The Linux kernel

Introduction

Edit this on GitHub

The Raspberry Pi kernel is hosted on GitHub; updates lag behind the upstream Linux kernel. The upstream kernel updates continuously, whereas Raspberry Pi integrates **long-term releases** of the Linux kernel into the Raspberry Pi kernel. We generate a `next` branch in raspberrypi/firmware for each long-term Linux kernel release. After extensive testing and discussion, we merge each `next` branch into the main branch of our repository.

Update

The usual Raspberry Pi OS update process automatically updates your kernel to the latest stable release. If you want to try the latest unstable test kernel, you can manually update.

Build the kernel

Edit this on GitHub

The default compilers and linkers distributed with an OS are configured to build executables to run on that OS. **Native builds** use these default compilers and linkers. **Cross-compilation** is the process of building code for a target other than the one running the build process.

Cross-compilation of the Raspberry Pi kernel allows you to build a 64-bit kernel from a 32-bit OS, and vice versa. Alternatively, you can cross-compile a 32-bit or 64-bit Raspberry Pi kernel from a device other than a Raspberry Pi.

The instructions below are divided into native builds and cross-compilation. Choose the section appropriate for your situation; although the two processes share many steps, there are also some important differences.

Download kernel source

Before you can build for any target, you need the kernel source. To get the kernel source, you need Git. Begin by installing Git on your device, if you don't already have it:

```
$ sudo apt install git
```

Next, download the source code for the latest Raspberry Pi kernel:

```
$ git clone --depth=1 https://github.com/raspberrypi/linux
```

This can take several minutes.

TIP

The `git clone` command above downloads the current active branch, which we build Raspberry Pi OS images from, without any history. Omit `--depth=1` to download the entire repository, including the full history of all branches. This takes much longer and occupies much more storage.

To download a different branch with no history, add the `--branch` option to the command above, replacing `<branch>` with the name of the branch you wish to download:

```
$ git clone --depth=1 --branch <branch> https://github.com/raspberrypi/linux
```

For a full list of available branches, see the the Raspberry Pi kernel repository.

Now that you have the kernel source, build a fresh kernel natively or via cross-compilation.

Natively build a kernel

This guide assumes that your Raspberry Pi runs the latest version of Raspberry Pi OS.

First, install the build dependencies:

```
$ sudo apt install bc bison flex libssl-dev make
```

Build configuration

This section describes how to apply the default configuration when you build a kernel. You can also configure your kernel in the following ways:

- enable and disable kernel features
- apply patches from another source

To prepare the default configuration, run the appropriate commands from the table below for your Raspberry Pi model.

Architecture	Model	Command

64-bit	Raspberry Pi 3	`$ cd linux`
		`$ KERNEL=kernel8`
		`$ make bcm2711_defconfig`

	Raspberry Pi Compute Module 3	
	Raspberry Pi 3+	
	Raspberry Pi Compute Module	

Architecture	Model	Command
	3+	
	Raspberry Pi Zero 2 W	
	Raspberry Pi 4	
	Raspberry Pi 400	
	Raspberry Pi Compute Module 4	
	Raspberry Pi Compute Module 4S	

	Raspberry Pi 5	`$ cd linux`
		`$ KERNEL=kernel_2712`
		`$ make bcm2712_defconfig`

32-bit	Raspberry Pi 1	`$ cd linux`
		`$ KERNEL=kernel`
		`$ make bcmrpi_defconfig`

	Raspberry Pi Compute Module 1	
	Raspberry Pi Zero	
	Raspberry Pi Zero W	

	Raspberry Pi 2	`$ cd linux`
		`$ KERNEL=kernel7`
		`$ make bcm2709_defconfig`

	Raspberry Pi 3	
	Raspberry Pi Compute Module 3	
	Raspberry Pi 3+	
	Raspberry Pi Compute Module 3+	
	Raspberry Pi Zero 2 W	

	Raspberry Pi 4	`$ cd linux`
		`$ KERNEL=kernel7l`
		`$ make bcm2711_defconfig`

	Raspberry Pi 400	
	Raspberry Pi Compute Module 4	

Architecture	Model	Command
	Raspberry Pi Compute Module 4S	

NOTE

The 32-bit distribution of Raspberry Pi OS on a Raspberry Pi 4B, 5, 400, Compute Module 4, or Compute Module 4S uses a 32-bit userland, but a *64-bit kernel*. To build a 32-bit kernel, set `ARCH=arm`. To boot a 32-bit kernel, set `arm_64bit=0` in `config.txt`.

