8.10. Compiling a Kernel

debian-handbook.info/browse/stable/sect.kernel-compilation.html

The kernels provided by Debian include the largest possible number of features, as well as the maximum of drivers, in order to cover the broadest spectrum of existing hardware configurations. This is why some users prefer to recompile the kernel in order to only include what they specifically need. There are two reasons for this choice. First, it may be to optimize memory consumption, since the kernel code, even if it is never used, occupies memory for nothing (and never “goes down” on the swap space, since it is actual RAM that it uses), which can decrease overall system performance. A locally compiled kernel can also limit the risk of security problems since only a fraction of the kernel code is compiled and run.

***NOTE* Security updates**

If you choose to compile your own kernel, you must accept the consequences: Debian cannot ensure security updates for your custom kernel. By keeping the kernel provided by Debian, you benefit from updates prepared by the Debian Project's security team.

Recompilation of the kernel is also necessary if you want to use certain features that are only available as patches (and not included in the standard kernel version).

8.10.1. Introduction and Prerequisites

Unsurprisingly Debian manages the kernel in the form of a package, which is not how kernels have traditionally been compiled and installed. Since the kernel remains under the control of the packaging system, it can then be removed cleanly, or deployed on several machines. Furthermore, the scripts associated with these packages automate the interaction with the bootloader and the initrd generator.

The upstream Linux sources contain everything needed to build a Debian package of the kernel. But you still need to install build-essential to ensure that you have the tools required to build a Debian package. Furthermore, the configuration step for the kernel requires the libncurses5-dev package. Finally, the fakeroot package will enable creation of the Debian package without using administrator's rights.

***CULTURE* The good old days of kernel-package**

Before the Linux build system gained the ability to build proper Debian packages, the recommended way to build such packages was to use `make-kpkg` from the kernel-package package.

8.10.2. Getting the Sources

Like anything that can be useful on a Debian system, the Linux kernel sources are available in a package. To retrieve them, just install the linux-source-*version* package. The `apt search ^linux-source` command lists the various kernel versions packaged by Debian. The latest version is available in the Unstable distribution: you can retrieve them without much risk (especially if your APT is configured according to the instructions of Section 6.2.6, “Working with Several Distributions”). Note that the source code contained in these packages does not correspond precisely with that published by Linus Torvalds

and the kernel developers; like all distributions, Debian applies a number of patches, which might (or might not) find their way into the upstream version of Linux. These modifications include backports of fixes/features/drivers from newer kernel versions, new features not yet (entirely) merged in the upstream Linux tree, and sometimes even Debian specific changes.
The remainder of this section focuses on the 4.19 version of the Linux kernel, but the examples can, of course, be adapted to the particular version of the kernel that you want.

We assume the linux-source-4.19 package has been installed. It contains `/usr/src/linux-source-4.19.tar.xz` , a compressed archive of the kernel sources. You must extract these files in a new directory (not directly under `/usr/src/` , since there is no need for special permissions to compile a Linux kernel): `~/kernel/` is appropriate.

```
$ mkdir ~/kernel; cd ~/kernel
$ tar -xaf /usr/src/linux-source-4.19.tar.xz
```

***CULTURE* Location of kernel sources**

Traditionally, Linux kernel sources would be placed in `/usr/src/linux/` thus requiring root permissions for compilation. However, working with administrator rights should be avoided when not needed. There is a `src` group that allows members to work in this directory, but working in `/usr/src/` should be avoided, nevertheless. By keeping the kernel sources in a personal directory, you get security on all counts: no files in `/usr/` unknown to the packaging system, and no risk of misleading programs that read `/usr/src/linux` when trying to gather information on the used kernel.

8.10.3. Configuring the Kernel

The next step consists of configuring the kernel according to your needs. The exact procedure depends on the goals.

When recompiling a more recent version of the kernel (possibly with an additional patch), the configuration will most likely be kept as close as possible to that proposed by Debian. In this case, and rather than reconfiguring everything from scratch, it is sufficient to copy the `/boot/config-version` file (the version is that of the kernel currently used, which can be found with the `uname -r` command) into a `.config` file in the directory containing the kernel sources.

```
$ cp /boot/config-4.19.0-5-amd64 ~/kernel/linux-source-4.19/.config
```

Unless you need to change the configuration, you can stop here and skip to Section 8.10.4, “Compiling and Building the Package”. If you need to change it, on the other hand, or if you decide to reconfigure everything from scratch, you must take the time to configure your kernel. There are various dedicated interfaces in the kernel source directory that can be used by calling the `make target` command, where *target* is one of the values described below.
`make menuconfig` compiles and executes a text-mode interface (this is where the libncurses5-dev package is required) which allows navigating the options available in a hierarchical structure. Pressing the **Space** key changes the value of the selected option, and **Enter** validates the button selected at the bottom of the screen; **Select** returns to the selected sub-menu; **Exit** closes the current screen and moves back up in the hierarchy; **Help** will display more detailed information on the role of the selected option.

The arrow keys allow moving within the list of options and buttons. To exit the configuration program, choose **Exit** from the main menu. The program then offers to save the changes you've made; accept if you are satisfied with your choices.

Other interfaces have similar features, but they work within more modern graphical interfaces; such as `make xconfig` which uses a Qt graphical interface, and `make gconfig` which uses GTK+. The former requires libqt4-dev, while the latter depends on libglade2-dev and libgtk2.0-dev.

When using one of those configuration interfaces, it is always a good idea to start from a reasonable default configuration. The kernel provides such configurations in `arch/arch/configs/*_defconfig` and you can put your selected configuration in place with a command like `make x86_64_defconfig` (in the case of a 64-bit PC) or `make i386_defconfig` (in the case of a 32-bit PC).

