# KERNEL BUILDING

There are two main methods for building the kernel. You can build locally on a Raspberry Pi, which will take a long time; or you can cross-compile, which is much quicker, but requires more setup.

## LOCAL BUILDING

On a Raspberry Pi, first install the latest version of Raspbian. Then boot your Pi, plug in Ethernet to give you access to the sources, and log in.

First install Git and the build dependencies:

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

Next get the sources, which will take some time:

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

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:

Run the following commands, depending on your Raspberry Pi version.

### RASPBERRY PI 1, PI 0, PI 0W, AND COMPUTE MODULE DEFAULT BUILD CONFIGURATION

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

### RASPBERRY PI 2, PI 3, AND COMPUTE MODULE 3 DEFAULT BUILD CONFIGURATION

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

Build and install the kernel, modules, and Device Tree blobs; this step takes a **long** time:

```
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
```

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

## CROSS-COMPILING

First, you will need a suitable Linux cross-compilation host. We tend to use Ubuntu; since Raspbian 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 TOOLCHAIN

Use the following command to download the toolchain to the home folder:

```
git clone https://github.com/raspberrypi/tools ~/
```

Updating the $PATH environment variable makes the system aware of file locations needed for cross-compilation. On a 32-bit host system you can update and reload it using:

```
echo PATH=\$PATH:~/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian/bin >> ~/.bashrc
source ~/.bashrc
```

If you are on a 64-bit host system, you should use:

```
echo PATH=\$PATH:~/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian-x64/bin >> ~/.bashrc
source ~/.bashrc
```

### GET SOURCES

To get the sources, refer to the original GitHub repository for the various branches.

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

## BUILD SOURCES

To build the sources for cross-compilation, there may be extra dependencies beyond those you've installed by default with Ubuntu. If you find you need other things, please submit a pull request to change the documentation.

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

For Pi 1, Pi 0, Pi 0 W, or Compute Module:

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

For Pi 2, Pi 3, or Compute Module 3:

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

Then, for both:

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

**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. Alternatively, feel free to experiment and see what works!

## 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 like this:

```
sdb
   sdb1
   sdb2
```

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

If it's a NOOBS card, you should see something like this:

```
sdb
  sdb1
  sdb2
  sdb5
  sdb6
  sdb7
```

with `sdb6` being the FAT (boot) partition, and `sdb7` being the ext4 filesystem (root) partition.

Mount these first, adjusting the partition numbers for NOOBS cards:

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

Next, install the modules:

```
sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_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:

```
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
```

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 Pi will boot into:

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

This has the advantage of keeping your kernel separate from the 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 Pi and boot it!