Configuration

The `raspi-config` Tool

Edit this on GitHub

`raspi-config` is the Raspberry Pi configuration tool originally written by Alex Bradbury. To open the configuration tool, type the following on the command line:

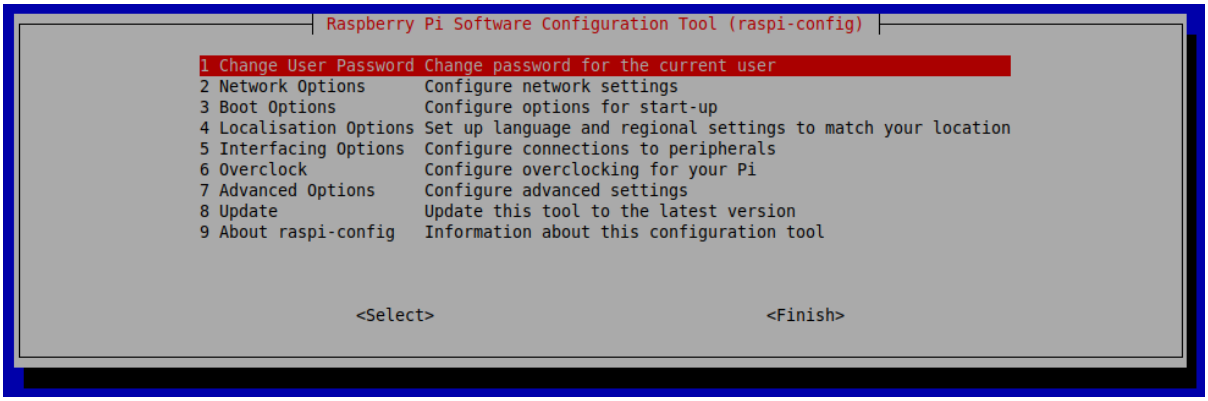
```
sudo raspi-config
```

The `sudo` is required because you will be changing files that you do not own as the `pi` user.

NOTE

> If you are using the Raspberry Pi desktop then you can use the graphical `Raspberry Pi Configuration` application from the `Preferences` menu to configure your Raspberry Pi.

You should then see a blue screen with options in a grey box:



NOTE

> The menu shown may differ slightly.

Use the `up` and `down` arrow keys to move the highlighted selection between the options available. Pressing the `right` arrow key will jump out of the Options menu and take you to the `<Select>` and `<Finish>` buttons. Pressing `left` will take you back to the options. Alternatively, you can use the `Tab` key to switch between these.

Generally speaking, `raspi-config` aims to provide the functionality to make the most common configuration changes. This may result in automated edits to `/boot/config.txt` and various standard Linux configuration files. Some options require a reboot to take effect. If you changed any of those, `raspi-config` will ask if you wish to reboot now when you select the `<Finish>` button.

NOTE

> In long lists of option values (like the list of timezone cities), you can also type a letter to skip to that section of the list. For example, entering `L` will skip you to Lisbon, just two options away from London, to save you scrolling all the way through the alphabet.

List of Options

NOTE

> Due to the continual development of the `raspi-config` tool, the list of options below may not be completely up to date. Also please be aware that different models of Raspberry Pi may have different options available.

System Options

The system options submenu allows you to make configuration changes to various parts of the boot, login and networking process, along with some other system level changes.

Wireless LAN

Allows setting of the wireless LAN SSID and passphrase.

Audio

Specify the audio output destination.

Password

You can change the 'default' user password.

NOTE

> Until recently the default user on Raspberry Pi OS was `pi` with the password `raspberry`. The default user is now set on first boot using a configuration wizard.

Hostname

Set the visible name for this Raspberry Pi on a network.

Boot / Auto login

From this submenu you can select whether to boot to console or desktop and whether you need to log in or not. If you select automatic login, you will be logged in as the `pi` user.

Network at Boot

Use this option to wait for a network connection before letting boot proceed.

Splash Screen

Enable or disable the splash screen displayed at boot time

Power LED

If the model of Raspberry Pi permits it, you can change the behaviour of the power LED using this option.

Display Options

Resolution

Define the default HDMI/DVI video resolution to use when the system boots without a TV or monitor being connected. This can have an effect on RealVNC if the VNC option is enabled.

Underscan

Old TV sets had a significant variation in the size of the picture they produced; some had cabinets that overlapped the screen. TV pictures were therefore given a black border so that none of the picture was lost; this is called overscan. Modern TVs and monitors don't need the border, and the signal doesn't allow for it. If the initial text shown on the screen disappears off the edge, you need to enable overscan to bring the border back.

Any changes will take effect after a reboot. You can have greater control over the settings by editing config.txt.

On some displays, particularly monitors, disabling overscan will make the picture fill the whole screen and correct the resolution. For other displays, it may be necessary to leave overscan enabled and adjust its values.

Pixel Doubling

Enable/disable 2x2 pixel mapping.

Composite Video

On the Raspberry Pi 4, enable composite video. On models prior to the Raspberry Pi 4, composite video is enabled by default so this option is not displayed.

Screen Blanking

Enable or disable screen blanking.

Interfacing Options

In this submenu there are the following options to enable/disable: Camera, SSH, VNC, SPI, I2C, Serial, 1-wire, and Remote GPIO.

Camera

Enable/disable the CSI camera interface.

SSH

Enable/disable remote command line access to your Raspberry Pi using SSH.

SSH allows you to remotely access the command line of the Raspberry Pi from another computer. SSH is disabled by default. Read more about using SSH on the SSH documentation page. If connecting your Raspberry Pi directly to a public network, you should not enable SSH unless you have set up secure passwords for all users.

VNC

Enable/disable the RealVNC virtual network computing server.

SPI

Enable/disable SPI interfaces and automatic loading of the SPI kernel module, needed for products such as PiFace.

I2C

Enable/disable I2C interfaces and automatic loading of the I2C kernel module.

Serial

Enable/disable shell and kernel messages on the serial connection.

1-wire

Enable/disable the Dallas 1-wire interface. This is usually used for DS18B20 temperature sensors.

Remote GPIO

Enable or disable remote access to the GPIO pins.

Performance Options

Overclock

On some models it is possible to overclock your Raspberry Pi's CPU using this tool. The overclocking you can achieve will vary; overclocking too high may result in instability. Selecting this option shows the following warning:

Be aware that overclocking may reduce the lifetime of your Raspberry Pi. If overclocking at a certain level causes system instability, try a more modest overclock. Hold down the Shift key during boot to temporarily disable overclocking.

GPU Memory

Change the amount of memory made available to the GPU.

Overlay File System

Enable or disable a read-only filesystem

Fan

Set the behaviour of a GPIO connected fan

Localisation Options

The localisation submenu gives you these options to choose from: keyboard layout, time zone, locale, and wireless LAN country code.

Locale

Select a locale, for example `en_GB.UTF-8 UTF-8`.

Time Zone

Select your local time zone, starting with the region, e.g. Europe, then selecting a city, e.g. London. Type a letter to skip down the list to that point in the alphabet.

Keyboard

This option opens another menu which allows you to select your keyboard layout. It will take a long time to display while it reads all the keyboard types. Changes usually take effect immediately, but may require a reboot.

WLAN Country

This option sets the country code for your wireless network.

Advanced Options

Expand Filesystem

This option will expand your installation to fill the whole SD card, giving you more space to use for files. You will need to reboot the Raspberry Pi to make this available.

WARNING

> There is no confirmation: selecting the option begins the partition expansion immediately.

GL Driver

Enable/disable the experimental GL desktop graphics drivers.

GL (Full KMS)

Enable/disable the experimental OpenGL Full KMS (kernel mode setting) desktop graphics driver.

GL (Fake KMS)

Enable/disable the experimental OpenGL Fake KMS desktop graphics driver.

Legacy

Enable/disable the original legacy non-GL VideoCore desktop graphics driver.

Compositor

Enable/Display the xcompmgr composition manager

Network Interface Names

Enable or disable predictable network interface names.

Network Proxy Settings

Configure the network's proxy settings.

Boot Order

On the Raspberry Pi 4, you can specify whether to boot from USB or network if the SD card isn't inserted. See this page for more information.

Bootloader Version

On the Raspberry Pi 4, you can tell the system to use the very latest boot ROM software, or revert to the factory default if the latest version causes problems.

Update

Update this tool to the latest version.

About raspi-config

Selecting this option shows the following text:

```
This tool provides a straightforward way of doing initial configuration of the
Raspberry Pi.
Although it can be run at any time, some of the options may have difficulties i
f you have heavily customised your installation.
```

Finish

Use this button when you have completed your changes. You will be asked whether you want to reboot or not. When used for the first time, it's best to reboot. There will be a delay in rebooting if you have chosen to resize your SD card.

Configuring Networking

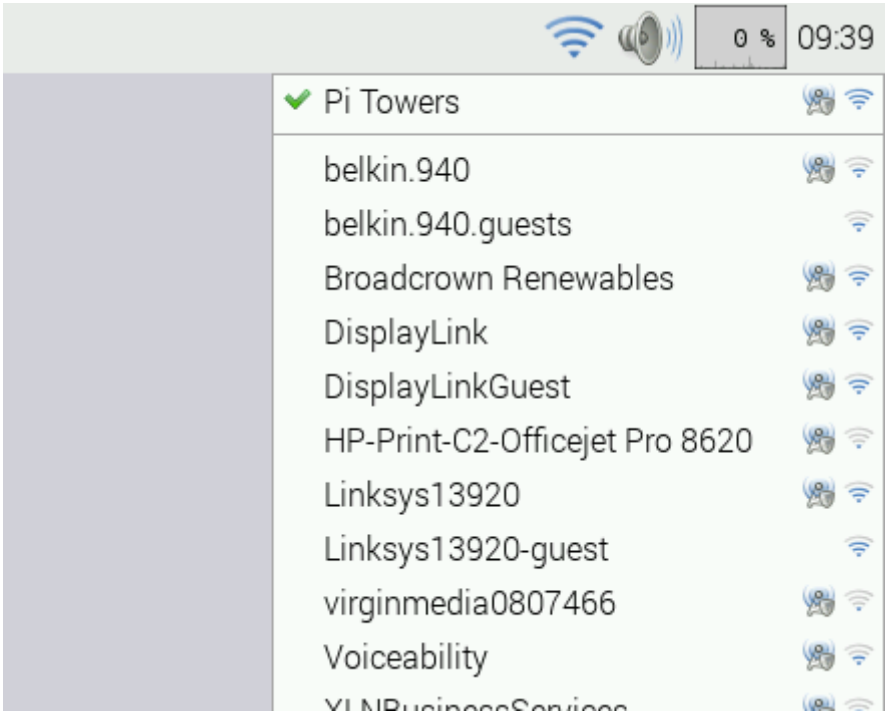
Edit this on GitHub

A GUI is provided for setting up wireless connections in Raspberry Pi OS with desktop. However if you are using Raspberry Pi OS Lite, you can set up wireless networking from the command line.

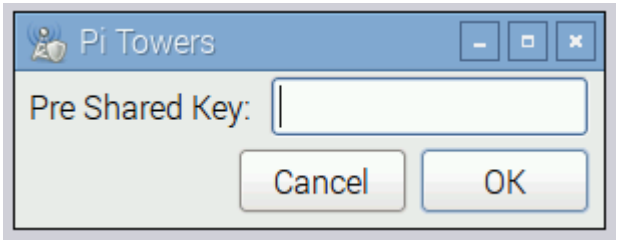
Using the Desktop

Wireless connections can be made via the network icon at the right-hand end of the menu bar. If you are using a Raspberry Pi with built-in wireless connectivity, or if a wireless dongle is plugged in, left-clicking this icon will bring up a list of available wireless networks, as shown below. If no networks are found, it will show the message 'No APs found - scanning...'. Wait a few seconds without closing the menu, and it should find your network.

Note that on Raspberry Pi devices that support the 5GHz band (Pi3B+, Pi4, CM4, Pi400), wireless networking is disabled for regulatory reasons, until the country code has been set. To set the country code, open the `Raspberry Pi Configuration` application from the Preferences Menu, select **Localisation** and set the appropriate code.



The icons on the right show whether a network is secured or not, and give an indication of its signal strength. Click the network that you want to connect to. If it is secured, a dialogue box will prompt you to enter the network key:



Enter the key and click **OK**, then wait a couple of seconds. The network icon will flash briefly to show that a connection is being made. When it is ready, the icon will stop flashing and show the signal strength.

Using the Command Line

This method is suitable if you don't have access to the graphical user interface normally used to set up a wireless LAN on the Raspberry Pi. It is particularly suitable for use with a serial console cable if you don't have access to a screen or wired Ethernet network. Note also that no additional software is required; everything you need is already included on the Raspberry Pi.

Using raspi-config

The quickest way to enable wireless networking is to use the command line `raspi-config` tool.

```
sudo raspi-config
```

Select the **Localisation Options** item from the menu, then the **Change wireless country** option. On a fresh install, for regulatory purposes, you will need to specify the country in which the device is being used. Then set the SSID of the network, and the passphrase for the network. If you do not know the SSID of the network you want to connect to, see the next section on how to list available networks prior to running `raspi-config`.

Note that `raspi-config` does not provide a complete set of options for setting up wireless networking; you may need to refer to the extra sections below for more details if `raspi-config` fails to connect the Raspberry Pi to your requested network.

Getting Wireless LAN Network Details

To scan for wireless networks, use the command `sudo iwlist wlan0 scan`. This will list all available wireless networks, along with other useful information. Look out for:

1. 'ESSID:"testing"' is the name of the wireless network.
2. 'IE: IEEE 802.11i/WPA2 Version 1' is the authentication used. In this case it's WPA2, the newer and more secure wireless standard which replaces WPA. This guide should work for WPA or WPA2, but may not work for WPA2 enterprise. You'll also need the password for the wireless network. For most home routers, this is found on a sticker on the back of the router. The ESSID (ssid) for the examples below is `testing` and the password (psk) is `testingPassword`.

Adding the Network Details to your Raspberry Pi

Open the `wpa-supplicant` configuration file in nano:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

Go to the bottom of the file and add the following:

```
network={
    ssid="testing"
    psk="testingPassword"
}
```

The password can be configured either as the ASCII representation, in quotes as per the example above, or as a pre-encrypted 32 byte hexadecimal number. You can use the `wpa_passphrase` utility to generate an encrypted PSK. This takes the SSID and the password, and generates the encrypted PSK. With the example from above, you can

generate the PSK with `wpa_passphrase "testing"`. Then you will be asked for the password of the wireless network (in this case `testingPassword`). The output is as follows:

```
network={
        ssid="testing"
        #psk="testingPassword"
        psk=131e1e221f6e06e3911a2d11ff2fac9182665c004de85300f9cac208a6a80531
}
```

Note that the plain text version of the code is present, but commented out. You should delete this line from the final `wpa_supplicant` file for extra security.

The `wpa_passphrase` tool requires a password with between 8 and 63 characters. To use a more complex password, you can extract the content of a text file and use it as input for `wpa_passphrase`. Store the password in a text file and input it to `wpa_passphrase` by calling `wpa_passphrase "testing" < file_where_password_is_stored`. For extra security, you should delete the `file_where_password_is_stored` afterwards, so there is no plain text copy of the original password on the system.

