Canonical

Ubuntu Kernel

Canonical

Contents

The Ubuntu Linux kernel is the core software enabling applications on Ubuntu to interact with system resources.

The Ubuntu kernel handles communication between system hardware and user-space applications, managing tasks like memory, processing, and security. Regular stable release updates (SRU) ensure the kernel stays secure, stable, and optimised.

Ubuntu kernels provide a reliable foundation for applications and system processes, meeting the need for secure, high-performance, Ubuntu environments. Kernels are also tested consistently for regressions to provide users with a reliable and smooth experience. Kernels are tailor made for Ubuntu Desktop, Ubuntu Server, a wide range of architectures, IoT devices, cloud providers, and more.

This documentation serves developers, partners, and others working with Ubuntu kernels, offering guidance on kernel workflows, tools, SRU timelines, and processes for customisation and maintenance.

---

1. In this documentation

Tutorials (page 2) **Start here**: a hands-on introduction to Example Product for new users

How-to guides (page 4) **Step-by-step guides** covering common tasks involved in kernel development.

Reference (page 2) **Technical information** - specifications, APIs, architecture

Explanation (page 13) **Discussion and clarification** about Ubuntu Linux kernel source repositories.

---

2. Project and community

Example Project is a member of the Ubuntu family. It's an open source project that warmly welcomes community projects, contributions, suggestions, fixes and constructive feedback.

- Code of conduct
- Get support
- Join our online chat
- Contribute
- Roadmap
- Thinking about using Example Product for your next project? Get in touch!

2.1. How-to guides

These guides accompany through the various stages and building and publishing kernel packages and components.

2.1.1. Build and publish

The steps to build a kernel is similar but may have slightly difference configuration requirements on different platforms and/or architectures.

Preparation

These guides cover processes related to obtaining kernel source trees and preparing the kernel before the build process.

How to obtain kernel source for an Ubuntu release using Git

The kernel source code for each Ubuntu release is maintained in its own repository in Launchpad. Downloading the kernel source may be needed for customisation, development, or troubleshooting the kernel.

This document shows how you can obtain the kernel source for an Ubuntu release using Git.

Prerequisites

You must have the git package[1] installed on your system.

```
sudo apt-get install git
```

[1] https://packages.ubuntu.com/search?keywords=git

Get local copy of kernel source for single release

You can use `git clone` with the selected protocol to obtain a local copy of the kernel source for the release you are interested in.

For example, to obtain a local copy of the Jammy kernel tree, run any of the following `git clone` commands:

```
git clone git://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
git clone git+ssh://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
git clone https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
```

See *Protocols for accessing kernel sources* (page 13) for more information.

Related topics

- *About Ubuntu Linux kernel sources* (page 13)
- https://wiki.ubuntu.com/Kernel/Dev/KernelGitGuide

Build and publish kernel

These guides cover how to build kernel packages, kernel snaps, and kernel components.

How to build an Ubuntu Linux kernel

If you have patches you need to apply to the Ubuntu Linux kernel, or you want to change some kernel configs, you may need to build your kernel from source. Follow these steps to customise and build the Ubuntu Linux kernel locally.

Prerequisites

This guide supports Xenial Xerus and newer.

It is recommended to have 8GB of RAM or more to build the Linux kernel.

If this is the first time you are building a kernel on your system, you will need to *set up your build environment* (page 6) and *install the required packages* (page 6).

Otherwise, skip ahead to *Obtain the source for an Ubuntu release* (page 6).

Set up build environment

To build an Ubuntu kernel, you will need enable the necessary source repositories in the `sources.list` or `ubuntu.sources` file.

Noble Numbat 24.04 (and newer)

Mantic Minotaur 23.10 (and older)

Add “deb-src” to the `Types:` line in the `/etc/apt/sources.list.d/ubuntu.sources` file.

```
Types: deb deb-src
URIs: http://archive.ubuntu.com/ubuntu
Suites: noble noble-updates noble-backports
Components: main universe restricted multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
```

Check that you have the following entries in the `/etc/apt/sources.list` file. If not, add or uncomment these lines for your Ubuntu release.

```
deb-src http://archive.ubuntu.com/ubuntu jammy main
deb-src http://archive.ubuntu.com/ubuntu jammy-updates main
```

Install required packages

To install the required packages and build dependencies, run:

```
sudo apt update
sudo apt build-dep -y linux linux-image-unsigned-$(uname -r)
sudo apt install -y fakeroot llvm libncurses-dev dwarves
```

Obtain the source for an Ubuntu release

There are different ways to get the kernel sources, depending on the kernel version you want to make changes to.