Customise the kernel version using `LOCALVERSION`

To prevent the kernel from overwriting existing modules in `/lib/modules` and to clarify that you run your own kernel in `uname` output, adjust `LOCALVERSION`.

To adjust `LOCALVERSION`, change the following line in `.config`:

```
CONFIG_LOCALVERSION="-v7l-MY_CUSTOM_KERNEL"
```

TIP

You can also change this setting graphically with `menuconfig` at **General setup** > **Local version - append to kernel release**. For more information about `menuconfig`, see the kernel configuration instructions.

Build

Next, build the kernel. This step can take a long time, depending on your Raspberry Pi model.

- Run the following commands to build a 64-bit kernel:

```
$ make -j6 Image.gz modules dtbs
```

- Run the following command to build a 32-bit kernel:

```
$ make -j6 zImage modules dtbs
```

TIP

On multi-core Raspberry Pi models, the `make -j<n>` option distributes work between cores. This can speed up compilation significantly. Run `nproc` to see how many processors you have; we recommend passing a number 1.5x your number of processors.

Install the kernel

Next, install the kernel modules onto the boot media:

```
$ sudo make -j6 modules_install
```

Then, install the kernel and Device Tree blobs into the boot partition, backing up your original kernel.

TIP

> If you don't want to install the freshly-compiled kernel onto the Raspberry Pi where you run this command, copy the compiled kernel to the boot partition of a separate boot media instead of `/boot/firmware/`.

To install the 64-bit kernel:

- Run the following commands to create a backup image of the current kernel, install the fresh kernel image, overlays, README, and unmount the partitions:

```
$ sudo cp /boot/firmware/$KERNEL.img /boot/firmware/$KERNEL-backup.img
$ sudo cp arch/arm64/boot/Image.gz /boot/firmware/$KERNEL.img
$ sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/firmware/
$ sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/firmware/overlays/
$ sudo cp arch/arm64/boot/dts/overlays/README /boot/firmware/overlays/
```

To install the 32-bit kernel:

1. Create a backup of your current kernel and install the fresh kernel image:

```
$ sudo cp /boot/firmware/$KERNEL.img /boot/firmware/$KERNEL-backup.img
$ sudo cp arch/arm/boot/zImage /boot/firmware/$KERNEL.img
```

2. Depending on your kernel version, run the following command:

    - For kernels up to version 6.4:

    ```
    $ sudo cp arch/arm/boot/dts/*.dtb /boot/firmware/
    ```

    - For kernels version 6.5 and above:

    ```
    $ sudo cp arch/arm/boot/dts/broadcom/*.dtb /boot/firmware/
    ```

3. Finally, copy over the overlays and README:

```
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/firmware/overlays/
$ sudo cp arch/arm/boot/dts/overlays/README /boot/firmware/overlays/
```

Finally, run the following command to reboot your Raspberry Pi and run your freshly-compiled kernel:

```
$ sudo reboot
```

TIP

> Alternatively, copy the kernel with a different filename (e.g. `kernel-myconfig.img`) instead of overwriting the `kernel.img` file. Then, edit `config.txt` in the boot partition to select your kernel:
>
> ```
> kernel=kernel-myconfig.img
> ```

Combine this approach with a custom `LOCALVERSION` to keep your custom kernel separate from the stock kernel image managed by the system. With this arrangement, you can quickly revert to a stock kernel in the event that your kernel cannot boot.

Cross-compile the kernel

First, you will need a suitable Linux cross-compilation host. We tend to use Ubuntu; since Raspberry Pi OS is also a Debian distribution, compilation commands are similar.