To use the `wpa_passphrase`--encrypted PSK, you can either copy and paste the encrypted PSK into the `wpa_supplicant.conf` file, or redirect the tool's output to the configuration file in one of two ways:

- Either change to `root` by executing `sudo su`, then call `wpa_passphrase "testing" >> /etc/wpa_supplicant/wpa_supplicant.conf` and enter the testing password when asked
- Or use `wpa_passphrase "testing" | sudo tee -a /etc/wpa_supplicant/wpa_supplicant.conf > /dev/null` and enter the testing password when asked; the redirection to `/dev/null` prevents `tee` from **also** outputting to the screen (standard output).

If you want to use one of these two options, **make sure you use `>>`, or use `-a` with `tee`** — either will **append** text to an existing file. Using a single chevron `>`, or omitting `-a` when using `tee`, will erase all contents and **then** append the output to the specified file.

Now save the file by pressing `Ctrl+X`, then `Y`, then finally press `Enter`.

Reconfigure the interface with `wpa_cli -i wlan0 reconfigure`.

You can verify whether it has successfully connected using `ifconfig wlan0`. If the `inet addr` field has an address beside it, the Raspberry Pi has connected to the network. If not, check that your password and ESSID are correct.

On the Raspberry Pi 3B+ and Raspberry Pi 4B, you will also need to set the country code, so that the 5GHz networking can choose the correct frequency bands. You can do this using the `raspi-config` application: select the 'Localisation Options' menu, then 'Change Wi-Fi Country'. Alternatively, you can edit the `wpa_supplicant.conf` file and add the

following. (Note: you need to replace 'GB' with the 2 letter ISO code of your country. See Wikipedia for a list of 2 letter ISO 3166-1 country codes.)

```
country=GB
```

Note that with the latest Buster Raspberry Pi OS release, you must ensure that the `wpa_supplicant.conf` file contains the following information at the top:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=<Insert 2 letter ISO 3166-1 country code here>
```

Using Unsecured Networks

If the network you are connecting to does not use a password, the `wpa_supplicant` entry for the network will need to include the correct `key_mgmt` entry. e.g.

```
network={
    ssid="testing"
    key_mgmt=NONE
}
```

WARNING

> You should be careful when using unsecured wireless networks.

Hidden Networks

If you are using a hidden network, an extra option in the `wpa_supplicant file`, `scan_ssid`, may help connection.

```
network={
    ssid="yourHiddenSSID"
    scan_ssid=1
    psk="Your_wireless_network_password"
}
```

You can verify whether it has successfully connected using `ifconfig wlan0`. If the `inet addr` field has an address beside it, the Raspberry Pi has connected to the network. If not, check your password and ESSID are correct.

Adding Multiple Wireless Network Configurations

On recent versions of Raspberry Pi OS, it is possible to set up multiple configurations for wireless networking. For example, you could set up one for home and one for school.

For example

```
network={
    ssid="SchoolNetworkSSID"
    psk="passwordSchool"
    id_str="school"
}

network={
    ssid="HomeNetworkSSID"
    psk="passwordHome"
    id_str="home"
}
```

If you have two networks in range, you can add the priority option to choose between them. The network in range, with the highest priority, will be the one that is connected.

```
network={
    ssid="HomeOneSSID"
    psk="passwordOne"
    priority=1
    id_str="homeOne"
}

network={
    ssid="HomeTwoSSID"
    psk="passwordTwo"
    priority=2
    id_str="homeTwo"
}
```

The DHCP Daemon

The Raspberry Pi uses `dhcpcd` to configure TCP/IP across all of its network interfaces. The `dhcpcd` daemon is intended to be an all-in-one ZeroConf client for UNIX-like systems. This includes assigning each interface an IP address, setting netmasks, and configuring DNS resolution via the Name Service Switch (NSS) facility.

By default, Raspberry Pi OS attempts to automatically configure all network interfaces by DHCP, falling back to automatic private addresses in the range 169.254.0.0/16 if DHCP fails. This is consistent with the behaviour of other Linux variants and of Microsoft Windows.

Static IP Addresses

WARNING

> If allocation of IP addresses is normally handled by a DHCP server on your network,

allocating your Raspberry Pi a static IP address may cause an address conflict which may lead to networking problems.

If you want to allocate a static IP address to your Raspberry Pi, the best way to do so is to reserve an address for it on your router. That way your Raspberry Pi will continue to have its address allocated via DHCP but will receive the same address each time. A "fixed" address can be allocated by your DHCP server associating it with the MAC address of your Raspberry Pi. Management of IP addresses will remain with the DHCP server and this will avoid address conflicts and potential network problems.

However, if you wish to disable automatic configuration for an interface, and instead configure it statically, you can do so in `/etc/dhcpcd.conf`. For example:

```
interface eth0
static ip_address=192.168.0.4/24
static routers=192.168.0.254
static domain_name_servers=192.168.0.254 8.8.8.8
```

Setting up a Headless Raspberry Pi

Edit this on GitHub

If you do not use a monitor or keyboard to run your Raspberry Pi (known as headless), but you still need to do some wireless setup, there is a facility to enable wireless networking and SSH when creating an image.

Once an image is created on an SD card, by inserting it into a card reader on a Linux or Windows machines the boot folder can be accessed. Adding certain files to this folder will activate certain setup features on the first boot of the Raspberry Pi.

IMPORTANT

If you are installing Raspberry Pi OS, and intend to run it headless, you will need to create a new user account. Since you will not be able to create the user account using the first-boot wizard as it requires both a monitor and a keyboard, you **MUST** add a `userconf.txt` file to the boot folder to create a user on first boot or configure the OS with a user account using the Advanced Menu in the Raspberry Pi Imager.

Configuring Networking

You will need to define a `wpa_supplicant.conf` file for your particular wireless network. Put this file onto the boot folder of the SD card. When the Raspberry Pi boots for the first time, it will copy that file into the correct location in the Linux root file system and use those settings to start up wireless networking.

The Raspberry Pi's IP address will not be visible immediately after power on, so this step is crucial to connect to it headlessly. Depending on the OS and editor you are creating this on,

the file could have incorrect newlines or the wrong file extension so make sure you use an editor that accounts for this. Linux expects the line feed (LF) newline character.

WARNING

> After your Raspberry Pi is connected to power, make sure to wait a few (up to 5) minutes for it to boot up and register on the network.

A `wpa_supplicant.conf` file example:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
country=<Insert 2 letter ISO 3166-1 country code here>
update_config=1

network={
 ssid="<Name of your wireless LAN>"
 psk="<Password for your wireless LAN>"
}
```

Where the country code should be set the two letter ISO/IEC alpha2 code for the country in which you are using, e.g.

- GB (United Kingdom)
- FR (France)
- DE (Germany)
- US (United States)
- SE (Sweden)

Here is a more elaborate example that should work for most typical wpa2 personal networks. This template below works for 2.4ghz/5ghz hidden or not networks. The utilization of quotes around the ssid - psk can help avoid any oddities if your network ssid or password has special chars (! @ # $ etc)

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=<Insert 2 letter ISO 3166-1 country code here>

network={
        scan_ssid=1
        ssid="<Name of your wireless LAN>"
        psk="<Password for your wireless LAN>"
        proto=RSN
        key_mgmt=WPA-PSK
        pairwise=CCMP
        auth_alg=OPEN
}
```

NOTE

Some older Raspberry Pi boards and some USB wireless dongles do not support 5GHz networks.

NOTE

With no keyboard or monitor, you will need some way of remotely accessing your headless Raspberry Pi. For headless setup, SSH can be enabled by placing a file named `ssh`, without any extension, onto the boot folder of the SD Card. For more information see the section on setting up an SSH server.

Configuring a User

You will need to add a `userconf.txt` in the boot partition of the SD card; this is the part of the SD card which can be seen when it is mounted in a Windows or MacOS computer.

This file should contain a single line of text, consisting of `username:password` – so your desired username, followed immediately by a colon, followed immediately by an **encrypted** representation of the password you want to use.

To generate the encrypted password, the easiest way is to use OpenSSL on a Raspberry Pi that is already running – open a terminal window and enter:

```
openssl passwd -6
```

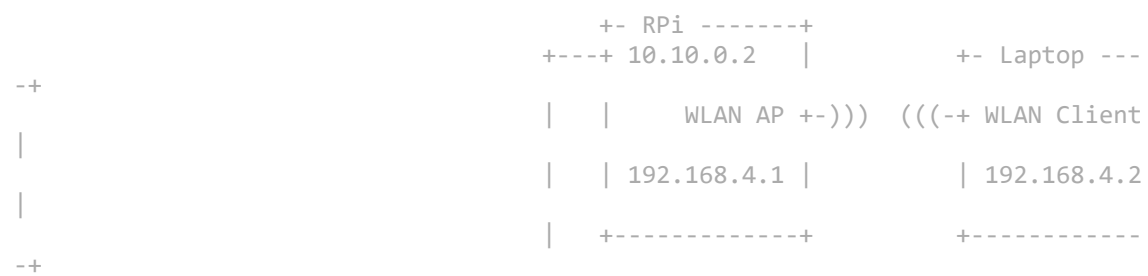
This will prompt you to enter your password, and verify it. It will then produce what looks like a string of random characters, which is actually an encrypted version of the supplied password.

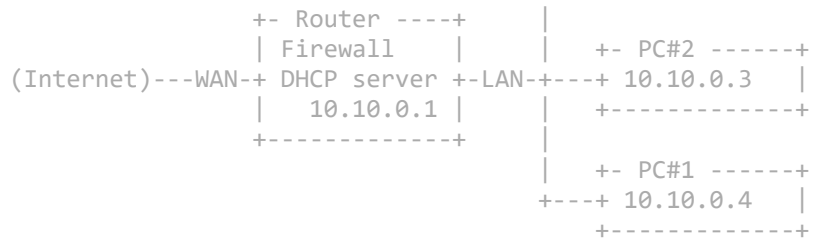
Setting up a Routed Wireless Access Point

Edit this on GitHub

A Raspberry Pi within an Ethernet network can be used as a wireless access point, creating a secondary network. The resulting new wireless network is entirely managed by the Raspberry Pi.

If you wish to extend an existing Ethernet network to wireless clients, consider instead setting up a bridged access point.





A routed wireless access point can be created using the inbuilt wireless features of the Raspberry Pi 4, Raspberry Pi 3 or Raspberry Pi Zero W, or by using a suitable USB wireless dongle that supports access point mode. It is possible that some USB dongles may need slight changes to their settings. If you are having trouble with a USB wireless dongle, please check the forums.

This documentation was tested on a Raspberry Pi 3B running a fresh installation of Raspberry Pi OS Buster.

Before you Begin

- Ensure you have administrative access to your Raspberry Pi. The network setup will be modified as part of the installation: local access, with screen and keyboard connected to your Raspberry Pi, is recommended.
- Connect your Raspberry Pi to the Ethernet network and boot the Raspberry Pi OS.
- Ensure the Raspberry Pi OS on your Raspberry Pi is up-to-date and reboot if packages were installed in the process.
- Take note of the IP configuration of the Ethernet network the Raspberry Pi is connected to:
    - In this document, we assume IP network `10.10.0.0/24` is configured on the Ethernet LAN, and the Raspberry Pi is going to manage IP network `192.168.4.0/24` for wireless clients.
    - Please select another IP network for wireless, e.g. `192.168.10.0/24`, if IP network `192.168.4.0/24` is already in use by your Ethernet LAN.
- Have a wireless client (laptop, smartphone, …) ready to test your new access point.

Install AP and Management Software

In order to work as an access point, the Raspberry Pi needs to have the `hostapd` access point software package installed:

```
sudo apt install hostapd
```

Enable the wireless access point service and set it to start when your Raspberry Pi boots:

```
sudo systemctl unmask hostapd
sudo systemctl enable hostapd
```

In order to provide network management services (DNS, DHCP) to wireless clients, the Raspberry Pi needs to have the `dnsmasq` software package installed:

```
sudo apt install dnsmasq
```

Finally, install `netfilter-persistent` and its plugin `iptables-persistent`. This utility helps by saving firewall rules and restoring them when the Raspberry Pi boots:

```
sudo DEBIAN_FRONTEND=noninteractive apt install -y netfilter-persistent iptable
s-persistent
```

Software installation is complete. We will configure the software packages later on.

Set up the Network Router

The Raspberry Pi will run and manage a standalone wireless network. It will also route between the wireless and Ethernet networks, providing internet access to wireless clients. If you prefer, you can choose to skip the routing by skipping the section "Enable routing and IP masquerading" below, and run the wireless network in complete isolation.

Define the Wireless Interface IP Configuration

The Raspberry Pi runs a DHCP server for the wireless network; this requires static IP configuration for the wireless interface (`wlan0`) in the Raspberry Pi. The Raspberry Pi also acts as the router on the wireless network, and as is customary, we will give it the first IP address in the network: `192.168.4.1`.

To configure the static IP address, edit the configuration file for `dhcpcd` with:

```
sudo nano /etc/dhcpcd.conf
```

Go to the end of the file and add the following:

```
interface wlan0
    static ip_address=192.168.4.1/24
    nohook wpa_supplicant
```

Enable Routing and IP Masquerading

This section configures the Raspberry Pi to let wireless clients access computers on the main (Ethernet) network, and from there the internet.

NOTE

> If you wish to block wireless clients from accessing the Ethernet network and the internet, skip this section.

To enable routing, i.e. to allow traffic to flow from one network to the other in the Raspberry Pi, create a file using the following command, with the contents below:

```
sudo nano /etc/sysctl.d/routed-ap.conf
```

File contents:

```
# Enable IPv4 routing
net.ipv4.ip_forward=1
```

Enabling routing will allow hosts from network `192.168.4.0/24` to reach the LAN and the main router towards the internet. In order to allow traffic between clients on this foreign wireless network and the internet without changing the configuration of the main router, the Raspberry Pi can substitute the IP address of wireless clients with its own IP address on the LAN using a "masquerade" firewall rule.

- The main router will see all outgoing traffic from wireless clients as coming from the Raspberry Pi, allowing communication with the internet.
- The Raspberry Pi will receive all incoming traffic, substitute the IP addresses back, and forward traffic to the original wireless client.

This process is configured by adding a single firewall rule in the Raspberry Pi:

```
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Now save the current firewall rules for IPv4 (including the rule above) and IPv6 to be loaded at boot by the `netfilter-persistent` service:

```
sudo netfilter-persistent save
```

Filtering rules are saved to the directory `/etc/iptables/`. If in the future you change the configuration of your firewall, make sure to save the configuration before rebooting.

Configure the DHCP and DNS services for the wireless network

The DHCP and DNS services are provided by `dnsmasq`. The default configuration file serves as a template for all possible configuration options, whereas we only need a few. It is easier to start from an empty file.

Rename the default configuration file and edit a new one:

```
sudo mv /etc/dnsmasq.conf /etc/dnsmasq.conf.orig
sudo nano /etc/dnsmasq.conf
```

Add the following to the file and save it:

```
interface=wlan0 # Listening interface
dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h
                # Pool of IP addresses served via DHCP
domain=wlan     # Local wireless DNS domain
address=/gw.wlan/192.168.4.1
                # Alias for this router
```

The Raspberry Pi will deliver IP addresses between `192.168.4.2` and `192.168.4.20`, with a lease time of 24 hours, to wireless DHCP clients. You should be able to reach the Raspberry Pi under the name `gw.wlan` from wireless clients.

NOTE

> There are three IP address blocks set aside for private networks. There is a Class A block from `10.0.0.0` to `10.255.255.255`, a Class B block from `172.16.0.0` to `172.31.255.255`, and probably the most frequently used, a Class C block from `192.168.0.0` to `192.168.255.255`.

