The Linux kernel

Kernel

Edit this on GitHub

The Raspberry Pi kernel is stored in GitHub and can be viewed at github.com/raspberrypi/linux; it follows behind the main Linux kernel. The main Linux kernel is continuously updating; we take long-term releases of the kernel, which are mentioned on the front page, and integrate the changes into the Raspberry Pi kernel. We then create a 'next' branch which contains an unstable port of the kernel; after extensive testing and discussion, we push this to the main branch.

Updating your Kernel

If you use the standard Raspberry Pi OS update and upgrade process, this will automatically update the kernel to the latest stable version. This is the recommended procedure. However, in certain circumstances, you may wish to update to the latest 'bleeding edge' or test kernel. You should only do this if recommended to do so by a Raspberry Pi engineer, or if there is a specific feature only available in this latest software.

Getting your Code into the Kernel

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

Initially, you should fork the Linux repository and clone that on your build system; this can be either on the Raspberry Pi or on a Linux machine you're using for cross-compiling. You can then make your changes, test them, and commit them into your fork.

Next, depending upon whether the code is Raspberry Pi-specific or not:

- For Raspberry Pi-specific changes or bug fixes, submit a pull request to the kernel.

- For general Linux kernel changes (i.e. a new driver), these need to be submitted upstream first. Once they've been submitted upstream and accepted, submit the pull request and we'll receive it.

Building the Kernel

Edit this on GitHub

The default compilers and linkers that come with an OS are configured to build executables to run on that OS - they are native tools - but that doesn't have to be the case. A cross-compiler is configured to build code for a target other than the one running the build process, and using it is called cross-compilation.

Cross-compilation of the Raspberry Pi kernel is useful for two reasons:

- it allows a 64-bit kernel to be built using a 32-bit OS, and vice versa, and
- even a modest laptop can cross-compile a Raspberry Pi kernel significantly faster than the Raspberry Pi itself.

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

Building the Kernel Locally

On a Raspberry Pi, first install the latest version of Raspberry Pi OS. Then boot your Raspberry Pi, log in, and ensure you're connected to the internet to give you access to the sources.

First install Git and the build dependencies:

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

Next get the sources, which will take some time:

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

Choosing Sources

The `git clone` command above will download the current active branch (the one we are building Raspberry Pi OS images from) without any history. Omitting the `--depth=1` will download the entire repository, including the full history of all branches, but this takes much longer and occupies much more storage.

To download a different branch (again with no history), use the `--branch` option:

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

where `<branch>` is the name of the branch that you wish to download.

Refer to the original GitHub repository for information about the available branches.

Kernel Configuration

Configure the kernel; as well as the default configuration, you may wish to configure your kernel in more detail or apply patches from another source, to add or remove required functionality.

Apply the Default Configuration

First, prepare the default configuration by running the following commands, depending on your Raspberry Pi model:

For Raspberry Pi 1, Zero and Zero W, and Raspberry Pi Compute Module 1 default (32-bit only) build configuration

```
cd linux
KERNEL=kernel
make bcmrpi_defconfig
```

For Raspberry Pi 2, 3, 3+ and Zero 2 W, and Raspberry Pi Compute Modules 3 and 3+ default 32-bit build configuration

```
cd linux
KERNEL=kernel7
make bcm2709_defconfig
```

For Raspberry Pi 4 and 400, and Raspberry Pi Compute Module 4 default 32-bit build configuration

```
cd linux
KERNEL=kernel7l
make bcm2711_defconfig
```

For Raspberry Pi 3, 3+, 4, 400 and Zero 2 W, and Raspberry Pi Compute Modules 3, 3+ and 4 default 64-bit build configuration

```
cd linux
KERNEL=kernel8
make bcm2711_defconfig
```

Customising the Kernel Version Using `LOCALVERSION`

In addition to your kernel configuration changes, you may wish to adjust the `LOCALVERSION` to ensure your new kernel does not receive the same version string as the upstream kernel. This both clarifies you are running your own kernel in the output of `uname` and ensures existing modules in `/lib/modules` are not overwritten.