Install required dependencies and toolchain

To build the sources for cross-compilation, install the required dependencies onto your device. Run the following command to install most dependencies:

```
$ sudo apt install bc bison flex libssl-dev make libc6-dev libncurses5-dev
```

Then, install the proper toolchain for the kernel architecture you wish to build:

- To install the 64-bit toolchain to build a 64-bit kernel, run the following command:

  ```
  $ sudo apt install crossbuild-essential-arm64
  ```

- To install the 32-bit toolchain to build a 32-bit kernel, run the following command:

  ```
  $ sudo apt install crossbuild-essential-armhf
  ```

Build configuration

This section describes how to apply the default configuration when you build a kernel. You can also configure your kernel in the following ways:

- enable and disable kernel features
- apply patches from another source

Enter the following commands to build the sources and Device Tree files:

Target Architecture	Target Model	Command

64-bit	Raspberry Pi 3	`$ cd linux`
		`$ KERNEL=kernel8`
		`$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig`

	Raspberry Pi Compute Module 3	
	Raspberry Pi 3+	
	Raspberry Pi Compute Module 3+	

Target Architecture	Target Model	Command
	Raspberry Pi Zero 2 W	
	Raspberry Pi 4	
	Raspberry Pi 400	
	Raspberry Pi Compute Module 4	
	Raspberry Pi Compute Module 4S	

	Raspberry Pi 5	`$ cd linux`
		`$ KERNEL=kernel_2712`
		`$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2712_defconfig`

32-bit	Raspberry Pi 1	`$ cd linux`
		`$ KERNEL=kernel`
		`$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcmrpi_defconfig`

	Raspberry Pi Compute Module 1	
	Raspberry Pi Zero	
	Raspberry Pi Zero W	

	Raspberry Pi 2	`$ cd linux`
		`$ KERNEL=kernel7`
		`$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig`

	Raspberry Pi 3	
	Raspberry Pi Compute Module 3	
	Raspberry Pi 3+	
	Raspberry Pi Compute Module 3+	
	Raspberry Pi Zero 2 W	

	Raspberry Pi 4	`$ cd linux`
		`$ KERNEL=kernel7l`
		`$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig`

	Raspberry Pi 400	
	Raspberry Pi Compute Module 4	
	Raspberry Pi Compute Module	

Target Architecture	Target Model	Command
	4S	

Customise the kernel version using `LOCALVERSION`

To prevent the kernel from overwriting existing modules in `/lib/modules` and to clarify that you run your own kernel in `uname` output, adjust `LOCALVERSION`.

To adjust `LOCALVERSION`, change the following line in `.config`:

```
CONFIG_LOCALVERSION="-v7l-MY_CUSTOM_KERNEL"
```

TIP

> You can also change this setting graphically with `menuconfig` at **General setup** > **Local version - append to kernel release**. For more information about `menuconfig`, see the kernel configuration instructions.

Build

- Run the following command to build a 64-bit kernel:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
```

- Run the following command to build a 32-bit kernel:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs
```

Install the kernel

Having built the kernel, you need to copy it onto your Raspberry Pi boot media (likely an SD card or SSD) and install the modules.

Find your boot media

First, run `lsblk`. Then, connect your boot media. Run `lsblk` again; the new device represents your boot media. You should see output similar to the following:

```
sdb
   sdb1
   sdb2
```

If `sdb` represents your boot media, `sdb1` represents the the `FAT32`-formatted **boot partition** and `sdb2` represents the (likely `ext4`-formatted) **root partition**.

First, mount these partitions as `mnt/boot` and `mnt/root`, adjusting the partition letter to match the location of your boot media:

```
$ mkdir mnt
$ mkdir mnt/boot
$ mkdir mnt/root
$ sudo mount /dev/sdb1 mnt/boot
$ sudo mount /dev/sdb2 mnt/root
```

Install

Next, install the kernel modules onto the boot media:

- For 64-bit kernels:

```
$ sudo env PATH=$PATH make -j12 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu
- INSTALL_MOD_PATH=mnt/root modules_install
```

- For 32-bit kernels:

```
$ sudo env PATH=$PATH make -j12 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf
- INSTALL_MOD_PATH=mnt/root modules_install
```

TIP

On multi-core devices, the `make -j<n>` option distributes work between cores. This can speed up compilation significantly. Run `nproc` to see how many processors you have; we recommend passing a number 1.5x your number of processors.