There are many more options for `dnsmasq`; see the default configuration file (`/etc/dnsmasq.conf`) or the online documentation for details.

Ensure Wireless Operation

Countries around the world regulate the use of telecommunication radio frequency bands to ensure interference-free operation. The Linux OS helps users comply with these rules by allowing applications to be configured with a two-letter "WiFi country code", e.g. `US` for a computer used in the United States.

In the Raspberry Pi OS, 5 GHz wireless networking is disabled until a WiFi country code has been configured by the user, usually as part of the initial installation process (see wireless configuration pages in this section for details.)

To ensure WiFi radio is not blocked on your Raspberry Pi, execute the following command:

```
sudo rfkill unblock wlan
```

This setting will be automatically restored at boot time. We will define an appropriate country code in the access point software configuration, next.

Configure the AP Software

Create the `hostapd` configuration file, located at `/etc/hostapd/hostapd.conf`, to add the various parameters for your new wireless network.

```
sudo nano /etc/hostapd/hostapd.conf
```

Add the information below to the configuration file. This configuration assumes we are using channel 7, with a network name of `NameOfNetwork`, and a password `AardvarkBadgerHedgehog`. Note that the name and password should **not** have quotes around them. The passphrase should be between 8 and 64 characters in length.

```
country_code=GB
interface=wlan0
ssid=NameOfNetwork
hw_mode=g
channel=7
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=AardvarkBadgerHedgehog
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

Note the line `country_code=GB`: it configures the computer to use the correct wireless frequencies in the United Kingdom. **Adapt this line** and specify the two-letter ISO code of your country. See Wikipedia for a list of two-letter ISO 3166-1 country codes.

To use the 5 GHz band, you can change the operations mode from `hw_mode=g` to `hw_mode=a`. Possible values for `hw_mode` are:

- a = IEEE 802.11a (5 GHz) (Raspberry Pi 3B+ onwards)
- b = IEEE 802.11b (2.4 GHz)
- g = IEEE 802.11g (2.4 GHz)

Note that when changing the `hw_mode`, you may need to also change the `channel` - see Wikipedia for a list of allowed combinations.

Running the new Wireless AP

Now restart your Raspberry Pi and verify that the wireless access point becomes automatically available.

```
sudo systemctl reboot
```

Once your Raspberry Pi has restarted, search for wireless networks with your wireless client. The network SSID you specified in file `/etc/hostapd/hostapd.conf` should now be present, and it should be accessible with the specified password.

If SSH is enabled on the Raspberry Pi, it should be possible to connect to it from your wireless client as follows, assuming the `pi` account is present: `ssh pi@192.168.4.1` or `ssh pi@gw.wlan`

If your wireless client has access to your Raspberry Pi (and the internet, if you set up routing), congratulations on setting up your new access point!

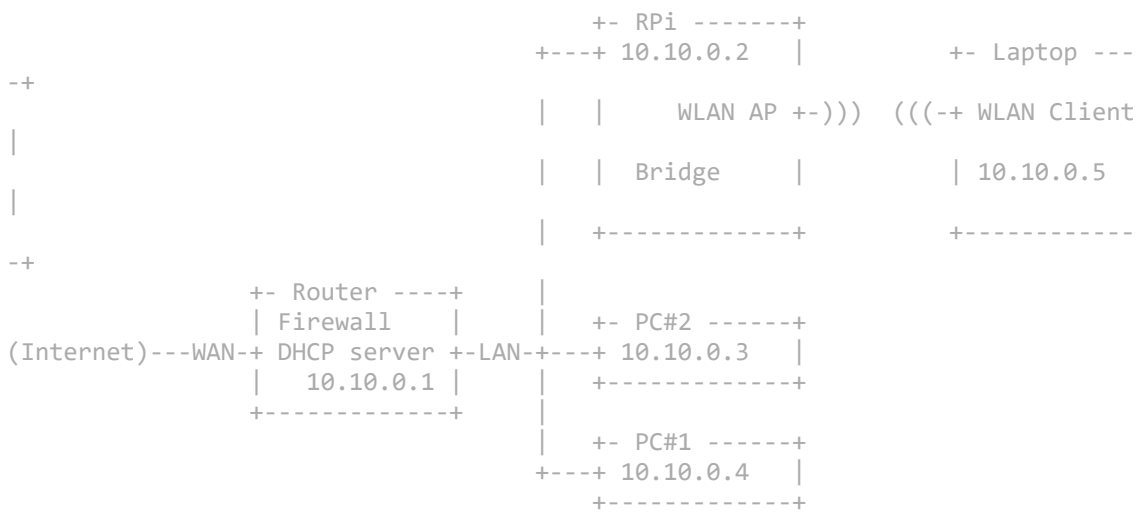
If you encounter difficulties, contact the forums for assistance. Please refer to this page in your message.

Setting up a Bridged Wireless Access Point

Edit this on GitHub

The Raspberry Pi can be used as a bridged wireless access point within an existing Ethernet network. This will extend the network to wireless computers and devices.

If you wish to create a standalone wireless network, consider instead setting up a routed access point.



A bridged wireless access point can be created using the inbuilt wireless features of the Raspberry Pi 4, Raspberry Pi 3 or Raspberry Pi Zero W, or by using a suitable USB wireless dongle that supports access point mode. It is possible that some USB dongles may need

slight changes to their settings. If you are having trouble with a USB wireless dongle, please check the forums.

This documentation was tested on a Raspberry Pi 3B running a fresh installation of Raspberry Pi OS Buster.

Before you Begin

- Ensure you have administrative access to your Raspberry Pi. The network setup will be entirely reset as part of the installation: local access, with screen and keyboard connected to your Raspberry Pi, is recommended.

  NOTE

  > If installing remotely via SSH, connect to your Raspberry Pi **by name** rather than by IP address, e.g. `ssh pi@raspberrypi.local`, as the address of your Raspberry Pi on the network will probably change after installation. You should also be ready to add screen and keyboard if needed in case you lose contact with your Raspberry Pi after installation.

- Connect your Raspberry Pi to the Ethernet network and boot the Raspberry Pi OS.
- Ensure the Raspberry Pi OS on your Raspberry Pi is up-to-date and reboot if packages were installed in the process.
- Have a wireless client (laptop, smartphone, …) ready to test your new access point.

Install AP and Management Software

In order to work as a bridged access point, the Raspberry Pi needs to have the `hostapd` access point software package installed:

```
sudo apt install hostapd
```

Enable the wireless access point service and set it to start when your Raspberry Pi boots:

```
sudo systemctl unmask hostapd
sudo systemctl enable hostapd
```

Software installation is complete. We will configure the access point software later on.

Setup the Network Bridge

A bridge network device running on the Raspberry Pi will connect the Ethernet and wireless networks using its built-in interfaces.

Create a bridge device and populate the bridge

Add a bridge network device named `br0` by creating a file using the following command, with the contents below:

```
sudo nano /etc/systemd/network/bridge-br0.netdev
```

File contents:

```
[NetDev]
Name=br0
Kind=bridge
```

In order to bridge the Ethernet network with the wireless network, first add the built-in Ethernet interface (`eth0`) as a bridge member by creating the following file:

```
sudo nano /etc/systemd/network/br0-member-eth0.network
```

File contents:

```
[Match]
Name=eth0

[Network]
Bridge=br0
```

NOTE

> The access point software will add the wireless interface `wlan0` to the bridge when the service starts. There is no need to create a file for that interface. This situation is particular to wireless LAN interfaces.

Now enable the `systemd-networkd` service to create and populate the bridge when your Raspberry Pi boots:

```
sudo systemctl enable systemd-networkd
```

Define the bridge device IP configuration

Network interfaces that are members of a bridge device are never assigned an IP address, since they communicate via the bridge. The bridge device itself needs an IP address, so that you can reach your Raspberry Pi on the network.

`dhcpcd`, the DHCP client on the Raspberry Pi, automatically requests an IP address for every active interface. So we need to block the `eth0` and `wlan0` interfaces from being processed, and let `dhcpcd` configure only `br0` via DHCP.

```
sudo nano /etc/dhcpcd.conf
```

Add the following line near the beginning of the file (above the first `interface xxx` line, if any):

```
denyinterfaces wlan0 eth0
```

Go to the end of the file and add the following:

```
interface br0
```

With this line, interface `br0` will be configured in accordance with the defaults via DHCP. Save the file to complete the IP configuration of the machine.

Ensure Wireless Operation

Countries around the world regulate the use of telecommunication radio frequency bands to ensure interference-free operation. The Linux OS helps users comply with these rules by allowing applications to be configured with a two-letter "WiFi country code", e.g. `US` for a computer used in the United States.

In the Raspberry Pi OS, 5 GHz wireless networking is disabled until a WiFi country code has been configured by the user, usually as part of the initial installation process (see wireless configuration pages in this section for details.)

To ensure WiFi radio is not blocked on your Raspberry Pi, execute the following command:

```
sudo rfkill unblock wlan
```

This setting will be automatically restored at boot time. We will define an appropriate country code in the access point software configuration, next.

Configure the AP Software

Create the `hostapd` configuration file, located at `/etc/hostapd/hostapd.conf`, to add the various parameters for your new wireless network.

```
sudo nano /etc/hostapd/hostapd.conf
```

Add the information below to the configuration file. This configuration assumes we are using channel 7, with a network name of `NameOfNetwork`, and a password `AardvarkBadgerHedgehog`. Note that the name and password should **not** have quotes around them. The passphrase should be between 8 and 64 characters in length.

```
country_code=GB
interface=wlan0
bridge=br0
ssid=NameOfNetwork
hw_mode=g
channel=7
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=AardvarkBadgerHedgehog
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

Note the lines `interface=wlan0` and `bridge=br0`: these direct `hostapd` to add the `wlan0` interface as a bridge member to `br0` when the access point starts, completing the bridge between Ethernet and wireless.

Note the line `country_code=GB`: it configures the computer to use the correct wireless frequencies in the United Kingdom. **Adapt this line** and specify the two-letter ISO code of your country. See Wikipedia for a list of two-letter ISO 3166-1 country codes.

To use the 5 GHz band, you can change the operations mode from `hw_mode=g` to `hw_mode=a`. Possible values for `hw_mode` are:

- a = IEEE 802.11a (5 GHz) (Raspberry Pi 3B+ onwards)
- b = IEEE 802.11b (2.4 GHz)
- g = IEEE 802.11g (2.4 GHz)

Note that when changing the `hw_mode`, you may need to also change the `channel` - see Wikipedia for a list of allowed combinations.

Run the new Wireless AP

Now restart your Raspberry Pi and verify that the wireless access point becomes automatically available.

```
sudo systemctl reboot
```

Once your Raspberry Pi has restarted, search for wireless networks with your wireless client. The network SSID you specified in file `/etc/hostapd/hostapd.conf` should now be present, and it should be accessible with the specified password.

If your wireless client has access to the local network and the internet, congratulations on setting up your new access point!

If you encounter difficulties, contact the forums for assistance. Please refer to this page in your message.

Using a Proxy Server

Edit this on GitHub

If you want your Raspberry Pi to access the Internet via a proxy server (perhaps from a school or other workplace), you will need to configure your Raspberry Pi to use the server before you can get online.

You will need:

- The IP address or hostname and port of your proxy server
- A username and password for your proxy (if required)

Configuring your Raspberry Pi

You will need to set up three environment variables (`http_proxy`, `https_proxy`, and `no_proxy`) so your Raspberry Pi knows how to access the proxy server.

Open a terminal window, and open the file `/etc/environment` using nano:

```
sudo nano /etc/environment
```

Add the following to the `/etc/environment` file to create the `http_proxy` variable:

```
export http_proxy="http://proxyipaddress:proxyport"
```

Replace `proxyipaddress` and `proxyport` with the IP address and port of your proxy.

NOTE

> If your proxy requires a username and password, add them using the following format:

```
export http_proxy="http://username:password@proxyipaddress:proxyport"
```

Enter the same information for the environment variable `https_proxy`:

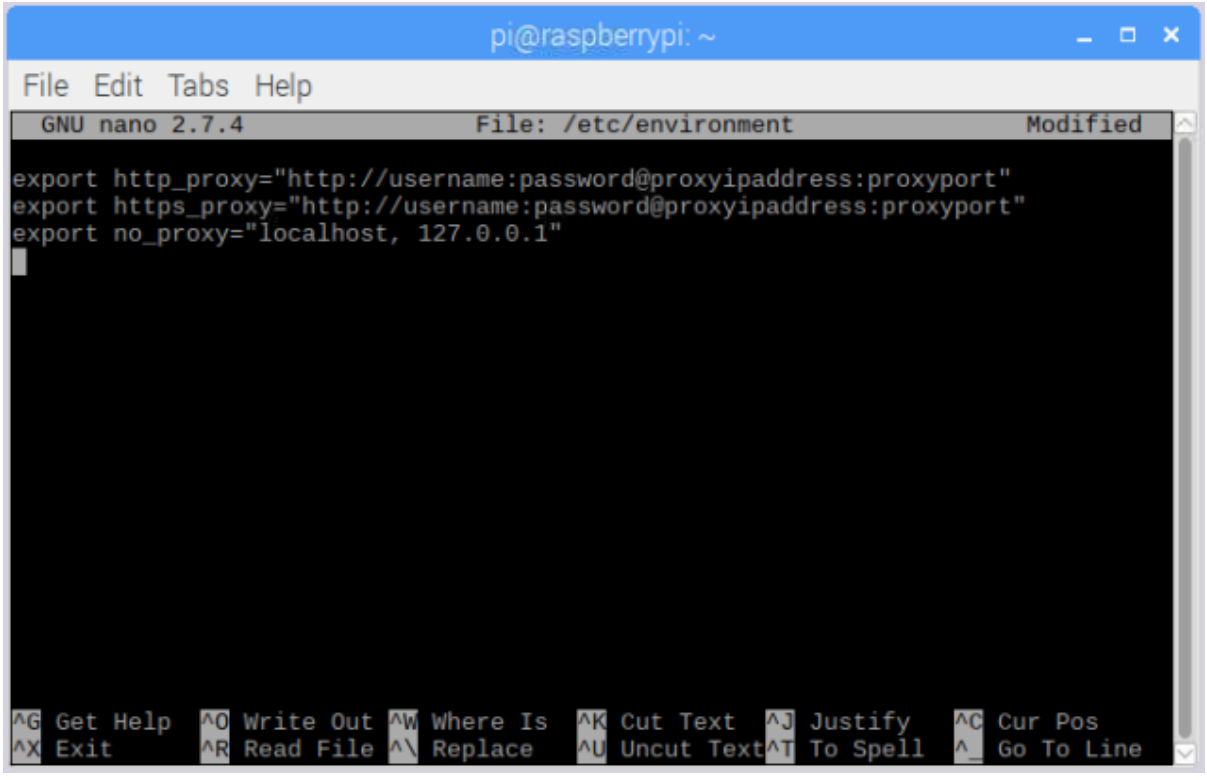
```
export https_proxy="http://username:password@proxyipaddress:proxyport"
```

Create the `no_proxy` environment variable, which is a comma-separated list of addresses your Raspberry Pi should not use the proxy for:

```
export no_proxy="localhost, 127.0.0.1"
```

Your `/etc/environment` file should now look like this:

```
export http_proxy="http://username:password@proxyipaddress:proxyport"
export https_proxy="http://username:password@proxyipaddress:proxyport"
export no_proxy="localhost, 127.0.0.1"
```



Press `Ctrl + X` to save and exit.

