


```

Local version - append to kernel release (LOCALVERSION) []
Automatically append version information to the version string
(LOCALVERSION_AUTO) [Y/n/?] Y
...

```

The kernel configuration program will step through every configuration option and ask you if you wish to enable this option or not. Typically, your choices for each option are shown in the format `[Y/m/n/?]`. The capitalized letter is the default, and can be selected by just pressing the Enter key. The four choices are:

- y Build directly into the kernel.
- n Leave entirely out of the kernel.
- m Build as a module, to be loaded if needed.
- ? Print a brief descriptive message and repeat the prompt.

The kernel contains almost two thousand different configuration options, so being asked for every individual one will take a very long time. Luckily, there is an easier way to configure a kernel: base the configuration on a pre-built configuration.

Default Configuration Options

Every kernel version comes with a “default” kernel configuration. This configuration is loosely based on the defaults that the kernel maintainer of that architecture feels are the best options to be used. In some cases, it is merely the configuration that is used by the kernel maintainer himself for his personal machines. This is true for the i386 architecture, where the default kernel configuration matches closely what Linus Torvalds uses for his main development machine.

To create this default configuration, do the following:

```

$ cd linux-2.6.17.10
$ make defconfig

```

A huge number of configuration options will scroll quickly by the screen, and a `.config` file will be written out and placed in the kernel directory. The kernel is now successfully configured, but it should be customized to your machine in order to make sure it will operate correctly.

Modifying the Configuration

Now that we have a basic configuration file created, it should be modified to support the hardware you have present in the system. For details on how to find out which configuration options you need to select to achieve this, please see Chapter 7. Here we will show you how to select the options you wish to change.

There are three different interactive kernel configuration tools: a terminal-based one called *menuconfig*, a GTK+-based graphical one called *gconfig*, and a QT-based graphical one called *xconfig*.



Figure 4-2. Device Drivers option selected



Figure 4-3. Device Drivers submenu

If the option is selected with `Y`, the angle brackets will contain a `*` character. If it is selected as a module with an `M`, they will contain an `M` character. If it is disabled with `N`, they will show only a blank space.

So, if you wish to change these three options to select only drivers that do not need external firmware at compile time, disable the option to prevent firmware from being built, and build the userspace firmware loader as a module, press `Y` for the first option, `N` for the second option, and `M` for the third, making the screen look like Figure 4-5.

After you are done with your changes to this screen, press either the `Escape` key or the right arrow followed by the `Enter` key to leave this submenu. All of the different kernel options can be explored in this manner.

Graphical Configuration Methods

The *gconfig* and *xconfig* methods of configuring a kernel use a graphical program to allow you to modify the kernel configuration. The two methods are almost identical, the only difference being the different graphical toolkit with which they are written. *gconfig* is written using the GTK+ toolkit and has a two-pane screen looking like Figure 4-7.



Figure 4-7. make *gconfig* screen

The *xconfig* method is written using the QT toolkit and has a three-pane screen looking like Figure 4-8.

Use the mouse to navigate the submenus and select options. For instance, you can use it in Figure 4-8 to select the Generic Driver Options submenu of the Device Drivers menu. This will change the *xconfig* screen to look like Figure 4-9. The corresponding *gconfig* screen is Figure 4-10.

Changing this submenu to disable the second option and make the third option be built as a module causes the screens to look like Figures 4-11 and 4-12.

Please note that in the *gconfig* method, a checked box signifies that the option will be built into the kernel, whereas a line through the box means the option will be built as a module. In the *xconfig* method, an option built as a module will be shown with a dot in the box.

Both of these methods prompt you to save your changed configuration when exiting the program, and offer the option to write that configuration out to a different file. In that way you can create multiple, differing configurations.

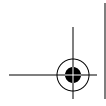


Figure 4-9. make xconfig Generic Driver Options

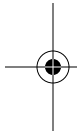


Figure 4-10. make gconfig Generic Driver Options




```
CHK    include/linux/compile.h
UPD    include/linux/compile.h
CC     init/version.o
CC     init/do_mounts.o
...
```

Running *make* causes the kernel build system to use the configuration you have selected to build a kernel and all modules needed to support that configuration.* While the kernel is building, *make* displays the individual filenames of what is currently happening, along with any build warnings or errors.

If the kernel build finished without any errors, you have successfully created a kernel image. However, it needs to be installed properly before you try to boot from it. See Chapter 5 for how to do this.

It is very unusual to get any build errors when building a released kernel version. If you do, please report them to the Linux kernel developers so they can be fixed.

Advanced Building Options

The kernel build system allows you to do many more things than just build the full kernel and modules. Chapter 10 includes the full list of options that the kernel build system provides. In this section, we will discuss some of these advanced build options. To see a full description of how to use other advanced build options, refer to the in-kernel documentation on the build system, which can be found in the *Documentation/kbuild* directory of the sources.

Building Faster on Multiprocessor Machines

The kernel build system works very well as a task that can be split up into little pieces and given to different processors. By doing this, you can use the full power of a multiprocessor machine and reduce the kernel build time considerably.

To build the kernel in a multithreaded way, use the *-j* option to the *make* program. It is best to give a number to the *-j* option that corresponds to twice the number of processors in the system. So, for a machine with two processors present, use:

```
$ make -j4
```

and for a machine with four processors, use:

```
$ make -j8
```

If you do not pass a numerical value to the *-j* option:

```
$ make -j
```

the build system will create a new thread for every subdirectory in the kernel tree, which can easily cause your machine to become unresponsive and take a much longer time to complete the build. Because of this, it is recommended that you always pass a number to the *-j* option.

* Older kernel versions prior to the 2.6 release required the additional step of *make modules* to build all needed kernel modules. That is no longer required.

Different Architectures

A very useful feature is building the kernel in a cross-compiled manner to allow a more powerful machine to build a kernel for a smaller embedded system, or just to check a build for a different architecture to ensure that a change to the source code did not break something unexpected. The kernel build system allows you to specify a different architecture from the current system with the `ARCH=` argument. The build system also allows you to specify the specific compiler that you wish to use for the build by using the `CC=` argument or a cross-compile toolchain with the `CROSS_COMPILE` argument.

For example, to get the default kernel configuration of the `x86_64` architecture, you would enter:

```
$ make ARCH=x86_64 defconfig
```

To build the whole kernel with an ARM toolchain located in `/usr/local/bin/`, you would enter:

```
$ make ARCH=arm CROSS_COMPILE=/usr/local/bin/arm-linux-
```

It is useful even for a non-cross-compiled kernel to change what the build system uses for the compiler. Examples of this are using the `distcc` or `ccache` programs, both of which help greatly reduce the time it takes to build a kernel. To use the `ccache` program as part of the build system, enter:

```
$ make CC="ccache gcc"
```

To use both `distcc` and `ccache` together, enter:

```
$ make CC="ccache distcc"
```