To do so, change the following line in `.config`:

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

You can also change that setting graphically as shown in the kernel configuration instructions. It is located in "General setup" => "Local version - append to kernel release".

Building the Kernel

Build and install the kernel, modules, and Device Tree blobs; this step can take a **long** time depending on the Raspberry Pi model in use. For the 32-bit kernel:

```
make -j4 zImage modules dtbs
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

For the 64-bit kernel:

```
make -j4 Image.gz modules dtbs
sudo make modules_install
sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm64/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm64/boot/Image.gz /boot/$KERNEL.img
```

NOTE

On a Raspberry Pi 2/3/4, the `-j4` flag splits the work between all four cores, speeding up compilation significantly.

If you now reboot, your Raspberry Pi should be running your freshly-compiled kernel!

Cross-Compiling 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, it means many aspects are similar, such as the command lines.

You can either do this using VirtualBox (or VMWare) on Windows, or install it directly onto your computer. For reference, you can follow instructions online at Wikihow.

Install Required Dependencies and Toolchain

To build the sources for cross-compilation, make sure you have the dependencies needed on your machine by executing:

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

If you find you need other things, please submit a pull request to change the documentation.

Install the 32-bit Toolchain for a 32-bit Kernel

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

Install the 64-bit Toolchain for a 64-bit Kernel

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

Get the Kernel Sources

To download the minimal source tree for the current branch, run:

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

See **Choosing sources** above for instructions on how to choose a different branch.

Build sources

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

32-bit Configs

For Raspberry Pi 1, Zero and Zero W, and Raspberry Pi Compute Module 1:

```
cd linux
KERNEL=kernel
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcmrpi_defconfig
```

For Raspberry Pi 2, 3, 3+ and Zero 2 W, and Raspberry Pi Compute Modules 3 and 3+:

```
cd linux
KERNEL=kernel7
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

For Raspberry Pi 4 and 400, and Raspberry Pi Compute Module 4:

```
cd linux
KERNEL=kernel7l
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig
```

64-bit Configs

For Raspberry Pi 3, 3+, 4, 400 and Zero 2 W, and Raspberry Pi Compute Modules 3, 3+ and 4:

```
cd linux
KERNEL=kernel8
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
```

Build with Configs

NOTE

To speed up compilation on multiprocessor systems, and get some improvement on single processor ones, use `-j n`, where n is the number of processors * 1.5. You can use the `nproc` command to see how many processors you have. Alternatively, feel free to experiment and see what works!

For all 32-bit Builds

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

For all 64-bit Builds

NOTE

Note the difference between Image target between 32 and 64-bit.

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

Install Directly onto the SD Card

Having built the kernel, you need to copy it onto your Raspberry Pi and install the modules; this is best done directly using an SD card reader.

First, use `lsblk` before and after plugging in your SD card to identify it. You should end up with something a lot like this:

```
sdb
   sdb1
   sdb2
```

with `sdb1` being the `FAT` filesystem (boot) partition, and `sdb2` being the `ext4` filesystem (root) partition.

Mount these first, adjusting the partition letter as necessary:

```
mkdir mnt
mkdir mnt/fat32
mkdir mnt/ext4
sudo mount /dev/sdb1 mnt/fat32
sudo mount /dev/sdb2 mnt/ext4
```

NOTE

You should adjust the drive letter appropriately for your setup, e.g. if your SD card appears as `/dev/sdc` instead of `/dev/sdb`.