Update the sudoers File

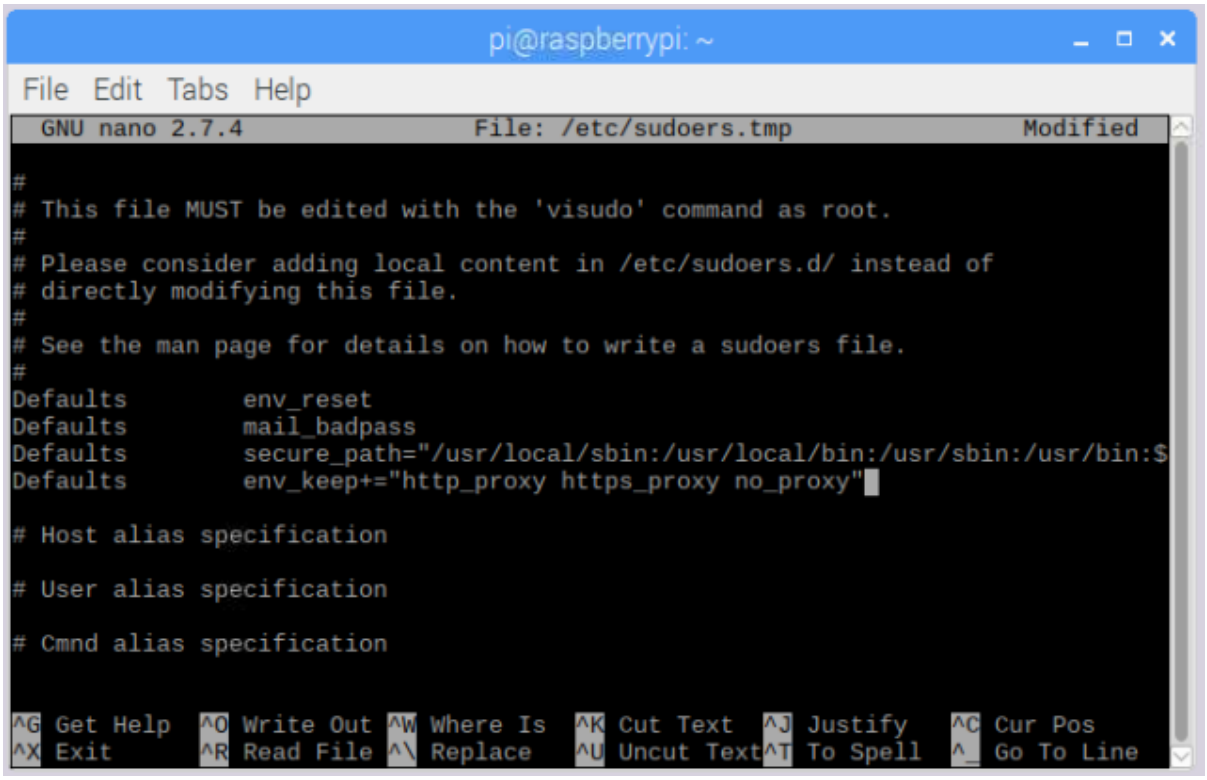
In order for operations that run as `sudo` (e.g. downloading and installing software) to use the new environment variables, you'll need to update `sudoers`.

Use the following command to open `sudoers`:

```
sudo visudo
```

Add the following line to the file so `sudo` will use the environment variables you just created:

```
Defaults        env_keep+="http_proxy https_proxy no_proxy"
```



Press `Ctrl + X` to save and exit.

Reboot your Raspberry Pi

Reboot your Raspberry Pi for the changes to take effect. You should now be able to access the internet via your proxy server.

HDMI Configuration

Edit this on GitHub

In the vast majority of cases, simply plugging your HDMI-equipped monitor into the Raspberry Pi using a standard HDMI cable will automatically result in the Raspberry Pi using the best resolution the monitor supports. The Raspberry Pi Zero, Zero W and Zero 2 W use a mini HDMI port, so you will need a mini-HDMI-to-full-size-HDMI lead or adapter. On the Raspberry Pi 4 and Raspberry Pi 400 there are two micro HDMI ports, so you will need a micro-HDMI-to-full-size-HDMI lead or adapter for each display you wish to attach. You should connect any HDMI leads before turning on the Raspberry Pi.

The Raspberry Pi 4 can drive up to two displays, with a resolution up to 1080p at a 60Hz refresh rate. At 4K resolution, if you connect two displays then you are limited to a 30Hz refresh rate. You can also drive a single display at 4K with a 60Hz refresh rate: this requires that the display is attached to the HDMI port adjacent to the USB-C power input (labelled HDMI0). You must also enable 4Kp60 output by setting the `hdmi_enable_4kp60=1` flag in config.txt. This flag can also be set using the 'Raspberry Pi Configuration' tool within the desktop environment.

If you are running the 3D graphics driver (also known as the FKMS driver), then in the Preferences menu you will find a graphical application for setting up standard displays, including multi-display setups.

NOTE

The Screen Configuration tool (`arandr`) is a graphical tool for selecting display modes and setting up multiple displays. You can find this tool in the desktop Preferences menu, but only if the 3D graphics driver is being used, as it is this driver that provides the required mode setting functionality. Use the Configure menu option to select the screen, resolution, and orientation. If you're using a multi-screen setup, drag around the displays to any position you want. When you have the required setup, click the Tick button to apply the settings.

If you are using legacy graphics drivers, or find yourself in circumstances where the Raspberry Pi may not be able to determine the best mode, or you may specifically wish to set a non-default resolution, the rest of this page may be useful.

NOTE

All the commands are documented fully in the config.txt section of the documentation.

HDMI Groups and Mode

HDMI has two common groups: CEA (Consumer Electronics Association, the standard typically used by TVs) and DMT (Display Monitor Timings, the standard typically used by monitors). Each group advertises a particular set of modes, where a mode describes the resolution, frame rate, clock rate, and aspect ratio of the output.

What Modes does my Device Support?

You can use the `tvservice` application on the command line to determine which modes are supported by your device, along with other useful data:

- `tvservice -s` displays the current HDMI status, including mode and resolution
- `tvservice -m CEA` lists all supported CEA modes
- `tvservice -m DMT` lists all supported DMT modes

If you are using a Raspberry Pi 4 with more than one display attached, then `tvservice` needs to be told which device to ask for information. You can get display IDs for all attached devices by using:

```
tvservice -l
```

You can specify which display `tvservice` uses by adding `-v <display id>` to the `tvservice` command, e.g:

- `tvservice -v 7 -m CEA`, lists all supported CEA modes for display ID 7

Setting a Specific HDMI Mode

Setting a specific mode is done using the `hdmi_group` and `hdmi_mode` config.txt entries. The group entry selects between CEA or DMT, and the mode selects the resolution and frame rate. You can find tables of modes on the config.txt Video Configuration page, but you should use the `tvservice` command described above to find out exactly which modes your device supports.

On the Raspberry Pi 4 and Raspberry Pi 400 to specify the HDMI port, add an index identifier to the `hdmi_group` or `hdmi_mode` entry in config.txt, e.g. `hdmi_mode:0` or `hdmi_group:1`.

Setting a Custom HDMI Mode

There are two options for setting a custom mode: `hdmi_cvt` and `hdmi_timings`.

`hdmi_cvt` sets a custom Coordinated Video Timing entry, which is described fully here: Video Configuration

In certain rare cases it may be necessary to define the exact clock requirements of the HDMI signal. This is a fully custom mode, and it is activated by setting `hdmi_group=2` and `hdmi_mode=87`. You can then use the `hdmi_timings` config.txt command to set the specific parameters for your display. `hdmi_timings` specifies all the timings that an HDMI signal needs to use. These timings are usually found in the datasheet of the display being used.

```
hdmi_timings=<h_active_pixels> <h_sync_polarity> <h_front_porch> <h_sync_pulse>
<h_back_porch> <v_active_pixels> <h_sync_polarity> <h_front_porch> <h_sync_puls
e> <h_back_porch> <v_active_lines> <v_sync_polarity> v_front_porch> <v_sync_pul
se> <v_back_porch> <v_sync_offset_a> <v_sync_offset_b> <pixel_rep> <frame_rate>
<interlaced> <pixel_freq> <aspect_ratio>
```

Timing	Purpose
`h_active_pixels`	The horizontal resolution
`h_sync_polarity`	0 or 1 to define the horizontal sync polarity
`h_front_porch`	Number of horizontal front porch pixels
`h_sync_pulse`	Width of horizontal sync pulse
`h_back_porch`	Number of horizontal back porch pixels
`v_active_lines`	The vertical resolution
`v_sync_polarity`	0 or 1 to define the vertical sync polarity
`v_front_porch`	Number of vertical front porch pixels
`v_sync_pulse`	Width of vertical sync pulse
`v_back_porch`	Number of vertical back porch pixels
`v_sync_offset_a`	Leave at 0
`v_sync_offset_b`	Leave at 0
`pixel_rep`	Leave at 0
`frame_rate`	Frame rate of mode
`interlaced`	0 for non-interlaced, 1 for interlaced
`pixel_freq`	The mode pixel frequency
`aspect_ratio`	The aspect ratio required

`aspect_ratio` should be one of the following:

Ratio	`aspect_ratio` ID
`4:3`	1
`14:9`	2
`16:9`	3
`5:4`	4
`16:10`	5
`15:9`	6
`21:9`	7
`64:27`	8

For the Raspberry Pi 4 and Raspberry Pi 400 to specify the HDMI port, you can add an index identifier to the config.txt. e.g. `hdmi_cvt:0=...` or `hdmi_timings:1=...`. If no port identifier is specified, the settings are applied to port 0.

Troubleshooting your HDMI

In some rare cases you may need to increase the HDMI drive strength, for example when there is speckling on the display or when you are using very long cables. There is a config.txt item to do this, `config_hdmi_boost`, which is documented on the config.txt video page.

NOTE

> The Raspberry Pi 4B does not yet support `config_hdmi_boost`, support for this option will be added in a future software update.

Rotating your Display

Edit this on GitHub

The options to rotate the display of your Raspberry Pi depend on which display driver software it is running, which may also depend on which Raspberry Pi you are using.

Fake or Full KMS Graphics Driver

NOTE

> This is the default for Raspberry Pi 4 Model B.

If you are running the Raspberry Pi desktop then rotation is achieved by using the `Screen Configuration Utility` from the desktop `Preferences` menu. This will bring up a graphical representation of the display or displays connected to the Raspberry Pi. Right click on the display you wish to rotate and select the required option.

It is also possible to change these settings using the command line `xrandr` option. The following commands give 0°, -90°, +90° and 180° rotations respectively.

```
xrandr --output HDMI-1 --rotate normal
xrandr --output HDMI-1 --rotate left
xrandr --output HDMI-1 --rotate right
xrandr --output HDMI-1 --rotate inverted
```

Note that the `--output` entry specifies to which device the rotation applies. You can determine the device name by simply typing `xrandr` on the command line which will display information, including the name, for all attached devices.

You can also use the command line to mirror the display using the `--reflect` option. Reflection can be one of 'normal' 'x', 'y' or 'xy'. This causes the output contents to be reflected across the specified axes. For example:

```
xrandr --output HDMI-1 --reflect x
```

If you are using the console only (no graphical desktop) then you will need to set the appropriate kernel command line flags. Change the console settings as described on the this page.

Legacy Graphics Driver

NOTE

> This is the default for models prior to the Raspberry Pi 4 Model B.

There are `config.txt` options for rotating when using the legacy display drivers.

`display_hdmi_rotate` is used to rotate the HDMI display, `display_lcd_rotate` is used to rotate any attached LCD panel (using the DSI or DPI interface). These options rotate both the desktop and console. Each option takes one of the following parameters :

display_*_rotate	**result**
0	no rotation
1	rotate 90 degrees clockwise
2	rotate 180 degrees clockwise
3	rotate 270 degrees clockwise
0x10000	horizontal flip
0x20000	vertical flip

Note that the 90 and 270 degree rotation options require additional memory on the GPU, so these will not work with the 16MB GPU split.

You can combine the rotation settings with the flips by adding them together. You can also have both horizontal and vertical flips in the same way. E.g. A 180 degree rotation with a vertical and horizontal flip will be 0x20000 + 0x10000 + 2 = 0x30002.

Audio Configuration

The Raspberry Pi has up to three audio output modes: HDMI 1 and 2, if present, and a headphone jack. You can switch between these modes at any time.

If your HDMI monitor or TV has built-in speakers, the audio can be played over the HDMI cable, but you can switch it to a set of headphones or other speakers plugged into the headphone jack. If your display claims to have speakers, sound is output via HDMI by default; if not, it is output via the headphone jack. This may not be the desired output setup, or the auto-detection is inaccurate, in which case you can manually switch the output.

Changing the Audio Output

There are two ways of setting the audio output; using the desktop volume control, or using the `raspi-config` command line tool.

Using the Desktop

Right-clicking the volume icon on the desktop taskbar brings up the audio output selector; this allows you to select between the internal audio outputs. It also allows you to select any external audio devices, such as USB sound cards and Bluetooth audio devices. A green tick is shown against the currently selected audio output device — simply left-click the desired output in the pop-up menu to change this. The volume control and mute operate on the currently selected device.

Using raspi-config

Open up raspi-config by entering the following into the command line:

```
sudo raspi-config
```

This will open the configuration screen:

Select `System Options` (Currently option 1, but yours may be different) and press `Enter`.

Now select the Option named, `Audio` (Currently option S2, but yours may be different) and press `Enter`:

Select your required mode, press `Enter` and press the right arrow key to exit the options list, then select `Finish` to exit the configuration tool.

After you have finished modifying your audio settings, you need to restart your Raspberry Pi in order for your changes to take effect.

Troubleshooting your HDMI

In some rare cases, it is necessary to edit `config.txt` to force HDMI mode (as opposed to DVI mode, which does not send sound). You can do this by editing `/boot/config.txt` and setting `hdmi_drive=2`, then rebooting for the change to take effect.

External Storage Configuration

Edit this on GitHub

You can connect your external hard disk, SSD, or USB stick to any of the USB ports on the Raspberry Pi, and mount the file system to access the data stored on it.

By default, your Raspberry Pi automatically mounts some of the popular file systems such as FAT, NTFS, and HFS+ at the `/media/pi/<HARD-DRIVE-LABEL>` location.

NOTE

> Raspberry Pi OS Lite does not implement automounting.

To set up your storage device so that it always mounts to a specific location of your choice, you must mount it manually.

Mounting a Storage Device

You can mount your storage device at a specific folder location. It is conventional to do this within the `/mnt` folder, for example `/mnt/mydisk`. Note that the folder must be empty.

1. Plug the storage device into a USB port on the Raspberry Pi.
2. List all the disk partitions on the Raspberry Pi using the following command:

   ```
   sudo lsblk -o UUID,NAME,FSTYPE,SIZE,MOUNTPOINT,LABEL,MODEL
   ```

   The Raspberry Pi uses mount points `/` and `/boot`. Your storage device will show up in this list, along with any other connected storage.
3. Use the SIZE, LABEL, and MODEL columns to identify the name of the disk partition that points to your storage device. For example, `sda1`.
4. The FSTYPE column contains the filesystem type. If your storage device uses an exFAT file system, install the exFAT driver:

   ```
   sudo apt update
   sudo apt install exfat-fuse
   ```

5. If your storage device uses an NTFS file system, you will have read-only access to it. If you want to write to the device, you can install the ntfs-3g driver:

```
sudo apt update
sudo apt install ntfs-3g
```

6. Run the following command to get the location of the disk partition:

```
sudo blkid
```

   For example, `/dev/sda1`.