Get kernel source for version installed on build machine

Use the `apt source` command to get the source code for the kernel version currently running on your build machine.

```
apt source linux-image-unsigned-$(uname -r)
```

This will download and unpack the kernel source files to your current working directory.

```
<working directory>
├── linux-X.Y.Z/
│   └── *
├── linux_X.Y.Z-*.diff.gz
├── linux_X.Y.Z-*.dsc
└── linux_X.Y.Z.orig.tar.gz
```

Get kernel source for other versions

Use Git to get the source code for other kernel versions. See *How to obtain kernel source for an Ubuntu release using Git* (page 4) for detailed instructions.

Prepare the kernel source

Once you have the kernel source, go to the kernel source working directory (e.g. `linux-6.8.0`) and run the following commands to ensure you have a clean build environment and the necessary scripts have execute permissions:

```
cd <kernel source working directory>
chmod a+x debian/rules
chmod a+x debian/scripts/*
chmod a+x debian/scripts/misc/*
fakeroot debian/rules clean
```

Modify ABI number

You should modify the kernel version number to avoid conflicts and to differentiate the development kernel from the kernel released by Canonical.

To do so, modify the ABI number (the number after the dash following the kernel version) to "999" in the first line of the `<kernel source working directory>/debian.master/changelog` file.

For example, modify the ABI number to "999" for Noble Numbat:

```
linux (6.8.0-999.48) noble; urgency=medium
```

If you are building something other than the generic Ubuntu Linux kernel, modify the ABI number in the `<kernel source working directory>/debian.<derivative>/changelog` file instead.

Modify kernel configuration

(Optional) To enable or disable any features using the kernel configuration, run:

```
cd <kernel source working directory>
fakeroot debian/rules editconfigs
```

This will invoke the `menuconfig` interface for you to edit specific configuration files related to the Ubuntu kernel package. You will need to explicitly respond with `Y` or `N` when making any config changes to avoid getting errors later in the build process.

Customise the kernel

(Optional) Add any firmware, binary blobs, or patches as needed.

Build the kernel

You are now ready to build the kernel.

```
cd <kernel source working directory>
fakeroot debian/rules clean
fakeroot debian/rules binary
```

> **Note**
>
> Run `fakeroot debian/rules clean` to clean the build environment each time before you recompile the kernel after making any changes to the kernel source or configuration.

If the build is successful, several .deb binary package files will be produced in the directory one level above the kernel source working directory.

For example, building a kernel with version “6.8.0-999.48” on an x86-64 system will produce the following .deb packages (and more):

- `linux-headers-6.8.0-999_6.8.0-999.48_all.deb`
- `linux-headers-6.8.0-999-generic_6.8.0-999.48_amd64.deb`
- `linux-image-unsigned-6.8.0-999-generic_6.8.0-999.48_amd64.deb`
- `linux-modules-6.8.0-999-generic_6.8.0-999.48_amd64.deb`

Install the new kernel

Install all the debian packages generated from the previous step (on your build system or a different target system with the same architecture) with `dpkg -i` and reboot:

```
cd <kernel source working directory>/../
sudo dpkg -i linux-headers-<kernel version>*_all.deb
sudo dpkg -i linux-headers-<kernel version>-<generic or derivative>*.deb
sudo dpkg -i linux-image-unsigned-<kernel version>-<generic or derivative>*.deb
sudo dpkg -i linux-modules-<kernel version>-<generic or derivative>*.deb
sudo reboot
```

Test the new kernel

Run any necessary testing to confirm that your changes and customisations have taken effect. You should also confirm that the newly installed kernel version matches the value in the `<kernel source working directory>/debian.master/changelog` file by running:

```
uname -r
```

How to build an Ubuntu Linux kernel snap

If you are running an Ubuntu Core system and want to use boot into a custom kernel, you will need a kernel snap.

This guide shows how to build a kernel snap for local development and testing.

> **Important**
>
> Kernel snaps built using the method described here is not intended for us in production.

