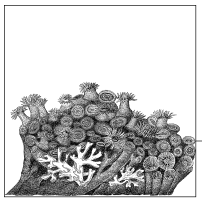

10

Kernel Build Command-Line Reference

As discussed in Chapter 4, the tool that ties together kernel builds is the *make* program, to which you pass a target that specifies what you want to build. Chapter 4 went over the basic targets needed to build the kernel properly, but the kernel build system also has a wide range of other targets. This chapter details these targets, and what they can be used for.

All of these targets are passed to the *make* program on the command line, and a number of them can be grouped together if desired. For example:

```
$ make mrproper xconfig
```

The targets are broken down into different types in the following sections.

You can get a summary of most of these targets by running, within the build directory:

```
$ make help
```

This target prints out a lot of the common *make* targets that are described in the rest of this chapter.

Informational Targets

Table 10-1 shows targets that print the kernel version, based on a number of different options. They are commonly used by scripts to determine the version of the kernel being built.

Table 10-1. Informational targets

Target	Description
`kernelrelease`	Displays the current kernel version, as determined by the build system.
`kernelversion`	Displays the current kernel version, as told by the main *Makefile*. This differs from the `kernelrelease` target in that it doesn't use any additional version information based on configuration options or *localversion* files.

Cleaning Targets

Table 10-2 shows targets that simply remove files from previous builds. Their use is highly recommended to make sure you don't contaminate new builds with files leftover that may have been built with different options. They differ in how much they remove; sometimes you want to keep around files you've changed.

Table 10-2. Cleaning targets

Target	Description
`clean`	Removes most of the files generated by the kernel build system, but keeps the kernel configuration.
`mrproper`	Removes all of the generated files by the kernel build system, including the configuration and some various backup files.
`distclean`	Does everything `mrproper` does and removes some editor backup and patch leftover files.

Configuration Targets

Table 10-3 shows targets that allow the kernel to be configured in a wide range of different ways.

Table 10-3. Configuration targets

Target	Description
`config`	Updates the current kernel configuration by using a line-oriented program.
`menuconfig`	Updates the current kernel configuration by using a text-based menu program.
`xconfig`	Updates the current kernel configuration by using a QT-based graphical program.
`gconfig`	Updates the current kernel configuration by using a GTK+-based graphical program.
`oldconfig`	Updates the current kernel configuration by using the current *.config* file and prompting for any new options that have been added to the kernel.
`silentoldconfig`	Just like `oldconfig`, but prints nothing to the screen except when a question needs to be answered.
`randconfig`	Generates a new kernel configuration with random answers to all of the different options.
`defconfig`	Generates a new kernel configuration with the default answer being used for all options. The default values are taken from a file located in the *arch/$ARCH/defconfig* file, where *$ARCH* refers to the specific architecture for which the kernel is being built.
`allmodconfig`	Generates a new kernel configuration in which modules are enabled whenever possible.
`allyesconfig`	Generates a new kernel configuration with all options set to `yes`.
`allnoconfig`	Generates a new kernel configuration with all options set to `no`.

Note that the `allyesconfig`, `allmodconfig`, `allnoconfig`, and `randconfig` targets also take advantage of the environment variable `KCONFIG_ALLCONFIG`. If that variable points to a file, that file will be used as a list of configuration values that you require to be set to a specific value. In other words, the file overrides the normal behavior of the *make* targets.

For example, if the file *~/linux/must_be_set* contains the following variables:

```
$ cat ~/linux/must_be_set
CONFIG_SWAP=y
CONFIG_DEBUG_FS=y
```

and you enter *make allnoconfig* with the proper KCONFIG_ALLCONFIG environment variable in effect:

```
$ KCONFIG_ALLCONFIG=../must_be_set make allnoconfig
$ grep CONFIG_SWAP .config
CONFIG_SWAP=y
```

then the results include:

```
$ grep CONFIG_DEBUG_FS .config
CONFIG_DEBUG_FS=y
```

This variable would not have normally been set to y otherwise.

If the KCONFIG_ALLCONFIG variable is not set, the build system checks for files in the top-level build directory named:

- *allmod.config*
- *allno.config*
- *allrandom.config*
- *allyes.config*

If any of those files are present, the build uses them as lists of configuration values that must be forced to the specified values. If none of those files are found, the build system finally looks for a file called *all.config* for a list of forced configuration values.

You can use these different files to set up a known good base configuration that will always work. Then the other configuration options can be used to generate different testing configurations for the needed situation.

Build Targets

Table 10-4 shows targets that build the kernel itself in a variety of ways.