7. Create a target folder to be the mount point of the storage device. The mount point name used in this case is `mydisk`. You can specify a name of your choice:

```
sudo mkdir /mnt/mydisk
```

8. Mount the storage device at the mount point you created:

```
sudo mount /dev/sda1 /mnt/mydisk
```

9. Verify that the storage device is mounted successfully by listing the contents:

```
ls /mnt/mydisk
```

Setting up Automatic Mounting

You can modify the `fstab` file to define the location where the storage device will be automatically mounted when the Raspberry Pi starts up. In the `fstab` file, the disk partition is identified by the universally unique identifier (UUID).

1. Get the UUID of the disk partition:

```
sudo blkid
```

2. Find the disk partition from the list and note the UUID. For example, `5C24-1453`.
3. Open the fstab file using a command line editor such as nano:

```
sudo nano /etc/fstab
```

4. Add the following line in the `fstab` file:

```
UUID=5C24-1453 /mnt/mydisk fstype defaults,auto,users,rw,nofail 0 0
```

   Replace `fstype` with the type of your file system, which you found in step 2 of 'Mounting a storage device' above, for example: `ntfs`.

5. If the filesystem type is FAT or NTFS, add `,umask=000` immediately after `nofail` - this will allow all users full read/write access to every file on the storage device.

Now that you have set an entry in `fstab`, you can start up your Raspberry Pi with or without the storage device attached. Before you unplug the device you must either shut down the Raspberry Pi, or manually unmount it using the steps in 'Unmounting a storage device' below.

NOTE

> If you do not have the storage device attached when the Raspberry Pi starts, the Raspberry Pi will take an extra 90 seconds to start up. You can shorten this by adding `,x-systemd.device-timeout=30` immediately after `nofail` in step 4. This will change the timeout to 30 seconds, meaning the system will only wait 30 seconds before giving up trying to mount the disk.

For more information on each Linux command, refer to the specific manual page using the `man` command. For example, `man fstab`.

Unmounting a Storage Device

When the Raspberry Pi shuts down, the system takes care of unmounting the storage device so that it is safe to unplug it. If you want to manually unmount a device, you can use the following command:

```
sudo umount /mnt/mydisk
```

If you receive an error that the 'target is busy', this means that the storage device was not unmounted. If no error was displayed, you can now safely unplug the device.

Dealing with 'target is busy'

The 'target is busy' message means there are files on the storage device that are in use by a program. To close the files, use the following procedure.

1. Close any program which has open files on the storage device.

2. If you have a terminal open, make sure that you are not in the folder where the storage device is mounted, or in a sub-folder of it.

3. If you are still unable to unmount the storage device, you can use the `lsof` tool to check which program has files open on the device. You need to first install `lsof` using `apt`:

   ```
   sudo apt update
   sudo apt install lsof
   ```

   To use lsof:

   ```
   lsof /mnt/mydisk
   ```

Localising your Raspberry Pi

Edit this on GitHub

You can set your Raspberry Pi up to match your regional settings.

Changing the Language

If you want to select a different language use raspi-config.

Configuring the Keyboard

If you want to select a different keyboard use raspi-config.

Changing the Timezone

Once again, this is something you can change using the raspi-config tool.

Changing the default pin configuration

Edit this on GitHub

WARNING

> This feature is intended for advanced users.

As of July 2014, the Raspberry Pi firmware supports custom default pin configurations through a user-provided Device Tree blob file. To find out whether your firmware is recent

enough, please run `vcgencmd version`.

Device Pins During Boot Sequence

During the bootup sequence, the GPIO pins go through various actions.

1. Power-on — pins default to inputs with default pulls; the default pulls for each pin are described in the datasheet
2. Setting by the bootrom
3. Setting by `bootcode.bin`
4. Setting by `dt-blob.bin` (this page)
5. Setting by the GPIO command in `config.txt`
6. Additional firmware pins (e.g. UARTS)
7. Kernel/Device Tree

On a soft reset, the same procedure applies, except for default pulls, which are only applied on a power-on reset.

Note that it may take a few seconds to get from stage 1 to stage 4. During that time, the GPIO pins may not be in the state expected by attached peripherals (as defined in `dtblob.bin` or `config.txt`). Since different GPIO pins have different default pulls, you should do **one of the following** for your peripheral:

- Choose a GPIO pins that defaults to pulls as required by the peripheral on reset
- Delay the peripheral's startup until stage 4/5 has been reached
- Add an appropriate pull-up/-down resistor

Providing a Custom Device Tree Blob

In order to compile a Device Tree source (`.dts`) file into a Device Tree blob (`.dtb`) file, the Device Tree compiler must be installed by running `sudo apt install device-tree-compiler`. The `dtc` command can then be used as follows:

```
sudo dtc -I dts -O dtb -o /boot/dt-blob.bin dt-blob.dts
```

Similarly, a `.dtb` file can be converted back to a `.dts` file, if required.

```
dtc -I dtb -O dts -o dt-blob.dts /boot/dt-blob.bin
```

Sections of the `dt-blob`

The `dt-blob.bin` is used to configure the binary blob (VideoCore) at boot time. It is not currently used by the Linux kernel, but a kernel section will be added at a later stage, when we reconfigure the Raspberry Pi kernel to use a dt-blob for configuration. The dt-blob can configure all versions of the Raspberry Pi, including the Compute Module, to use the alternative settings. The following sections are valid in the dt-blob:

1. `videocore`

   This section contains all of the VideoCore blob information. All subsequent sections must be enclosed within this section.

2. `pins_*`

   There are a number of separate `pins_*` sections, based on particular Raspberry Pi models, namely:

   - **pins_rev1** Rev1 pin setup. There are some differences because of the moved I2C pins.
   - **pins_rev2** Rev2 pin setup. This includes the additional codec pins on P5.
   - **pins_bplus1** Raspberry Pi 1 Model B+ rev 1.1, including the full 40pin connector.
   - **pins_bplus2** Raspberry Pi 1 Model B+ rev 1.2, swapping the low-power and lan-run pins.
   - **pins_aplus** Raspberry Pi 1 Model A+, lacking Ethernet.
   - **pins_2b1** Raspberry Pi 2 Model B rev 1.0; controls the SMPS via I2C0.
   - **pins_2b2** Raspberry Pi 2 Model B rev 1.1; controls the SMPS via software I2C on 42 and 43.
   - **pins_3b1** Raspberry Pi 3 Model B rev 1.0
   - **pins_3b2** Raspberry Pi 3 Model B rev 1.2
   - **pins_3bplus** Raspberry Pi 3 Model B+
   - **pins_3aplus** Raspberry Pi 3 Model A+
   - **pins_pi0** Raspberry Pi Zero
   - **pins_pi0w** Raspberry Pi Zero W
   - **pins_cm** Raspberry Pi Compute Module 1. The default for this is the default for the chip, so it is a useful source of information about default pull ups/downs on the chip.

- **pins_cm3** Raspberry Pi Compute Module 3

  Each `pins_*` section can contain `pin_config` and `pin_defines` sections.

3. `pin_config`

   The `pin_config` section is used to configure the individual pins. Each item in this section must be a named pin section, such as `pin@p32`, meaning GPIO32. There is a special section `pin@default`, which contains the default settings for anything not specifically named in the pin_config section.

4. `pin@pinname`

   This section can contain any combination of the following items:

   a. `polarity`

      - `active_high`
      - `active_low`

   b. `termination`

      - `pull_up`
      - `pull_down`
      - `no_pulling`

   c. `startup_state`

      - `active`
      - `inactive`

   d. `function`

      - `input`
      - `output`
      - `sdcard`
      - `i2c0`
      - `i2c1`
      - `spi`
      - `spi1`
      - `spi2`

- `smi`
- `dpi`
- `pcm`
- `pwm`
- `uart0`
- `uart1`
- `gp_clk`
- `emmc`
- `arm_jtag`

e. `drive_strength_mA` The drive strength is used to set a strength for the pins. Please note that you can only specify a single drive strength for the bank. <8> and <16> are valid values.

5. `pin_defines`

   This section is used to set specific VideoCore functionality to particular pins. This enables the user to move the camera power enable pin to somewhere different, or move the HDMI hotplug position: things that Linux does not control. Please refer to the example DTS file below.

Clock Configuration

It is possible to change the configuration of the clocks through this interface, although it can be difficult to predict the results! The configuration of the clocking system is very complex. There are five separate PLLs, and each one has its own fixed (or variable, in the case of PLLC) VCO frequency. Each VCO then has a number of different channels which can be set up with a different division of the VCO frequency. Each of the clock destinations can be configured to come from one of the clock channels, although there is a restricted mapping of source to destination, so not all channels can be routed to all clock destinations.

Here are a couple of example configurations that you can use to alter specific clocks. We will add to this resource when requests for clock configurations are made.

```
clock_routing {
   vco@PLLA  {    freq = <1966080000>; };
   chan@APER {    div  = <4>; };
   clock@GPCLK0 { pll = "PLLA"; chan = "APER"; };
};

clock_setup {
   clock@PWM { freq = <2400000>; };
```

```
    clock@GPCLK0 { freq = <12288000>; };
    clock@GPCLK1 { freq = <25000000>; };
};
```

The above will set the PLLA to a source VCO running at 1.96608GHz (the limits for this VCO are 600MHz - 2.4GHz), change the APER channel to /4, and configure GPCLK0 to be sourced from PLLA through APER. This is used to give an audio codec the 12288000Hz it needs to produce the 48000 range of frequencies.

Sample Device Tree Source File

The example file comes from the firmware repository, https://github.com/raspberrypi/firmware/blob/master/extra/dt-blob.dts. This is the master Raspberry Pi blob, from which others are usually derived.

Device Trees, Overlays, and Parameters

Edit this on GitHub

Raspberry Pi kernels and firmware use a Device Tree (DT) to describe the hardware present in the Raspberry Pi. These Device Trees may include DT parameters that provide a degree of control over some onboard features. DT overlays allow optional external hardware to be described and configured, and they also support parameters for more control.

The firmware loader (`start.elf` and its variants) is responsible for loading the DTB (Device Tree Blob - a machine readable DT file). It chooses which one to load based on the board revision number, and makes certain modifications to further tailor it (memory size, Ethernet addresses etc.). This runtime customisation avoids the need for lots of DTBs with only minor differences.

`config.txt` is scanned for user-provided parameters, along with any overlays and their parameters, which are then applied. The loader examines the result to learn (for example) which UART, if any, is to be used for the console. Finally it launches the kernel, passing a pointer to the merged DTB.

Device Trees

A Device Tree (DT) is a description of the hardware in a system. It should include the name of the base CPU, its memory configuration, and any peripherals (internal and external). A DT should not be used to describe the software, although by listing the hardware modules it does usually cause driver modules to be loaded. It helps to remember that DTs are supposed to be OS-neutral, so anything which is Linux-specific probably shouldn't be there.

A Device Tree represents the hardware configuration as a hierarchy of nodes. Each node may contain properties and subnodes. Properties are named arrays of bytes, which may contain strings, numbers (big-endian), arbitrary sequences of bytes, and any combination thereof. By analogy to a filesystem, nodes are directories and properties are files. The locations of nodes and properties within the tree can be described using a path, with slashes as separators and a single slash (`/`) to indicate the root.

Basic DTS syntax

Device Trees are usually written in a textual form known as Device Tree Source (DTS) and stored in files with a `.dts` suffix. DTS syntax is C-like, with braces for grouping and semicolons at the end of each line. Note that DTS requires semicolons after closing braces: think of C `struct`s rather than functions. The compiled binary format is referred to as Flattened Device Tree (FDT) or Device Tree Blob (DTB), and is stored in `.dtb` files.

The following is a simple tree in the `.dts` format:

```
/dts-v1/;
/include/ "common.dtsi";

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        cousin: child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
            my-cousin = <&cousin>;
        };
    };
};

/node2 {
    another-property-for-node2;
};
```

This tree contains:

- a required header: `/dts-v1/`.
- The inclusion of another DTS file, conventionally named `*.dtsi` and analogous to a `.h` header file in C - see *An aside about /include/* below.
- a single root node: `/`

- a couple of child nodes: `node1` and `node2`
- some children for node1: `child-node1` and `child-node2`
- a label (`cousin`) and a reference to that label (`&cousin`): see *Labels and References* below.
- several properties scattered through the tree
- a repeated node (`/node2`) - see *An aside about /include/* below.

Properties are simple key-value pairs where the value can either be empty or contain an arbitrary byte stream. While data types are not encoded in the data structure, there are a few fundamental data representations that can be expressed in a Device Tree source file.

Text strings (NUL-terminated) are indicated with double quotes:

```
string-property = "a string";
```

Cells are 32-bit unsigned integers delimited by angle brackets:

```
cell-property = <0xbeef 123 0xabcd1234>;
```

Arbitrary byte data is delimited with square brackets, and entered in hex:

```
binary-property = [01 23 45 67 89 ab cd ef];
```

Data of differing representations can be concatenated using a comma:

```
mixed-property = "a string", [01 23 45 67], <0x12345678>;
```

Commas are also used to create lists of strings:

```
string-list = "red fish", "blue fish";
```

An aside about `/include/`

The `/include/` directive results in simple textual inclusion, much like C's `#include` directive, but a feature of the Device Tree compiler leads to different usage patterns. Given that nodes are named, potentially with absolute paths, it is possible for the same node to appear twice in a DTS file (and its inclusions). When this happens, the nodes and properties are combined, interleaving and overwriting properties as required (later values override earlier ones).

In the example above, the second appearance of `/node2` causes a new property to be added to the original:

```
/node2 {
    an-empty-property;
    a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
    another-property-for-node2;
    child-node1 {
        my-cousin = <&cousin>;
    };
};
```

It is thus possible for one `.dtsi` to overwrite, or provide defaults for, multiple places in a tree.

Labels and references

It is often necessary for one part of the tree to refer to another, and there are four ways to do this:

1. Path strings

   Paths should be self-explanatory, by analogy with a filesystem - `/soc/i2s@7e203000` is the full path to the I2S device in BCM2835 and BCM2836. Note that although it is easy to construct a path to a property (for example, `/soc/i2s@7e203000/status`), the standard APIs don't do that; you first find a node, then choose properties of that node.

2. phandles

   A phandle is a unique 32-bit integer assigned to a node in its `phandle` property. For historical reasons, you may also see a redundant, matching `linux,phandle`. phandles are numbered sequentially, starting from 1; 0 is not a valid phandle. They are usually allocated by the DT compiler when it encounters a reference to a node in an integer context, usually in the form of a label (see below). References to nodes using phandles are simply encoded as the corresponding integer (cell) values; there is no markup to indicate that they should be interpreted as phandles, as that is application-defined.

3. Labels

   Just as a label in C gives a name to a place in the code, a DT label assigns a name to a node in the hierarchy. The compiler takes references to labels and converts them into paths when used in string context (`&node`) and phandles in integer context (`<&node>`); the original labels do not appear in the compiled output. Note that labels contain no structure; they are just tokens in a flat, global namespace.