Prerequisites

Before you begin, you will need:

- A Launchpad account
- To be part of the Launchpad team that owns the project (for private repositories)
- A build machine running Ubuntu
- A device running an Ubuntu Core image with “dangerous” model assertion grade to install the custom kernel snap

> **Note**
>
> The Ubuntu version of the build host must match the version of the device where the kernel snap will be installed. For example, use an Ubuntu 22.04 (Jammy) host to build the kernel snap for an Ubuntu Core 22 device.
> See Snap - Build environment options[2] for more information.

[2] https://snapcraft.io/docs/build-options#heading--snapcraft

Set up build environment

Set up the host machine which will be used to build the kernel snap.

Install snapcraft

Snapcraft is used to create a managed environment to build the kernel snap. You are recommended to use the latest/stable version of the snapcraft snap from the Snap Store.

On the build machine, remove any existing snapcraft debian package and install snapcraft by running:

```
sudo apt-get update
sudo apt-get -y upgrade
sudo apt purge -y snapcraft
sudo snap install snapcraft --classic
```

Configure source repositories

Configure the package source repositories for the host architecture by specifying the architecture (e.g. "[arch=amd64]" for x86-64 hosts) for each deb source list in the data sources file.

Ubuntu 24.04 (Noble) and newer

Ubuntu 23.10 (Mantic) and older

Update the `/etc/apt/sources.list.d/ubuntu.sources` file.

For example, on a x86-64 host running Ubuntu 24.04 (Noble):

```
[...]
Types: deb deb-src
URIs: http://archive.ubuntu.com/ubuntu
Suites: noble noble-updates noble-backports
Components: main universe restricted multiverse
Architectures: amd64
[...]
```

Update the `/etc/apt/sources.list` file. For example, on a x86-64 host running Ubuntu 22.04 (Jammy):

```
deb [arch=amd64] http://archive.ubuntu.com/ubuntu focal main restricted
```

Alternatively, if you are running a default installation of Ubuntu, you can do a global update of all sources in the `/etc/apt/sources.list` file.

```
sudo sed -ie 's/deb http/deb [arch=amd64] http/g' /etc/apt/sources.list
```

Add support for cross-compilation

Add the target architecture (e.g. “arm64”) to the list of supported architectures. This step is only required if the build machine is running on a different architecture than the target device for the kernel snap.

For example, if you want to build a kernel snap for an ARM64 device on a x86-64 host, run:

```
sudo dpkg --add-architecture arm64
sudo apt update
```

Confirm that support for the target architecture has been added successfully by running `dpkg --print-foreign-architectures`:

```
user@host:~$ dpkg --print-foreign-architectures arm64
```

Configure SSH settings for Launchpad access

Enable SSH access to `git.launchpad.net` for your Launchpad account. This step is only required if you are building a snap from a private repository in Launchpad.

Add the following in the `~/.ssh/config` file:

```
Host git.launchpad.net
  User <your Launchpad username>
```

Clone the kernel snap recipe

Once you have set up your host machine, clone the Ubuntu Linux kernel snap recipe.

```
git clone <kernel-source-repository>
```

Customise the kernel

Add any firmware or binary blobs, or customise `initrd` as needed. This step is only required if you want to make your own changes to the kernel.

Build the kernel snap

You are now ready to build the kernel snap.

1. Go to the directory with the cloned kernel repository.

   ```
   cd <kernel-source-repository>
   ```

2. Create an alias for the `snapcraft.yaml` file. This is only required if there are multiple YAML configuration files in the `snap/local/` tree.

```
ln -s snap/local/<project>.yaml snapcraft.yaml
```

3. (Optional) Add the `sed` command in the `snapcraft.yaml` file to set the Kconfig value `CONFIG_MODULE_SIG_ALL` to `n` for your target architecture. This allows unverified modules to be loaded into the kernel and should only be set to `n` for local testing and development.

