kernel, you may want to explore some ways to make life even better.

8.1. Using an existing configuration

When you installed your custom kernel, its configuration was copied to `/boot/config-x.y.z`. You may want to squirrel it away in another location (or several other locations) for safekeeping or later re-use.

Suppose that just last week you picked your way through the maze of options, and settled on a configuration that supports all the hardware and features you want. Today a new stable kernel is released and of course you want to upgrade right away. Is it necessary to start from scratch configuring the new kernel? Not at all.

Download and expand the new source tarball and make a new symlink. Then change directories to `/usr/src/linux`, copy your existing configuration there and do "make oldconfig":

```
bash:/usr/src/linux$ cp /boot/config-2.4.18.030320 .config
bash:/usr/src/linux$ make oldconfig
```

You'll be asked questions about new kernel options. It's generally safe to answer "no" to all the questions; make notes of the new options that interest you.

> *After you finish oldconfig you can run xconfig or menuconfig to review your selections, change your answers, or investigate the options you noted.*

Then do "fakeroot make-kpkg clean" and you're ready to build the latest stable kernel with your own configuration.

8.2. Kernel-package and grub

Although LILO is still Debian's default boot loader, grub is an outstanding alternative, and it works and plays well with make-kpkg.

After you've installed grub, run "grub-install /dev/hda" (substitute your boot device for "/dev/hda"). Then run "update-grub". Edit `/boot/grub/menu.lst` and substitute your defaults on the "#groot" and "#kopt" lines (and any other lines you need for your situation). Run "update-grub" again.

Create the file `/etc/kernel-img.conf` and insert the following lines:

postinst_hook=/sbin/update-grub
postrm_hook=/sbin/update-grub

Now whenever you install a new kernel image, update-grub will scan your `/boot` directory, inventory the kernels there, and write a new boot menu. If you took my suggestion and used "--append-to-version=.yymmdd", your kernels will be sorted by kernel version and date. The grub default is to start the first kernel in the list (your newest kernel), unless you choose a different one before the menu times out. When you remove a kernel, update-grub will remove its entry from the boot menu.

> *When you're comfortable with grub, do "dpkg -P lilo" and you won't be asked about lilo when you install a new kernel. :)*

8.3. Third-party modules

"Third-party modules" is the term I use for kernel modules with source code outside the kernel source tree. An example is "pcmcia-cs", which many laptop users need for the latest PCMCIA support.

To use pcmcia-cs, apt-get the "pcmcia-source" package; the source tarball should be in `/usr/src`. As a normal user, expand the source tarball to `/usr/src/modules/pcmcia-cs`. If you don't already have a modules directory one will be created.

When you configure your kernel, disable PCMCIA support. (If you have PCMCIA support enabled, pcmcia-cs won't build drivers.) If you want to use wireless LAN, enable it under

"Network device support" but don't select any devices.

When you are ready to build your kernel, add "modules_image" to your command line:

```
bash:/usr/src/linux$ fakeroot make-kpkg --append-to-version=.030401 kernel_image modules_image
```

(As you learn more and more the make-kpkg command line gets longer and longer.)

After your kernel-image .deb is made, make-kpkg will go to `/usr/src/modules` and build a modules .deb for each subdirectory, with the same version and --append-to-version as your kernel. Install the modules-image .debs after you install the kernel-image .deb.

That's it. Could this *get* any easier?

8.4. Adding third-party modules to an existing kernel

You've been monitoring the debian-user mailing list, and you notice that more and more people are interested in alsa (Advanced Linux Sound Architecture). You realize that your new sound card doesn't sound as good as it should, and you want to try alsa yourself. You fire up "make menuconfig" (you *are* more experienced now) but in the "Sound" section you don't see anything labelled "alsa". What to do?

Install the "alsa-source" package. By now you know to look for a new tarball in `/usr/src`, and there it is:

```
bash:/usr/src$ ls
alsa-driver.tar.bz2
kernel-image-2.4.18.030401_10.00.Custom_i386.deb
kernel-source-2.4.18
kernel-source-2.4.18.tar.bz2
modules
pcmcia-modules-2.4.18.030401_3.1.33-6+10.00.Custom_i386.deb
```

Expand the alsa-driver tarball and re-create the symlink. Note the --append-to-version value of your kernel image; you'll need it as an option for make-kpkg:

```
bash:/usr/src$ cd linux
bash:/usr/src/linux$ fakeroot make-kpkg --append-to-version=.030401
--added-modules=alsa-driver modules_image
```

Notice that you don't need the "kernel_image" target here. Unless you reconfigured the kernel, there's no need to rebuild it. Using the "--added-modules" option lets you build only the modules you need, instead of all the modules in `/usr/src/modules`.

> *If you want to build a modules .deb with a different --revision than the kernel, you may need to appease make-kpkg by doing "fakeroot make-kpkg clean".*

8.5. Debian kernel patches

Debian offers a wide variety of kernel patches, from the leading-edge innovative to the boringly safe. I like boringly safe, so I downloaded kernel-patch-debianlogo.

The debianlogo patch replaces Tux in the upper-left corner of your framebuffer boot console with another nice image. If you don't have a `kernel-patches` directory it will be created when you install your first kernel-patch package.

> *The `kernel-patches` directory is only for Debian kernel-patch packages. If you have other kernel patches they should go somewhere else.*

Installing a kernel-patch package does not apply the patch. That is done by invoking make-kpkg, with the option "--added-patches":

```
bash:/usr/src/linux$ fakeroot make-kpkg --append-to-version=.030401
--added-patches=debianlogo kernel_image modules_image
```

If you have several patches to apply list them after "--added-patches=", separated by commas.

By default make-kpkg runs "make oldconfig" after applying kernel patches. If your patches includes new kernel options, you will be asked about them. If you want to review the entire configuration instead, you can use yet another command-line option, "--config=menuconfig". When you save your configuration and exit menuconfig, make-kpkg will continue.

After the kernel image is built, make-kpkg reverses the patches you applied so that you begin each configure/build cycle with an unpatched source tree.

---

9. Checklist

1. Make backup boot diskette.
2. Install required packages.
   a. gcc, kernel-package, kernel-source-x.y.z, libc6-dev, tk8.3, libncurses5-dev, fakeroot
3. Setup source tree.
   a. Add yourself to src. Logout, login. Use `groups` to confirm.
   b. Remove old symlink.
   c. Expand source tarball.
   d. Make and check new symlink.
   e. Check Current Minimal Requirements.
4. Configure kernel.
   a. Use old config file if available. `cp /boot/config-x.y.z .config`
   b. `make oldconfig` or `make xconfig` or `make menuconfig`.
   c. Save `.config` file somewhere else for later use.
5. Build kernel-image package.
   a. `fakeroot make-kpkg clean`
   b. `fakeroot make-kpkg --append-to-version=alpha+numeric.but.no.underscore --added-modules=pcmcia-cs,alsa-driver --added-patches=debianlogo,other,patches --config=menuconfig kernel_image modules_image`
6. Install kernel-image package.

a. `dpkg -i kernel-image-x.y.z.yy.mm.dd_10.00.Custom_i386.deb`

b. Reboot.

7. What next?

a. Hold your package. `echo "kernel-image-x.y.z.yymmdd hold" | dpkg --set-selections`

b. Remove symlink.

c. Back up kernel image.

d. Remove old kernel image packages.

e. Make new boot diskette.