4. Aliases

Aliases are similar to labels, except that they do appear in the FDT output as a form of index. They are stored as properties of the `/aliases` node, with each property mapping an alias name to a path string. Although the aliases node appears in the source, the path strings usually appear as references to labels (`&node`), rather then being written out in full. DT APIs that resolve a path string to a node typically look at the first character of the path, treating paths that do not start with a slash as aliases that must first be converted to a path using the `/aliases` table.

Device Tree semantics

How to construct a Device Tree, and how best to use it to capture the configuration of some hardware, is a large and complex subject. There are many resources available, some of which are listed below, but several points deserve mentioning in this document:

`compatible` properties are the link between the hardware description and the driver software. When an OS encounters a node with a `compatible` property, it looks it up in its database of device drivers to find the best match. In Linux, this usually results in the driver module being automatically loaded, provided it has been appropriately labelled and not blacklisted.

The `status` property indicates whether a device is enabled or disabled. If the `status` is `ok`, `okay` or absent, then the device is enabled. Otherwise, `status` should be `disabled`, so that the device is disabled. It can be useful to place devices in a `.dtsi` file with the status set to `disabled`. A derived configuration can then include that `.dtsi` and set the status for the devices which are needed to `okay`.

Device Tree Overlays

A modern SoC (System on a Chip) is a very complicated device; a complete Device Tree could be hundreds of lines long. Taking that one step further and placing the SoC on a board with other components only makes matters worse. To keep that manageable, particularly if there are related devices that share components, it makes sense to put the common elements in `.dtsi` files, to be included from possibly multiple `.dts` files.

When a system like Raspberry Pi also supports optional plug-in accessories such as HATs, the problem grows. Ultimately, each possible configuration requires a Device Tree to describe it, but once you factor in all the different base models and the large number of available accessories, the number of combinations starts to multiply rapidly.

What is needed is a way to describe these optional components using a partial Device Tree, and then to be able to build a complete tree by taking a base DT and adding a number of optional elements. You can do this, and these optional elements are called "overlays".

Unless you want to learn how to write overlays for Raspberry Pis, you might prefer to skip on to Part 3: Using Device Trees on Raspberry Pi.

Fragments

A DT overlay comprises a number of fragments, each of which targets one node and its subnodes. Although the concept sounds simple enough, the syntax seems rather strange at first:

```
// Enable the i2s interface
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2835";

    fragment@0 {
        target = <&i2s>;
        __overlay__ {
            status = "okay";
            test_ref = <&test_label>;
            test_label: test_subnode {
                dummy;
            };
        };
    };
};
```

The `compatible` string identifies this as being for BCM2835, which is the base architecture for the Raspberry Pi SoCs; if the overlay makes use of features of a Raspberry Pi 4 then `brcm,bcm2711` is the correct value to use, otherwise `brcm,bcm2835` can be used for all Raspberry Pi overlays. Then comes the first (and in this case only) fragment. Fragments should be numbered sequentially from zero. Failure to adhere to this may cause some or all of your fragments to be missed.

Each fragment consists of two parts: a `target` property, identifying the node to apply the overlay to; and the `__overlay__` itself, the body of which is added to the target node. The example above can be interpreted as if it were written like this:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2835";
};

&i2s {
    status = "okay";
    test_ref = <&test_label>;
    test_label: test_subnode {
        dummy;
    };
};
```

(In fact, with a sufficiently new version of `dtc` you can write it exactly like that and get identical output, but some homegrown tools don't understand this format yet so any overlay that you might want to be included in the standard Raspberry Pi OS kernel should be written in the old format for now).

The effect of merging that overlay with a standard Raspberry Pi base Device Tree (e.g. `bcm2708-rpi-b-plus.dtb`), provided the overlay is loaded afterwards, would be to enable the I2S interface by changing its status to `okay`. But if you try to compile this overlay using:

```
dtc -I dts -O dtb -o 2nd.dtbo 2nd-overlay.dts
```

you will get an error:

```
Label or path i2s not found
```

This shouldn't be too unexpected, since there is no reference to the base `.dtb` or `.dts` file to allow the compiler to find the `i2s` label.

Trying again, this time using the original example and adding the `-@` option to allow unresolved references (and `-Hepapr` to remove some clutter):

```
dtc -@ -Hepapr -I dts -O dtb -o 1st.dtbo 1st-overlay.dts
```

If `dtc` returns an error about the third line, it doesn't have the extensions required for overlay work. Run `sudo apt install device-tree-compiler` and try again - this time, compilation should complete successfully. Note that a suitable compiler is also available in the kernel tree as `scripts/dtc/dtc`, built when the `dtbs` make target is used:

```
make ARCH=arm dtbs
```

It is interesting to dump the contents of the DTB file to see what the compiler has generated:

```
fdtdump 1st.dtbo
/dts-v1/;
// magic:               0xd00dfeed
// totalsize:           0x207 (519)
// off_dt_struct:       0x38
// off_dt_strings:      0x1c8
// off_mem_rsvmap:      0x28
// version:             17
// last_comp_version:   16
// boot_cpuid_phys:     0x0
// size_dt_strings:     0x3f
// size_dt_struct:      0x190

/ {
    compatible = "brcm,bcm2835";
    fragment@0 {
        target = <0xffffffff>;
        __overlay__ {
            status = "okay";
            test_ref = <0x00000001>;
            test_subnode {
```

```
                dummy;
                phandle = <0x00000001>;
            };
        };
    };
    __symbols__ {
        test_label = "/fragment@0/__overlay__/test_subnode";
    };
    __fixups__ {
        i2s = "/fragment@0:target:0";
    };
    __local_fixups__ {
        fragment@0 {
            __overlay__ {
                test_ref = <0x00000000>;
            };
        };
    };
};
```

After the verbose description of the file structure there is our fragment. But look carefully - where we wrote `&i2s` it now says `0xffffffff`, a clue that something strange has happened (older versions of dtc might say `0xdeadbeef` instead). The compiler has also added a `phandle` property containing a unique (to this overlay) small integer to indicate that the node has a label, and replaced all references to the label with the same small integer.

After the fragment there are three new nodes:

- `__symbols__` lists the labels used in the overlay (`test_label` here), and the path to the labelled node. This node is the key to how unresolved symbols are dealt with.
- `__fixups__` contains a list of properties mapping the names of unresolved symbols to lists of paths to cells within the fragments that need patching with the phandle of the target node, once that target has been located. In this case, the path is to the `0xffffffff` value of `target`, but fragments can contain other unresolved references which would require additional fixes.
- `__local_fixups__` holds the locations of any references to labels that exist within the overlay - the `test_ref` property. This is required because the program performing the merge will have to ensure that phandle numbers are sequential and unique.

Back in section 1.3 it says that "the original labels do not appear in the compiled output", but this isn't true when using the `-@` switch. Instead, every label results in a property in the `__symbols__` node, mapping a label to a path, exactly like the `aliases` node. In fact, the mechanism is so similar that when resolving symbols, the Raspberry Pi loader will search the "aliases" node in the absence of a `__symbols__` node. This was useful at one time because providing sufficient aliases allowed very old versions of `dtc` to be used to build the base DTB files, but fortunately that is ancient history now.

Device Tree parameters

To avoid the need for lots of Device Tree overlays, and to reduce the need for users of peripherals to modify DTS files, the Raspberry Pi loader supports a new feature - Device Tree parameters. This permits small changes to the DT using named parameters, similar to the way kernel modules receive parameters from `modprobe` and the kernel command line. Parameters can be exposed by the base DTBs and by overlays, including HAT overlays.

Parameters are defined in the DTS by adding an `__overrides__` node to the root. It contains properties whose names are the chosen parameter names, and whose values are a sequence comprising a phandle (reference to a label) for the target node, and a string indicating the target property; string, integer (cell) and boolean properties are supported.

String parameters

String parameters are declared like this:

```
name = <&label>,"property";
```

where `label` and `property` are replaced by suitable values. String parameters can cause their target properties to grow, shrink, or be created.

Note that properties called `status` are treated specially; non-zero/true/yes/on values are converted to the string `"okay"`, while zero/false/no/off becomes `"disabled"`.

Integer parameters

Integer parameters are declared like this:

```
name = <&label>,"property.offset"; // 8-bit
name = <&label>,"property;offset"; // 16-bit
name = <&label>,"property:offset"; // 32-bit
name = <&label>,"property#offset"; // 64-bit
```

where `label`, `property` and `offset` are replaced by suitable values; the offset is specified in bytes relative to the start of the property (in decimal by default), and the preceding separator dictates the size of the parameter. In a change from earlier implementations, integer parameters may refer to non-existent properties or to offsets beyond the end of an existing property.

Boolean parameters

Device Tree encodes boolean values as zero-length properties; if present then the property is true, otherwise it is false. They are defined like this:

```
boolean_property; // Set 'boolean_property' to true
```

Note that a property is assigned the value `false` by not defining it. Boolean parameters are declared like this:

```
name = <&label>,"property?";
```

where `label` and `property` are replaced by suitable values.

Inverted booleans invert the input value before applying it in the same was as a regular boolean; they are declared similarly, but use `!` to indicate the inversion:

```
name = <&label>,"property!";
```

Boolean parameters can cause properties to be created or deleted, but they can't delete a property that already exists in the base DTB.

Byte string parameters

Byte string properties are arbitrary sequences of bytes, e.g. MAC addresses. They accept strings of hexadecimal bytes, with or without colons between the bytes.

```
mac_address = <&ethernet0>,"local_mac_address[";
```

The `[` was chosen to match the DT syntax for declaring a byte string:

```
local_mac_address = [aa bb cc dd ee ff];
```

Parameters with multiple targets

There are some situations where it is convenient to be able to set the same value in multiple locations within the Device Tree. Rather than the ungainly approach of creating multiple parameters, it is possible to add multiple targets to a single parameter by concatenating them, like this:

```
__overrides__ {
    gpiopin = <&w1>,"gpios:4",
              <&w1_pins>,"brcm,pins:0";
    ...
};
```

(example taken from the `w1-gpio` overlay)

NOTE

> It is even possible to target properties of different types with a single parameter. You could reasonably connect an "enable" parameter to a `status` string, cells containing

zero or one, and a proper boolean property.

Literal assignments

As seen in 2.2.5, the DT parameter mechanism allows multiple targets to be patched from the same parameter, but the utility is limited by the fact that the same value has to be written to all locations (except for format conversion and the negation available from inverted booleans). The addition of embedded literal assignments allows a parameter to write arbitrary values, regardless of the parameter value supplied by the user.

Assignments appear at the end of a declaration, and are indicated by a `=`:

```
str_val  = <&target>,"strprop=value";              // 1
int_val  = <&target>,"intprop:0=42                 // 2
int_val2 = <&target>,"intprop:0=",<42>;            // 3
bytes    = <&target>,"bytestr[=b8:27:eb:01:23:45"; // 4
```

Lines 1, 2 and 4 are fairly obvious, but line 3 is more interesting because the value appears as an integer (cell) value. The DT compiler evaluates integer expressions at compile time, which might be convenient (particularly if macro values are used), but the cell can also contain a reference to a label:

```
// Force an LED to use a GPIO on the internal GPIO controller.
exp_led = <&led1>,"gpios:0=",<&gpio>,
          <&led1>,"gpios:4";
```

When the overlay is applied, the label will be resolved against the base DTB in the usual way. Note that it is a good idea to split multi-part parameters over multiple lines like this to make them easier to read - something that becomes more necessary with the addition of cell value assignments like this.

Bear in mind that parameters do nothing unless they are applied - a default value in a lookup table is ignored unless the parameter name is used without assigning a value.

Lookup tables

Lookup tables allow parameter input values to be transformed before they are used. They act as associative arrays, rather like switch/case statements:

```
phonetic = <&node>,"letter{a=alpha,b=bravo,c=charlie,d,e,='tango uniform'}";
bus      = <&fragment>,"target:0{0=",<&i2c0>,"1=",<&i2c1>,"}";
```

A key with no `=value` means to use the key as the value, an `=` with no key before it is the default value in the case of no match, and starting or ending the list with a comma (or an empty key=value pair anywhere) indicates that the unmatched input value should be used unaltered; otherwise, not finding a match is an error.

NOTE

The comma separator within the table string after a cell integer value is implicit - adding one explicitly creates an empty pair (see above).

NOTE

As lookup tables operate on input values and literal assignments ignore them, it's not possible to combine the two - characters after the closing `}` in the lookup declaration are treated as an error.

Overlay/fragment parameters

The DT parameter mechanism as described has a number of limitations, including no easy way to create arrays of integers and the inability to create new nodes. One way to overcome some of these limitations is to conditionally include or exclude certain fragments.

A fragment can be excluded from the final merge process (disabled) by renaming the `__overlay__` node to `__dormant__`. The parameter declaration syntax has been extended to allow the otherwise illegal zero target phandle to indicate that the following string contains operations at fragment or overlay scope. So far, four operations have been implemented:

```
+<n>    // Enable fragment <n>
-<n>    // Disable fragment <n>
=<n>    // Enable fragment <n> if the assigned parameter value is true, otherwi
se disable it
!<n>    // Enable fragment <n> if the assigned parameter value is false, otherw
ise disable it
```

Examples:

```
just_one    = <0>,"+1-2"; // Enable 1, disable 2
conditional = <0>,"=3!4"; // Enable 3, disable 4 if value is true,
                          // otherwise disable 3, enable 4.
```

The `i2c-rtc` overlay uses this technique.

Special properties

A few property names, when targeted by a parameter, get special handling. One you may have noticed already - `status` - which will convert a boolean to either `okay` for true and `disabled` for false.

Assigning to the `bootargs` property appends to it rather than overwriting it - this is how settings can be added to the kernel command line.

The `reg` property is used to specify device addresses - the location of a memory-mapped hardware block, the address on an I2C bus, etc. The names of child nodes should be qualified with their addresses in hexadecimal, using `@` as a separator:

```
bmp280@76 {
    reg = <0x77>;
    ...
};
```

When assigning to the `reg` property, the address portion of the parent node name will be replaced with the assigned value. This can be used to prevent a node name clash when using the same overlay multiple times - a technique used by the `i2c-gpio` overlay.

The `name` property is a pseudo-property - it shouldn't appear in a DT, but assigning to it causes the name of its parent node to be changed to the assigned value. Like the `reg` property, this can be used to give nodes unique names.

The overlay map file

The introduction of the Raspberry Pi 4, built around the BCM2711 SoC, brought with it many changes; some of these changes are additional interfaces, and some are modifications to (or removals of) existing interfaces. There are new overlays intended specifically for the Raspberry Pi 4 that don't make sense on older hardware, e.g. overlays that enable the new SPI, I2C and UART interfaces, but other overlays don't apply correctly even though they control features that are still relevant on the new device.

There is therefore a need for a method of tailoring an overlay to multiple platforms with differing hardware. Supporting them all in a single .dtbo file would require heavy use of hidden ("dormant") fragments and a switch to an on-demand symbol resolution mechanism so that a missing symbol that isn't needed doesn't cause a failure. A simpler solution is to add a facility to map an overlay name to one of several implementation files depending on the current platform.

The overlay map, which is rolling out with the switch to Linux 5.4, is a file that gets loaded by the firmware at bootup. It is written in DTS source format - `overlay_map.dts`, compiled to `overlay_map.dtb` and stored in the overlays directory.

This is an edited version of the current map file (see the full version):