Next, install the kernel modules onto the SD card:

For 32-bit

```
sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MO
D_PATH=mnt/ext4 modules_install
```

For 64-bit

```
sudo env PATH=$PATH make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- INSTALL_MO
D_PATH=mnt/ext4 modules_install
```

Finally, copy the kernel and Device Tree blobs onto the SD card, making sure to back up your old kernel:

For 32-bit

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm/boot/zImage mnt/fat32/$KERNEL.img
sudo cp arch/arm/boot/dts/*.dtb mnt/fat32/
sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

For 64-bit

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/fat32/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

Another option is to copy the kernel into the same place, but with a different filename - for instance, `kernel-myconfig.img` - rather than overwriting the `kernel.img` file. You can then edit the `config.txt` file to select the kernel that the Raspberry Pi will boot:

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

This has the advantage of keeping your custom kernel separate from the stock kernel image managed by the system and any automatic update tools, and allowing you to easily revert to a stock kernel in the event that your kernel cannot boot.

Finally, plug the card into the Raspberry Pi and boot it!

Configuring 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 for new users.

Preparing to Configure

The `menuconfig` tool requires the `ncurses` development headers to compile properly. These can be installed with the following command:

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

You'll also need to download and prepare your kernel sources, as described in the build guide. In particular, ensure you have installed the default configuration.

Using `menuconfig`

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

```
make menuconfig
```

If you're cross-compiling a 32-bit kernel:

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

Or, if you are cross-compiling a 64-bit kernel:

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

The `menuconfig` utility has simple keyboard navigation. After a brief compilation, you'll be presented with a list of submenus containing all the options you can configure; there's a lot, so take your time to read through them and get acquainted.

Use the arrow keys to navigate, the Enter key to enter a submenu (indicated by `--->`), Escape twice to go up a level or exit, and the space bar to cycle the state of an option. Some options have multiple choices, in which case they'll appear as a submenu and the Enter key will select an option. You can press `h` on most entries to get help about that specific option or menu.

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.

Saving your Changes

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

Patching 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.

Patchsets are often provided with newer hardware as a temporary measure, before the patches are applied to the upstream Linux kernel ('mainline') and then propagated down to the Raspberry Pi kernel sources. However, patchsets for other purposes exist, for instance to enable a fully pre-emptible kernel for real-time usage.

Version Identification

It's important to check what version of the kernel you have when downloading and applying patches. In a kernel source directory, running `head Makefile -n 3` will show you the version the sources relate to:

```
VERSION = 3
PATCHLEVEL = 10
SUBLEVEL = 25
```

In this instance, the sources are for a 3.10.25 kernel. You can see what version you're running on your system with the `uname -r` command.

Applying Patches

How you apply patches depends on the format in which the patches are made available. Most patches are a single file, and applied with the `patch` utility. For example, let's download and patch our example kernel version with the real-time kernel patches:

```
 wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.10/older/patch-3.1
0.25-rt23.patch.gz
 gunzip patch-3.10.25-rt23.patch.gz
 cat patch-3.10.25-rt23.patch | patch -p1
```

In our example we simply download the file, uncompress it, and then pass it to the `patch` utility using the `cat` tool and a Unix pipe.

Some patchsets come as mailbox-format patchsets, arranged as a folder of patch files. We can use Git to apply these patches to our kernel, but first we must configure Git to let it know who we are when we make these changes:

```
git config --global user.name "Your name"
git config --global user.email "your email in here"
```

Once we've done this we can apply the patches:

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

If in doubt, consult with the distributor of the patches, who should tell you how to apply them. Some patchsets will require a specific commit to patch against; follow the details provided by the patch distributor.

Kernel Headers

Edit this on GitHub

If you are compiling a kernel module or similar, you will need the Linux Kernel headers. These provide the various function and structure definitions required when compiling code that interfaces with the kernel.

If you have cloned the entire kernel from github, the headers are already included in the source tree. If you don't need all the extra files, it is possible to install only the kernel headers from the Raspberry Pi OS repo.

```
sudo apt install raspberrypi-kernel-headers
```

NOTE

It can take quite a while for this command to complete, as it installs a lot of small files. There is no progress indicator.

When a new kernel release is made, you will need the headers that match that kernel version. It can take several weeks for the repo to be updated to reflect the latest kernel version. If this happens, the best approach is to clone the kernel as described in the Build Section.