   For example, if the kernel snap is for an ARM64 device, set `'arm64': 'n'`:

```
[...]
parts:
  kernel:
    override-build: |
      [...]
      # override configs
      sed -i "s/^\(CONFIG_MODULE_SIG_FORCE\).*/\\1 policy\<{'arm64': 'n', 'armhf': 'n
'}\>/" ${DEBIAN}/config/annotations
      sed -i "s/^\(CONFIG_MODULE_SIG_ALL.*\)'arm64': 'y'\(.*\)/\\1'arm64': 'n'\\2/" $
{DEBIAN}/config/annotations
      [...]
```

4. Build the kernel snap package.

   UC24 and UC22

   UC20

```
sudo snapcraft --build-for=arm64 --destructive-mode
```

```
sudo snapcraft --target-arch=arm64 --destructive-mode --enable-experimental-target-
arch
```

5. You should get a `<name>_<version>_<arch>.snap` file in the kernel repository root, where:

   - `<name>` is the identified set in `snapcraft.yaml`
   - `<version>` is the kernel version
   - `<arch>` is the target architecture for the kernel snap

6. Copy the kernel snap to your target device and reboot into latest kernel to verify your changes.

```
snap install --dangerous --devmode <name>_<version>_<arch>.snap
```

> **Note**
>
> Local snaps can only be installed if the Ubuntu Core image on the target device was created with a model assertion that specifies the “dangerous” grade.

2.2. About Ubuntu Linux kernel sources

Ubuntu Linux kernel source packages are essential for users and developers who want to build, modify, or understand the kernel that powers Ubuntu systems. These packages are stored in Launchpad and organised by series (or release), making it easy to find and work with the appropriate kernel version for any given Ubuntu release.

2.2.1. Launchpad Git URL structure for Ubuntu kernel sources

The Launchpad Git repository URL for Ubuntu Linux kernel sources follow one of the general formats below:

```
https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/<source>/+git/<series>
https://git.launchpad.net/~canonical-kernel/ubuntu/+source/<source>/+git/<series>
```

For example, the source for the generic Jammy Jellyfish (Ubuntu 22.04 LTS) can be found at:

```
https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
```

While the URL for the AWS kernel variant for Noble Numbat (Ubuntu 24.04 LTS) is:

```
https://git.launchpad.net/~canonical-kernel/ubuntu/+source/linux-aws/+git/noble
```

You can get the correct URL by checking the list of Git repositories for the Ubuntu Kernel Repositories team[3] or Canonical Kernel team[4].

2.2.2. Kernel source repository branches

You will find the following branches in each Ubuntu kernel source repository.

- `master`: The source for the Ubuntu distro kernel.
- `master-next`: Contains the commits that will be merged into the `master` branch for the next stable release update (SRU) for the series.

2.2.3. Protocols for accessing kernel sources

Protocol	Authentication needed?	Use case	Command sample
Git protocol	No	Public repositories, require read-only access	`git clone git://<kernel source URL>`
SSH protocol	Yes (SSH key)	Private repositories, require write access	`git clone git+ssh://<kernel source URL>`
HTTPS protocol	Yes (if private)	Public and private repositories,for easy access	`git clone https://<kernel source URL>`

[3] https://code.launchpad.net/~ubuntu-kernel/+git
[4] https://code.launchpad.net/~canonical-kernel/+git

2.3. How to contribute

We believe that everyone has something valuable to contribute, whether you're a coder, a writer, or a tester. Here's how and why you can get involved:

- **Why join us?** Work with like-minded people, develop your skills, connect with diverse professionals, and make a difference.
- **What do you get?** Personal growth, recognition for your contributions, early access to new features, and the joy of seeing your work appreciated.
- **Start early, start easy**: Dive into code contributions, improve documentation, or be among the first testers. Your presence matters, regardless of experience or the size of your contribution.

The guidelines below will help keep your contributions effective and meaningful.

2.3.1. Code of conduct

When contributing, you must abide by the Ubuntu Code of Conduct[5].

2.3.2. Licence and copyright

By default, all contributions to ACME are made under the AGPLv3 licence. See the licence[6] in the ACME GitHub repository for details.

All contributors must sign the Canonical contributor licence agreement[7], which grants Canonical permission to use the contributions. The author of a change remains the copyright owner of their code (no copyright assignment occurs).

2.3.3. Releases and versions

ACME uses semantic versioning[8]; major releases occur once or twice a year.

The release notes can be found TODO: here[9].

2.3.4. Environment setup

To work on the project, you need the following prerequisites:

- TODO: Prerequisite 1[10]
- TODO: Prerequisite 2[11]

To install and configure these tools:

```
TODO: prerequisite command 1
TODO: prerequisite command 2
```

[5] https://ubuntu.com/community/ethos/code-of-conduct
[6] https://github.com/canonical/ACME/blob/main/COPYING
[7] https://ubuntu.com/legal/contributors
[8] https://semver.org/
[9] https://example.com
[10] http://example.com
[11] http://example.com

2.3.5. Submissions

If you want to address an issue or a bug in ACME, notify in advance the people involved to avoid confusion; also, reference the issue or bug number when you submit the changes.