TIP **Dealing with outdated `.config` files**

When you provide a `.config` file that has been generated with another (usually older) kernel version, you will have to update it. You can do so with `make oldconfig`, it will interactively ask you the questions corresponding to the new configuration options. If you want to use the default answer to all those questions you can use `make olddefconfig`. With `make oldnoconfig`, it will assume a negative answer to all questions.

8.10.4. Compiling and Building the Package

NOTE **Clean up before rebuilding**

If you have already compiled once in the directory and wish to rebuild everything from scratch (for example, because you substantially changed the kernel configuration), you will have to run `make clean` to remove the compiled files. `make distclean` removes even more generated files, including your `.config` file too, so make sure to backup it first. If you copied the configuration from `/boot/`, you must change the system trusted keys option, providing an empty string is enough: `CONFIG_SYSTEM_TRUSTED_KEYS = ""`.

Once the kernel configuration is ready, a simple `make deb-pkg` will generate up to 5 Debian packages: linux-image-*version* that contains the kernel image and the associated modules, linux-headers-*version* which contains the header files required to build external modules, linux-firmware-image-*version* which contains the firmware files needed by some drivers (this package might be missing when you build from the kernel sources provided by Debian), linux-image-*version*-dbg which contains the debugging symbols for the kernel image and its modules, and linux-libc-dev which contains headers relevant to some user-space libraries like GNU glibc.

The *version* is defined by the concatenation of the upstream version (as defined by the variables `VERSION`, `PATCHLEVEL`, `SUBLEVEL` and `EXTRAVERSION` in the `Makefile`), of the `LOCALVERSION` configuration parameter, and of the `LOCALVERSION` environment variable. The package version reuses the same version string with an appended revision that is regularly incremented (and stored in `.version`), except if you override it with the `KDEB_PKGVERSION` environment variable.

```
$ make deb-pkg LOCALVERSION=-falcot KDEB_PKGVERSION=$(make kernelversion)-1
[...]
$ ls ../*.deb
../linux-headers-4.19.37-falcot_4.19.37-1_amd64.deb
../linux-image-4.19.37-falcot_4.19.37-1_amd64.deb
../linux-libc-dev_4.19.37-1_amd64.deb
```

8.10.5. Compiling External Modules

Some modules are maintained outside of the official Linux kernel. To use them, they must be compiled alongside the matching kernel. A number of common third party modules are provided by Debian in dedicated packages, such as vpb-driver-source (extra modules for Voicetronix telefony hardware) or leds-alix-source (driver of PCEngines ALIX 2/3 boards).

These packages are many and varied, `apt-cache rdepends module-assistant$` can show the list provided by Debian. However, a complete list isn't particularly useful since there is no particular reason for compiling external modules except when you know you need it. In such cases, the device's documentation will typically detail the specific module(s) it needs to function under Linux.

For example, let's look at the dahdi-source package: after installation, a `.tar.bz2` of the module's sources is stored in `/usr/src/`. While we could manually extract the tarball and build the module, in practice we prefer to automate all this using DKMS. Most modules offer the required DKMS integration in a package ending with a `-dkms` suffix. In our case, installing dahdi-dkms is all that is needed to compile the kernel module for the current kernel provided that we have the linux-headers-* package matching the installed kernel. For instance, if you use linux-image-amd64, you would also install linux-headers-amd64.

```
$ sudo apt install dahdi-dkms

[...]
Setting up xtables-addons-dkms (2.12-0.1) ...
Loading new xtables-addons-2.12 DKMS files...
Building for 4.19.0-5-amd64
Building initial module for 4.19.0-5-amd64
Done.

dahdi_dummy.ko:
Running module version sanity check.
 - Original module
   - No original module exists within this kernel
 - Installation
   - Installing to /lib/modules/4.19.0-5-amd64/updates/dkms/
[...]
DKMS: install completed.
$ sudo dkms status
dahdi, DEB_VERSION, 4.19.0-5-amd64, x86_64: installed
$ sudo modinfo dahdi_dummy
filename:       /lib/modules/4.19.0-5-amd64/updates/dkms/dahdi_dummy.ko
license:        GPL v2
author:         Robert Pleh <robert.pleh@hermes.si>
description:    Timing-Only Driver
[...]
```

***ALTERNATIVE* module-assistant**

Before DKMS, module-assistant was the simplest solution to build and deploy kernel modules. It can still be used, in particular for packages lacking DKMS integration: with a simple command like `module-assistant auto-install dadhi` (or `m-a a-i dahdi` for short), the modules are compiled for the current kernel, put in a new Debian package, and that package gets installed on the fly.

8.10.6. Applying a Kernel Patch

Some features are not included in the standard kernel due to a lack of maturity or to some disagreement with the kernel maintainers. Such features may be distributed as patches that anyone is then free to apply to the kernel sources.

Debian sometimes provides some of these patches in linux-patch-* packages, but they often don't make it into stable releases (sometimes for the very same reasons that they are not merged into the official upstream kernel). These packages install files in the `/usr/src/kernel-patches/` directory.

To apply one or more of these installed patches, use the `patch` command in the sources directory then start compilation of the kernel as described above.

```
$ cd ~/kernel/linux-source-4.9
$ make clean
$ zcat /usr/src/kernel-patches/diffs/grsecurity2/grsecurity-3.1-4.9.11-201702181444.patch.gz |
patch -p1
```

Note that a given patch may not necessarily work with every version of the kernel; it is possible for `patch` to fail when applying them to kernel sources. An error message will be displayed and give some details about the failure; in this case, refer to the documentation available in the Debian package of the patch (in the `/usr/share/doc/linux-patch-*/` directory). In most cases, the maintainer indicates for which kernel versions their patch is intended.