Next, install the kernel and Device Tree blobs into the boot partition, backing up your original kernel.

To install the 64-bit kernel:

- Run the following commands to create a backup image of the current kernel, install the fresh kernel image, overlays, README, and unmount the partitions:

```
$ sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
$ sudo cp arch/arm64/boot/Image mnt/boot/$KERNEL.img
$ sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/boot/
$ sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/boot/overlays/
$ sudo cp arch/arm64/boot/dts/overlays/README mnt/boot/overlays/
$ sudo umount mnt/boot
$ sudo umount mnt/root
```

To install the 32-bit kernel:

1. Run the following commands to create a backup image of the current kernel and install the fresh kernel image:

```
$ sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
$ sudo cp arch/arm/boot/zImage mnt/boot/$KERNEL.img
```

2. Depending on your kernel version, run the following command to install Device Tree blobs:

    - For kernels up to version 6.4:

```
$ sudo cp arch/arm/boot/dts/*.dtb mnt/boot/
```

- For kernels version 6.5 and above:

```
$ sudo cp arch/arm/boot/dts/broadcom/*.dtb mnt/boot/
```

3. Finally, install the overlays and README, and unmount the partitions:

```
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/boot/overlays/
$ sudo cp arch/arm/boot/dts/overlays/README mnt/boot/overlays/
$ sudo umount mnt/boot
$ sudo umount mnt/root
```

Finally, connect the boot media to your Raspberry Pi and connect it to power to run your freshly-compiled kernel.

TIP

Alternatively, copy the kernel with a different filename (e.g. `kernel-myconfig.img`) instead of overwriting the `kernel.img` file. Then, edit `config.txt` in the boot partition to select your kernel:

```
kernel=kernel-myconfig.img
```

Combine this approach with a custom `LOCALVERSION` to keep your custom kernel separate from the stock kernel image managed by the system. With this arrangement, you can quickly revert to a stock kernel in the event that your kernel cannot boot.

Configure the kernel

Edit this on GitHub

The Linux kernel is highly configurable. Advanced users may wish to modify the default configuration to customise it to their needs, such as enabling a new or experimental network protocol, or enabling support for new hardware.

Configuration is most commonly done through the `make menuconfig` interface. Alternatively, you can modify your `.config` file manually, but this can be more difficult.

Prepare to configure

The `menuconfig` tool requires the `ncurses` development headers to compile properly. To install these headers, run the following command:

```
$ sudo apt install libncurses5-dev
```

Next, download your kernel sources. In particular, ensure you have installed the default native configuration or default cross-compilation configuration.

`menuconfig`

Once you've got everything set up, you can compile and run the `menuconfig` utility as follows:

```
$ make menuconfig
```

To cross-compile a 64-bit kernel:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

To cross-compile a 32-bit kernel:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

To navigate the `menuconfig` utility, use your keyboard:

- to navigate directionally, use the **arrow keys**
- to enter a submenu (indicated by `--->`), press the **Enter** key
- to go up a level or exit, press **Escape** twice
- to toggle the state of a binary option, press the **space bar**
- to select the state of a multiple choice option, press **Enter** to open a submenu, the **arrow keys** to navigate the submenu, and press **Enter** again to select a state
- to get help with an option or menu, press the **H** key

After a brief compilation, `menuconfig` presents a list of submenus containing all the options you can configure. There are many options, so take your time to read through them. Resist the temptation to enable or disable a lot of things on your first attempt; it's relatively easy to break your configuration, so start small and get comfortable with the configuration and build process.

Save your changes

Once you're done making changes, press **Escape** until you're prompted to save your new configuration. By default, this saves to the `.config` file. You can save and load configurations by copying this file.

After customising, you should now be ready to build the kernel.

Patch the kernel

Edit this on GitHub

When building your custom kernel, you may wish to apply patches or collections of patches (patchsets) to the Linux kernel.

Hardware makers sometimes provide patchsets as a temporary measure to support new hardware before the patches make it into the Linux kernel and the Raspberry Pi kernel. However, patchsets for other purposes exist, for instance to enable a fully pre-emptible kernel for real-time usage.