Table 10-4. Build targets

Target	Description
all	Builds all of the different targets needed for this kernel to be able to be used. This includes both the modules and the static portion of the kernel.
vmlinux	Builds just the static portion of the kernel, not any loadable modules.
modules	Builds all of the loadable kernel modules for this configuration.
modules_install	Installs all of the modules into the specified location. If no location is specified with the INSTALL_MODULE_PATH environment variable, they are installed in the default root directory of the machine.
dir/	Builds all of the files in the specified directory and in all subdirectories below it.
dir/file.[o|i|s]	Builds only the specified file.
dir/file.ko	Builds all of the needed files and links them together to form the specified module.
tags	Builds all of the needed tags that most common text editors can use while editing the source code.

Table 10-4. Build targets (continued)

Target	Description
TAGS	Builds all of the needed tags that most common text editors can use while editing the source code.
cscope	Builds a *cscope* image, useful in source tree searches, of the source tree for the architecture specified by the configuration file (not all of the kernel source files).

You can also pass a number of environment variables to *make* that will change the build. These can be specified for almost any target, as shown in Table 10-5.

Table 10-5. Environment variables

Variable	Value	Description
V	0	This tells the build system to run in a quiet manner, showing only the file that is currently being built, and not the entire command that is running in order to build that file. This is the default option for the build system.
V	1	This tells the build system to operate in a verbose way, showing the full command that is being used to generate each of the specific files.
O	dir	This tells the build system to locate all output files in the *dir* directory, including the kernel configuration files. This allows the kernel to be built from a read-only filesystem and have the output placed in another location.
C	1	This checks all C files that are about to be built with the *sparse* tool, which detects common programming errors in the kernel source files. *sparse* can be downloaded using *git* from *git://git.kernel.org/pub/scm/devel/sparse/sparse.git*. Nightly snapshots can be found at *http://www.codemonkey.org.uk/projects/git-snapshots/sparse/*. More information on how to use *sparse* can be found in the *Documentation/sparse.txt* file in the kernel source tree.
C	2	This forces all C files to be checked with the *sparse* tool, even if they did not need to be built.

Packaging Targets

These targets package up a built kernel into a standalone package that can be installed on a wide range of different machines, as shown in Table 10-6.

Table 10-6. Packaging targets

Target	Description
rpm	Builds the kernel first and then packages it up as a RPM package that can be installed.
rpm-pkg	Builds a source RPM package containing the base kernel.
binrpm-pkg	Builds a RPM package that contains a compiled kernel and modules.
deb-pkg	Builds a Debian package that contains the compiled kernel and modules.
tar-pkg	Builds a tarball that contains the compiled kernel and modules.
targz-pkg	Builds a *gzip*-compressed tarball that contains the compiled kernel and modules.
tarbz2-pkg	Builds a *bzip2*-compressed tarball that contains the compiled kernel and modules.

Documentation Targets

Table 10-7 shows targets that build the internal kernel documentation in a variety of different formats.

Table 10-7. Documentation targets

Target	Description
`xmldocs`	Builds the kernel documentation as XML DocBook files.
`psdocs`	Builds the kernel documentation as PostScript files.
`pdfdocs`	Builds the kernel documentation as PDF files.
`htmldocs`	Builds the kernel documentation as HTML files.
`mandocs`	Builds the kernel documentation as a set of manpages, which can then be installed with the *install-mandocs* target.

Architecture-Specific Targets

Each kernel architecture has a set of specific targets unique to it. Table 10-8 shows the targets available for the 32-bit Intel architecture.

Table 10-8. 32-bit Intel architecture-specific targets

Target	Description
`bzImage`	Creates a compressed kernel image and places it in the *arch/i386/boot/bzImage* file. This is the default target for the i386 kernel build.
`install`	Installs the kernel image using the distribution-specific */sbin/installkernel* program. Note that this does not install the kernel modules; that must be done with the *modules_install* target.
`bzdisk`	Creates a boot floppy image and writes it to the */dev/fd0* device.
`fdimage`	Creates a boot floppy image and places it in the file *arch/i386/boot/fdimage*. The *mtools* package must be present on your system in order for this to work properly.
`isoimage`	Creates a CD-ROM boot image and places it in the file *arch/i396/boot/image.iso*. The *syslinux* package must be present on your system in order for this to work properly.

Analysis Targets

Table 10-9 shows targets that are good for trying to find problem code in the kernel. It's a good idea to create a stack space list when creating new code to determine that your changes are not taking up too much kernel stack space. The `namespacecheck` target is useful for determining whether your changes can safely add its symbols to the kernel's global namespace.

Table 10-9. Analysis targets

Target	Description
`checkstack`	Generate a list of the functions that use the most of the kernel stack space.
`namespacecheck`	Generate a list of all of the kernel symbols and their namespaces. This will be a large list.