- Fork our GitHub repository[12] and add the changes to your fork, properly structuring your commits, providing detailed commit messages, and signing your commits.
- Make sure the updated project builds and runs without warnings or errors; this includes linting, documentation, code, and tests.
- Submit the changes as a pull request (PR)[13].

Your changes will be reviewed in due time; if approved, they will eventually be merged.

Describing pull requests

To be properly considered, reviewed, and merged, your pull request must provide the following details:

- **Title**: Summarise the change in a short, descriptive title.
- **Description**: Explain the problem that your pull request solves. Mention any new features, bug fixes, or refactoring.
- **Relevant issues**: Reference any related issues, pull requests, and repositories[14].
- **Testing**: Explain whether new or updated tests are included.
- **Reversibility**: If you propose decisions that may be costly to reverse, list the reasons and suggest steps to reverse the changes if necessary.

Commit structure and messages

Use separate commits for each logical change, and for changes to different components. Prefix your commit messages with the names of components they affect, using the code tree structure. For example, start a commit that updates the ACME service with `ACME/service:`.

Use conventional commits[15] to ensure consistency across the project:

```
Ensure correct permissions and ownership for the content mounts

* Work around an ACME issue regarding empty dirs: https://github.com/canonical/ACME/
issues/12345

* Ensure the source directory is owned by the user running a container.

Links:
- ...
- ...
```

[12] https://github.com/canonical/ACME
[13] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requ
creating-a-pull-request-from-a-fork
[14] https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/
autolinked-references-and-urls
[15] https://www.conventionalcommits.org/

Such structure makes it easier to review contributions and simplifies porting fixes to other branches.

Signing commits

To improve contribution tracking, we use the developer certificate of origin (DCO 1.1[16]) and require a “sign-off” for any changes going into each branch.

The sign-off is a simple line at the end of the commit message certifying that you wrote it or have the right to commit it as an open-source contribution.

To sign off on a commit, use the `--signoff` option in `git commit`.

2.3.6. Code

Formatting and linting

ACME relies on these formatting and linting tools:

- TODO: Tool 1[17]
- TODO: Tool 2[18]

To configure and run them:

```
TODO: lint command 1
TODO: lint command 2
```

Structure

- **Check linked code elements**: Ensure coupled code elements, files, and directories are adjacent. For instance, store test data close to the corresponding test code.
- **Group variable declaration and initialisation**: Declare and initialise variables together to improve code organisation and readability.
- **Split large expressions**: Break down large expressions into smaller self-explanatory parts. Use multiple variables where appropriate to make the code more understandable and choose names that reflect their purpose.
- **Use blank lines for logical separation**: Insert a blank line between two logically separate sections of code to improve its structure and readability.
- **Avoid nested conditions**: Avoid nesting conditions to improve readability and maintainability.
- **Remove dead code and redundant comments**: Drop unused or obsolete code and comments to promote a cleaner code base and reduce confusion.
- **Normalise symmetries**: Treat identical operations consistently, using a uniform approach to improve consistency and readability.

[16] https://developercertificate.org/
[17] http://example.com
[18] http://example.com

2.3.7. Tests

All code contributions must include tests.

To run the tests locally before submitting your changes:

```
TODO: test command 1
TODO: test command 2
```

2.3.8. Documentation

ACME's documentation is stored in the `DOCDIR` directory of the repository. It is based on the Canonical starter pack[19] and hosted on Read the Docs[20].

For general guidance, refer to the starter pack guide[21].

For syntax help and guidelines, refer to the Canonical style guides[22].

In structuring, the documentation employs the Diátaxis[23] approach.

To run the documentation locally before submitting your changes:

```
make run
```

Automatic checks

GitHub runs automatic checks on the documentation to verify spelling, validate links, and suggest inclusive language.

You can (and should) run the same checks locally:

```
make spelling
make linkcheck
make woke
```

[19] https://canonical-starter-pack.readthedocs-hosted.com/latest/
[20] https://about.readthedocs.com/
[21] https://canonical-starter-pack.readthedocs-hosted.com/latest/
[22] https://canonical-documentation-with-sphinx-and-readthedocscom.readthedocs-hosted.com/#style-guides
[23] https://diataxis.fr/