Identify your kernel version

To check the kernel version currently running on your device, run the following command:

```
$ uname -r
```

Always check your version of the kernel before applying patches. In a kernel source directory, run the following command to see the kernel version:

```
$ head Makefile -n 4
```

You should see output similar to the following:

```
# SPDX-License-Identifier: GPL-2.0
VERSION = 6
PATCHLEVEL = 1
SUBLEVEL = 38
```

In this instance, the sources are for a 6.1.38 kernel.

Apply patches

The application of patches depends on the format used to distribute the patch.

Developers distribute most patches as a single file. Use the `patch` utility to apply these patches. The following commands download, uncompress, and patch our example kernel version with real-time kernel patches:

```
$ wget https://www.kernel.org/pub/linux/kernel/projects/rt/6.1/patch-6.1.38-rt1
3-rc1.patch.gz
$ gunzip patch-6.1.38-rt13-rc1.patch.gz
$ cat patch-6.1.38-rt13-rc1.patch | patch -p1
```

Some developers distribute patches in the **mailbox format**, a folder containing multiple patch files. Use Git to apply these patches.

NOTE

Before using Git to apply mailbox patches, configure your local Git installation with a name and email:

```
$ git config --global user.name "your name"
$ git config --global user.email "your email"
```

To apply the mailbox formatted patches with Git, run the following command:

```
$ git am -3 /path/to/patches/*
```

Always follow the instructions provided by the patch distributor. For instance, some patchsets require patching against a specific commit.

Kernel headers

Edit this on GitHub

To compile a kernel module, you need the Linux kernel headers. These provide the function and structure definitions required to compile code that interfaces with the kernel.

If you cloned the entire kernel from GitHub, the headers are already included in the source tree. If you don't need all the extra files, you can instead install only the kernel headers with `apt`.

TIP

> When a new kernel is released, you need the headers that match that kernel version. It can take several weeks to update the `apt` package to reflect the latest kernel version. For the latest header versions, clone the kernel.

If you use a 64-bit version of Raspberry Pi OS, run the following command to install the kernel headers:

```
$ sudo apt install linux-headers-rpi-v8
```

If you use a 32-bit version of Raspberry Pi OS, run the following command to install the kernel headers:

```
$ sudo apt install linux-headers-rpi-{v6,v7,v7l}
```

NOTE

> Installation can take several minutes. There is no progress indicator.

Contribute

Edit this on GitHub

There are many reasons you may want to put something into the kernel:

- You've written some Raspberry Pi-specific code that you want everyone to benefit from
- You've written a generic Linux kernel driver for a device and want everyone to use it
- You've fixed a generic kernel bug
- You've fixed a Raspberry Pi-specific kernel bug

For Raspberry Pi-specific changes or bug fixes, submit a pull request to the Raspberry Pi kernel. For general Linux kernel changes (i.e. a new driver), submit a pull request to the upstream Linux kernel first. Once the Linux kernel accepts your change, we'll receive it.

Contribute to the Raspberry Pi Kernel

First, fork the Raspberry Pi kernel repository and clone it to your development device. You can then make your changes, test them, and commit them into your fork.

Then, submit a pull request containing your changes to the Raspberry Pi kernel repository. Raspberry Pi engineers will review your contribution and suggest improvements. Once approved, we'll merge in your changes, and they'll eventually make their way to the stable release of the Raspberry Pi kernel.

Contribute to the Linux kernel

First, clone the Linux kernel tree to your development device. You can then make your changes, test them, and commit them into your local tree.

Once your change is ready you can submit it to the Linux kernel community. Linux kernel development happens on mailing lists, rather than on GitHub. In order for your change to become part of Linux, please email it to the community as a patch. Please follow Submitting patches: the essential guide to getting your code into the kernel and Linux kernel coding style in the Linux kernel documentation. Linux kernel contributors will review your contribution and suggest improvements. Once approved, they'll merge in your changes. Eventually, they'll make their way into a long-term release of the Linux kernel. Once we've tested that long-term release for compatibility with the Raspberry Pi kernel, your changes will make their way into a stable release of the Raspberry Pi kernel.