```
/ {
    vc4-kms-v3d {
        bcm2835;
        bcm2711 = "vc4-kms-v3d-pi4";
    };

    vc4-kms-v3d-pi4 {
        bcm2711;
    };

    uart5 {
        bcm2711;
```

```
    };

    pi3-disable-bt {
        renamed = "disable-bt";
    };

    lirc-rpi {
        deprecated = "use gpio-ir";
    };
};
```

Each node has the name of an overlay that requires special handling. The properties of each node are either platform names or one of a small number of special directives. The current supported platforms are `bcm2835`, which includes all Raspberry Pis built around the BCM2835, BCM2836 and BCM2837 SoCs, and `bcm2711` for Raspberry Pi 4B.

A platform name with no value (an empty property) indicates that the current overlay is compatible with the platform; for example, `vc4-kms-v3d` is compatible with the `bcm2835` platform. A non-empty value for a platform is the name of an alternative overlay to use in place of the requested one; asking for `vc4-kms-v3d` on BCM2711 results in `vc4-kms-v3d-pi4` being loaded instead. Any platform not included in an overlay's node is not compatible with that overlay.

The second example node - `vc4-kms-v3d-pi4` - could be inferred from the content of `vc4-kms-v3d`, but that intelligence goes into the construction of the file, not its interpretation.

In the event that a platform is not listed for an overlay, one of the special directives may apply:

- The `renamed` directive indicates the new name of the overlay (which should be largely compatible with the original), but also logs a warning about the rename.
- The `deprecated` directive contains a brief explanatory error message which will be logged after the common prefix `overlay '...' is deprecated:`.

Remember: only exceptions need to be listed - the absence of a node for an overlay means that the default file should be used for all platforms.

Accessing diagnostic messages from the firmware is covered in Debugging.

The `dtoverlay` and `dtmerge` utilities have been extended to support the map file:

- `dtmerge` extracts the platform name from the compatible string in the base DTB.
- `dtoverlay` reads the compatible string from the live Device Tree at `/proc/device-tree`, but you can use the `-p` option to supply an alternate platform name (useful for dry runs on a different platform).

They both send errors, warnings and any debug output to STDERR.

Examples

Here are some examples of different types of properties, with parameters to modify them:

```
/ {
    fragment@0 {
        target-path = "/";
        __overlay__ {

            test: test_node {
                string = "hello";
                status = "disabled";
                bytes = /bits/ 8 <0x67 0x89>;
                u16s = /bits/ 16 <0xabcd 0xef01>;
                u32s = /bits/ 32 <0xfedcba98 0x76543210>;
                u64s = /bits/ 64 < 0xaaaaa5a55a5a5555 0x0000111122223333>;
                bool1; // Defaults to true
                       // bool2 defaults to false
                mac = [01 23 45 67 89 ab];
                spi = <&spi0>;
            };
        };
    };

    fragment@1 {
        target-path = "/";
        __overlay__ {
            frag1;
        };
    };

    fragment@2 {
        target-path = "/";
        __dormant__ {
            frag2;
        };
    };

    __overrides__ {
        string =      <&test>,"string";
        enable =      <&test>,"status";
        byte_0 =      <&test>,"bytes.0";
        byte_1 =      <&test>,"bytes.1";
        u16_0 =       <&test>,"u16s;0";
        u16_1 =       <&test>,"u16s;2";
        u32_0 =       <&test>,"u32s:0";
        u32_1 =       <&test>,"u32s:4";
        u64_0 =       <&test>,"u64s#0";
        u64_1 =       <&test>,"u64s#8";
        bool1 =       <&test>,"bool1!";
        bool2 =       <&test>,"bool2?";
        entofr =      <&test>,"english",
                      <&test>,"french{hello=bonjour,goodbye='au revoir',weeken
d}";
        pi_mac =      <&test>,"mac[{1=b8273bfedcba,2=b8273b987654}";
        spibus =      <&test>,"spi:0[0=",<&spi0>,"1=",<&spi1>,"2=",<&spi2>;

        only1 =       <0>,"+1-2";
        only2 =       <0>,"-1+2";
        enable1 =     <0>,"=1";
        disable2 =    <0>,"!2";
    };
};
```

For further examples, there is a large collection of overlay source files hosted in the Raspberry Pi Linux GitHub repository.

Exporting labels

The overlay handling in the firmware and the run-time overlay application using the `dtoverlay` utility treat labels defined in an overlay as being private to that overlay. This avoids the need to invent globally unique names for labels (which keeps them short), and it allows the same overlay to be used multiple times without clashing (provided some tricks are used - see Special properties).

Sometimes, however, it is very useful to be able to create a label with one overlay and use it from another. Firmware released since 14th February 2020 has the ability to declare some labels as being global - the `__exports__` node:

```
        ...
        public: ...

        __exports__ {
            public; // Export the label 'public' to the base DT
        };
    };
```

When this overlay is applied, the loader strips out all symbols except those that have been exported, in this case `public`, and rewrites the path to make it relative to the target of the fragment containing the label. Overlays loaded after this one can then refer to `&public`.

Overlay application order

Under most circumstances it shouldn't matter which order the fragments are applied, but for overlays that patch themselves (where the target of a fragment is a label in the overlay, known as an intra-overlay fragment) it becomes important. In older firmware, fragments are applied strictly in order, top to bottom. With firmware released since 14th February 2020, fragments are applied in two passes:

1. First the fragments that target other fragments are applied and hidden.
2. Then the regular fragments are applied.

This split is particularly important for runtime overlays, since step (i) occurs in the `dtoverlay` utility, and step (ii) is performed by the kernel (which can't handle intra-overlay fragments).

Using Device Trees on Raspberry Pi

DTBs, overlays and config.txt

On a Raspberry Pi it is the job of the loader (one of the `start.elf` images) to combine overlays with an appropriate base device tree, and then to pass a fully resolved Device Tree to the kernel. The base Device Trees are located alongside `start.elf` in the FAT partition

(/boot from Linux), named `bcm2711-rpi-4-b.dtb`, `bcm2710-rpi-3-b-plus.dtb`, etc. Note that some models (3A+, A, A+) will use the "b" equivalents (3B+, B, B+), respectively. This selection is automatic, and allows the same SD card image to be used in a variety of devices.

NOTE

> DT and ATAGs are mutually exclusive, and passing a DT blob to a kernel that doesn't understand it will cause a boot failure. The firmware will always try to load the DT and pass it to the kernel, since all kernels since rpi-4.4.y will not function without a DTB. You can override this by adding `device_tree=` in config.txt, which forces the use of ATAGs, which can be useful for simple "bare-metal" kernels.

[The firmware used to look for a trailer appended to kernels by the `mkknlimg` utility, but support for this has been withdrawn.]

The loader now supports builds using bcm2835_defconfig, which selects the upstreamed BCM2835 support. This configuration will cause `bcm2835-rpi-b.dtb` and `bcm2835-rpi-b-plus.dtb` to be built. If these files are copied with the kernel, then the loader will attempt to load one of those DTBs by default.

In order to manage Device Tree and overlays, the loader supports a number of `config.txt` directives:

```
dtoverlay=acme-board
dtparam=foo=bar,level=42
```

This will cause the loader to look for `overlays/acme-board.dtbo` in the firmware partition, which Raspberry Pi OS mounts on `/boot`. It will then search for parameters `foo` and `level`, and assign the indicated values to them.

The loader will also search for an attached HAT with a programmed EEPROM, and load the supporting overlay from there - either directly or by name from the "overlays" directory; this happens without any user intervention.

There are several ways to tell that the kernel is using Device Tree:

1. The "Machine model:" kernel message during bootup has a board-specific value such as "Raspberry Pi 2 Model B", rather than "BCM2709".
2. `/proc/device-tree` exists, and contains subdirectories and files that exactly mirror the nodes and properties of the DT.

With a Device Tree, the kernel will automatically search for and load modules that support the indicated enabled devices. As a result, by creating an appropriate DT overlay for a device you save users of the device from having to edit `/etc/modules`; all of the configuration goes in `config.txt`, and in the case of a HAT, even that step is unnecessary. Note, however, that layered modules such as `i2c-dev` still need to be loaded explicitly.

The flipside is that because platform devices don't get created unless requested by the DTB, it should no longer be necessary to blacklist modules that used to be loaded as a result of platform devices defined in the board support code. In fact, current Raspberry Pi OS images ship with no blacklist files (except for some WLAN devices where multiple drivers are available).

DT parameters

As described above, DT parameters are a convenient way to make small changes to a device's configuration. The current base DTBs support parameters for enabling and controlling the onboard audio, I2C, I2S and SPI interfaces without using dedicated overlays. In use, parameters look like this:

```
dtparam=audio=on,i2c_arm=on,i2c_arm_baudrate=400000,spi=on
```

NOTE

> Multiple assignments can be placed on the same line, but ensure you don't exceed the 80-character limit.

If you have an overlay that defines some parameters, they can be specified either on subsequent lines like this:

```
dtoverlay=lirc-rpi
dtparam=gpio_out_pin=16
dtparam=gpio_in_pin=17
dtparam=gpio_in_pull=down
```

or appended to the overlay line like this:

```
dtoverlay=lirc-rpi,gpio_out_pin=16,gpio_in_pin=17,gpio_in_pull=down
```

Overlay parameters are only in scope until the next overlay is loaded. In the event of a parameter with the same name being exported by both the overlay and the base, the parameter in the overlay takes precedence; for clarity, it's recommended that you avoid doing this. To expose the parameter exported by the base DTB instead, end the current overlay scope using:

```
dtoverlay=
```

Board-specific labels and parameters

Raspberry Pi boards have two I2C interfaces. These are nominally split: one for the ARM, and one for VideoCore (the "GPU"). On almost all models, `i2c1` belongs to the ARM and `i2c0` to VC, where it is used to control the camera and read the HAT EEPROM. However, there are two early revisions of the Model B that have those roles reversed.

To make it possible to use one set of overlays and parameters with all Raspberry Pis, the firmware creates some board-specific DT parameters. These are:

```
i2c/i2c_arm
i2c_vc
i2c_baudrate/i2c_arm_baudrate
i2c_vc_baudrate
```

These are aliases for `i2c0`, `i2c1`, `i2c0_baudrate`, and `i2c1_baudrate`. It is recommended that you only use `i2c_vc` and `i2c_vc_baudrate` if you really need to - for example, if you are programming a HAT EEPROM (which is better done using a software I2C bus using the `i2c-gpio` overlay). Enabling `i2c_vc` can stop the Raspberry Pi Camera or Raspberry Pi Touch Display functioning correctly.

For people writing overlays, the same aliasing has been applied to the labels on the I2C DT nodes. Thus, you should write:

```
fragment@0 {
    target = <&i2c_arm>;
    __overlay__ {
        status = "okay";
    };
};
```

Any overlays using the numeric variants will be modified to use the new aliases.

HATs and Device Tree

A Raspberry Pi HAT is an add-on board with an embedded EEPROM designed for a Raspberry Pi with a 40-pin header. The EEPROM includes any DT overlay required to enable the board (or the name of an overlay to load from the filing system), and this overlay can also expose parameters.

The HAT overlay is automatically loaded by the firmware after the base DTB, so its parameters are accessible until any other overlays are loaded, or until the overlay scope is ended using `dtoverlay=`. If for some reason you want to suppress the loading of the HAT overlay, put `dtoverlay=` before any other `dtoverlay` or `dtparam` directive.

Dynamic Device Tree

As of Linux 4.4, Raspberry Pi kernels support the dynamic loading of overlays and parameters. Compatible kernels manage a stack of overlays that are applied on top of the

base DTB. Changes are immediately reflected in `/proc/device-tree` and can cause modules to be loaded and platform devices to be created and destroyed.

The use of the word "stack" above is important - overlays can only be added and removed at the top of the stack; changing something further down the stack requires that anything on top of it must first be removed.

There are some new commands for managing overlays:

The `dtoverlay` command

`dtoverlay` is a command line utility that loads and removes overlays while the system is running, as well as listing the available overlays and displaying their help information. Use `dtoverlay -h` to get usage information:

```
Usage:
  dtoverlay <overlay> [<param>=<val>...]
                          Add an overlay (with parameters)
  dtoverlay -D [<idx>]    Dry-run (prepare overlay, but don't apply -
                          save it as dry-run.dtbo)
  dtoverlay -r [<overlay>] Remove an overlay (by name, index or the last)
  dtoverlay -R [<overlay>] Remove from an overlay (by name, index or all)
  dtoverlay -l            List active overlays/params
  dtoverlay -a            List all overlays (marking the active)
  dtoverlay -h            Show this usage message
  dtoverlay -h <overlay>  Display help on an overlay
  dtoverlay -h <overlay> <param>..  Or its parameters
    where <overlay> is the name of an overlay or 'dtparam' for dtparams
Options applicable to most variants:
    -d <dir>    Specify an alternate location for the overlays
                (defaults to /boot/overlays or /flash/overlays)
    -v          Verbose operation
```

Unlike the `config.txt` equivalent, all parameters to an overlay must be included in the same command line - the dtparam command is only for parameters of the base DTB.

Two points to note:

1. Command variants that change kernel state (adding and removing things) require root privilege, so you may need to prefix the command with `sudo`.
2. Only overlays and parameters applied at run-time can be unloaded - an overlay or parameter applied by the firmware becomes "baked in" such that it won't be listed by `dtoverlay` and can't be removed.

The `dtparam` command

`dtparam` creates and loads an overlay that has largely the same effect as using a dtparam directive in `config.txt`. In usage it is largely equivalent to `dtoverlay` with an overlay name of `-`, but there are a few differences:

1. `dtparam` will list the help information for all known parameters of the base DTB. Help on the dtparam command is still available using `dtparam -h`.

2. When indicating a parameter for removal, only index numbers can be used (not names).

3. Not all Linux subsystems respond to the addition of devices at runtime - I2C, SPI and sound devices work, but some won't.

Guidelines for writing runtime-capable overlays

This area is poorly documented, but here are some accumulated tips:

- The creation or deletion of a device object is triggered by a node being added or removed, or by the status of a node changing from disabled to enabled or vice versa. Beware - the absence of a "status" property means the node is enabled.

- Don't create a node within a fragment that will overwrite an existing node in the base DTB - the kernel will rename the new node to make it unique. If you want to change the properties of an existing node, create a fragment that targets it.

- ALSA doesn't prevent its codecs and other components from being unloaded while they are in use. Removing an overlay can cause a kernel exception if it deletes a codec that is still being used by a sound card. Experimentation found that devices are deleted in the reverse of fragment order in the overlay, so placing the node for the card after the nodes for the components allows an orderly shutdown.

Caveats

The loading of overlays at runtime is a recent addition to the kernel, and so far there is no accepted way to do this from userspace. By hiding the details of this mechanism behind commands the aim is to insulate users from changes in the event that a different kernel interface becomes standardised.

- Some overlays work better at run-time than others. Parts of the Device Tree are only used at boot time - changing them using an overlay will not have any effect.

- Applying or removing some overlays may cause unexpected behaviour, so it should be done with caution. This is one of the reasons it requires `sudo`.

- Unloading the overlay for an ALSA card can stall if something is actively using ALSA - the LXPanel volume slider plugin demonstrates this effect. To enable overlays for sound cards to be removed, the `lxpanelctl` utility has been given two new options - `alsastop` and `alsastart` - and these are called from the auxiliary scripts `dtoverlay-pre` and `dtoverlay-post` before and after overlays are loaded or unloaded, respectively.

- Removing an overlay will not cause a loaded module to be unloaded, but it may cause the reference count of some modules to drop to zero. Running `rmmod -a` twice

will cause unused modules to be unloaded.

- Overlays have to be removed in reverse order. The commands will allow you to remove an earlier one, but all the intermediate ones will be removed and re-applied, which may have unintended consequences.

- Only Device Tree nodes at the top level of the tree and children of a bus node will be probed. For nodes added at run-time there is the further limitation that the bus must register for notifications of the addition and removal of children. However, there are exceptions that break this rule and cause confusion: the kernel explicitly scans the entire tree for some device types - clocks and interrupt controller being the two main ones - in order to (for clocks) initialise them early and/or (for interrupt controllers) in a particular order. This search mechanism only happens during booting and so doesn't work for nodes added by an overlay at run-time. It is therefore recommended for overlays to place fixed-clock nodes in the root of the tree unless it is guaranteed that the overlay will not be used at run-time.

Supported overlays and parameters

As it is too time-consuming to document the individual overlays here, please refer to the README file found alongside the overlay `.dtbo` files in `/boot/overlays`. It is kept up-to-date with additions and changes.

Firmware parameters

The firmware uses the special /chosen node to pass parameters between the bootloader and/or firmware and the operating system.

`boot-mode` - 32-bit integer

The boot-mode used to load the kernel. See BOOT_ORDER.

`overlay_prefix` - string

The overlay_prefix string selected by `config.txt`.

`os_prefix` - string

The os_prefix string selected by `config.txt`.

`partition` - 32-bit integer

The partition number used during boot. If a `boot.img` ramdisk is loaded then this refers to partition that the ramdisk was loaded from rather than the partition number within the ramdisk.

`pm_rsts` - 32-bit integer

The value of the `PM_RSTS` register during boot.

`rpi-boardrev-ext` - 32-bit integer

The extended board revision code from OTP row 33.

`rpi-country-code` - 32-bit integer

The country code used used by PiWiz - Pi400 only.

`tryboot` - 32-bit integer

Set to `1` if the `tryboot` flag was set at boot.

BCM2711 bootloader properties `/chosen/bootloader`

The following properties are specific to BCM2711 SPI EEPROM bootloader.

`build_timestamp` - 32-bit integer

The UTC build time for the EEPROM bootloader.

`capabilities` - 32-bit integer

This bit-field describes the features supported by the current bootloader. This may be used to check whether a feature (e.g. USB boot) is supported before enabling it in the bootloader EEPROM config.

Bit	Feature
0	USB boot using the VLI USB host controller.
1	Network boot
2	TRYBOOT_A_B mode.
3	TRYBOOT
4	USB boot using the BCM2711 USB host controller.
5	RAM disk - boot.img
6	NVMe boot
7	Secure Boot

`update_timestamp` - 32-bit integer

The UTC update timestamp set by `rpi-eeprom-update`.

`signed_boot` - 32-bit integer

If secure-boot is enabled then this bit-field will be non-zero. The individual bits indicate the current secure-boot configuration.

Bit	Description
0	`SIGNED_BOOT` was defined in the EEPROM config file.
1	Reserved.
2	The ROM development key has been revoked. See revoke_devkey.
3	The customer public key digest has been written to OTP. See program_pubkey.
4..31	Reserved.

`version` - string

The Git version string for the bootloader.

BCM2711 USB boot properties `/chosen/bootloader/usb`

The following properties are defined if the system was booted from USB. These may be used to uniquely identify the USB boot device.

`usb-version` - 32-bit integer

The USB major protocol version (2 or 3).

`route-string` - 32-bit integer The USB route-string identifier for the device as defined by the USB 3.0 specification.

`root-hub-port-number` - 32-bit integer

The root hub port number that the boot device is connected to - possibly via other USB hubs.

`lun` - 32-bit integer

The Logical Unit Number for the mass-storage device.

NVMEM nodes

The firmware provides read-only, in-memory copies of portions of the bootloader EEPROM via the NVMEM Subsystem.

Each region appears as an NVMEM device under `/sys/bus/nvmem/devices/` with a named alias under `/sys/firmware/devicetree/base/aliases`.

Example shell script code for reading an NVMEM mode from rpi-eeprom-update

```
blconfig_alias="/sys/firmware/devicetree/base/aliases/blconfig"
blconfig_nvmem_path=""

if [ -f "${blconfig_alias}" ]; then
   blconfig_ofnode_path="/sys/firmware/devicetree/base"$(strings "${blconfig_al
ias}")""
   blconfig_ofnode_link=$(find -L /sys/bus/nvmem -samefile "${blconfig_ofnode_p
ath}" 2>/dev/null)
   if [ -e "${blconfig_ofnode_link}" ]; then
      blconfig_nvmem_path=$(dirname "${blconfig_ofnode_link}")
      fi
   fi
fi
```

`blconfig`

The `blconfig` alias refers to an NVMEM device that stores a copy of the bootloader EEPROM config file.

`blpubkey`

The `blpubkey` alias points to an NVMEM device that stores a copy of the bootloader EEPROM public key (if defined) in binary format. The rpi-bootloader-key-convert utility can be used to convert the data into PEM format for use with OpenSSL.

See also: secure-boot

Troubleshooting

Debugging

The loader will skip over missing overlays and bad parameters, but if there are serious errors, such as a missing or corrupt base DTB or a failed overlay merge, then the loader will fall back to a non-DT boot. If this happens, or if your settings don't behave as you expect, it is worth checking for warnings or errors from the loader:

```
sudo vcdbg log msg
```

Extra debugging can be enabled by adding `dtdebug=1` to `config.txt`.

You can create a human-readable representation of the current state of DT like this:

```
dtc -I fs /proc/device-tree
```

This can be useful to see the effect of merging overlays onto the underlying tree.

If kernel modules don't load as expected, check that they aren't blacklisted in `/etc/modprobe.d/raspi-blacklist.conf`; blacklisting shouldn't be necessary when using Device Tree. If that shows nothing untoward, you can also check that the module is exporting the correct aliases by searching `/lib/modules/<version>/modules.alias` for the `compatible` value. Otherwise, your driver is probably missing either:

```
.of_match_table = xxx_of_match,
```

or:

```
MODULE_DEVICE_TABLE(of, xxx_of_match);
```

Failing that, `depmod` has failed or the updated modules haven't been installed on the target filesystem.

Testing overlays using dtmerge, dtdiff and ovmerge

Alongside the `dtoverlay` and `dtparam` commands is a utility for applying an overlay to a DTB - `dtmerge`. To use it you first need to obtain your base DTB, which can be obtained in one of two ways:

a) generate it from the live DT state in `/proc/device-tree`:

```
dtc -I fs -O dtb -o base.dtb /proc/device-tree
```

This will include any overlays and parameters you have applied so far, either in `config.txt` or by loading them at runtime, which may or may not be what you want. Alternatively…

b) copy it from the source DTBs in /boot. This won't include overlays and parameters, but it also won't include any other modifications by the firmware. To allow testing of all overlays, the `dtmerge` utility will create some of the board-specific aliases ("i2c_arm", etc.), but this means that the result of a merge will include more differences from the original DTB than you might expect. The solution to this is to use dtmerge to make the copy:

```
dtmerge /boot/bcm2710-rpi-3-b.dtb base.dtb -
```

(the - indicates an absent overlay name).

You can now try applying an overlay or parameter:

```
dtmerge base.dtb merged.dtb - sd_overclock=62
dtdiff base.dtb merged.dtb
```

which will return:

```
--- /dev/fd/63  2016-05-16 14:48:26.396024813 +0100
+++ /dev/fd/62  2016-05-16 14:48:26.396024813 +0100
@@ -594,7 +594,7 @@
                };

                sdhost@7e202000 {
-                       brcm,overclock-50 = <0x0>;
+                       brcm,overclock-50 = <0x3e>;
                        brcm,pio-limit = <0x1>;
                        bus-width = <0x4>;
                        clocks = <0x8>;
```

You can also compare different overlays or parameters.

```
dtmerge base.dtb merged1.dtb /boot/overlays/spi1-1cs.dtbo
dtmerge base.dtb merged2.dtb /boot/overlays/spi1-2cs.dtbo
dtdiff merged1.dtb merged2.dtb
```

to get:

```
--- /dev/fd/63  2016-05-16 14:18:56.189634286 +0100
+++ /dev/fd/62  2016-05-16 14:18:56.189634286 +0100
@@ -453,7 +453,7 @@

                        spi1_cs_pins {
                                brcm,function = <0x1>;
-                               brcm,pins = <0x12>;
+                               brcm,pins = <0x12 0x11>;
                                phandle = <0x3e>;
                        };

@@ -725,7 +725,7 @@
                        #size-cells = <0x0>;
                        clocks = <0x13 0x1>;
                        compatible = "brcm,bcm2835-aux-spi";
-                       cs-gpios = <0xc 0x12 0x1>;
+                       cs-gpios = <0xc 0x12 0x1 0xc 0x11 0x1>;
                        interrupts = <0x1 0x1d>;
                        linux,phandle = <0x30>;
                        phandle = <0x30>;
@@ -743,6 +743,16 @@
                                spi-max-frequency = <0x7a120>;
                                status = "okay";
                        };
+
+                       spidev@1 {
+                               #address-cells = <0x1>;
+                               #size-cells = <0x0>;
+                               compatible = "spidev";
+                               phandle = <0x41>;
+                               reg = <0x1>;
+                               spi-max-frequency = <0x7a120>;
+                               status = "okay";
+                       };
                };

                spi@7e2150C0 {
```

The Utils repo includes another DT utility - `ovmerge`. Unlike `dtmerge`, `ovmerge` combines file and applies overlays in source form. Because the overlay is never compiled, labels are preserved and the result is usually more readable. It also has a number of other tricks, such as the ability to list the order of file inclusion.

Forcing a specific Device Tree

If you have very specific needs that aren't supported by the default DTBs, or if you just want to experiment with writing your own DTs, you can tell the loader to load an alternate DTB file like this:

```
device_tree=my-pi.dtb
```

Disabling Device Tree usage

Since the switch to the 4.4 kernel and the use of more upstream drivers, Device Tree usage is required in Raspberry Pi Linux kernels. However, for bare metal and other OSs, the method of disabling DT usage is to add:

```
device_tree=
```

to `config.txt`.

Shortcuts and syntax variants

The loader understands a few shortcuts:

```
dtparam=i2c_arm=on
dtparam=i2s=on
```

can be shortened to:

```
dtparam=i2c,i2s
```

(`i2c` is an alias of `i2c_arm`, and the `=on` is assumed). It also still accepts the long-form versions: `device_tree_overlay` and `device_tree_param`.

Other DT commands available in config.txt

`device_tree_address` This is used to override the address where the firmware loads the device tree (not dt-blob). By default the firmware will choose a suitable place.

`device_tree_end` This sets an (exclusive) limit to the loaded device tree. By default the device tree can grow to the end of usable memory, which is almost certainly what is required.

`dtdebug` If non-zero, turn on some extra logging for the firmware's device tree processing.

`enable_uart` Enable the primary/console UART (ttyS0 on a Raspberry Pi 3, 4, 400, Zero W and Zero 2 W, ttyAMA0 otherwise - unless swapped with an overlay such as miniuart-bt). If the primary UART is ttyAMA0 then `enable_uart` defaults to 1 (enabled), otherwise it defaults to 0 (disabled). This is because it is necessary to stop the core frequency from changing which would make ttyS0 unusable, so `enable_uart=1` implies `core_freq=250` (unless `force_turbo=1`). In some cases this is a performance hit, so it is off by default.

`overlay_prefix` Specifies a subdirectory/prefix from which to load overlays - defaults to "overlays/". Note the trailing "/". If desired you can add something after the final "/" to add a prefix to each file, although this is not likely to be needed.

Further ports can be controlled by the DT, for more details see section 3.

Further help

If you've read through this document and not found the answer to a Device Tree problem, there is help available. The author can usually be found on Raspberry Pi forums, particularly the Device Tree forum.

The Kernel Command Line

Edit this on GitHub

The Linux kernel accepts a command line of parameters during boot. On the Raspberry Pi, this command line is defined in a file in the boot partition, called cmdline.txt. This is a simple text file that can be edited using any text editor, e.g. Nano.

```
sudo nano /boot/cmdline.txt
```

NOTE

> We have to use `sudo` to edit anything in the boot partition, and all parameters in `cmdline.txt` must be on the same line (no carriage returns).

The command line that was passed to the kernel at boot time can be displayed using `cat /proc/cmdline`. It will not be exactly the same as that in `cmdline.txt` as the firmware can make changes to it prior to launching the kernel.

Command Line Options

There are many kernel command line parameters, some of which are defined by the kernel. Others are defined by code that the kernel may be using, such as the Plymouth splash screen system.

Standard Entries

- console: defines the serial console. There are usually two entries:
  - `console=serial0,115200`
  - `console=tty1`
- root: defines the location of the root filesystem, e.g. `root=/dev/mmcblk0p2` means multimedia card block 0 partition 2.
- rootfstype: defines what type of filesystem the rootfs uses, e.g. `rootfstype=ext4`
- quiet: sets the default kernel log level to `KERN_WARNING`, which suppresses all but very serious log messages during boot.

Display Entries in FKMS and KMS modes

The firmware automatically adds a preferred resolution and overscan settings via an entry such as:

```
video=HDMI-A-1:1920x1080M@60,margin_left=0,margin_right=0,margin_top=0,margin_b
ottom=0
```

This default entry can be modified by duplicating the entry above manually in /boot/cmdline.txt and making required changes to the margin parameters. In addition, it is possible to add rotation and reflect parameters as documented in the standard Linux framebuffer documentation. By default the `margin_*` options are set from the `overscan` entries in config.txt, if present. The firmware can be prevented from making any KMS specific changes to the command line by adding `disable_fw_kms_setup=1` to `config.txt`

An example entry may be as follows:

```
video=HDMI-A-1:1920x1080M@60,margin_left=0,margin_right=0,margin_top=0,margin_b
ottom=0,rotate=90,reflect_x`
```

Possible options for the display type, the first part of the `video=` entry, are as follows:

video option	Display
HDMI-A-1	HDMI 1 (HDMI 0 on silkscreen of Raspberry Pi 4B, HDMI on single HDMI boards)

---

video option	Display
HDMI-A-2	HDMI 2 (HDMI 1 on silkscreen of Raspberry Pi 4B)
DSI-1	DSI or DPI
Composite-1	Composite

Other Entries (not exhaustive)

- splash: tells the boot to use a splash screen via the Plymouth module.
- plymouth.ignore-serial-consoles: normally if the Plymouth module is enabled it will prevent boot messages from appearing on any serial console which may be present. This flag tells Plymouth to ignore all serial consoles, making boot messages visible again, as they would be if Plymouth was not running.
- dwc_otg.lpm_enable=0: turns off Link Power Management (LPM) in the dwc_otg driver; the dwc_otg driver is the driver for the USB controller built into the processor used on Raspberry Pi computers.

  NOTE

  On Raspberry Pi 4 this controller is disabled by default, and is only connected to the USB type C power input connector; the USB type A ports on Raspberry Pi 4 are driven by a separate USB controller which is not affected by this setting.

- dwc_otg.speed: sets the speed of the USB controller built into the processor on Raspberry Pi computers. `dwc_otg.speed=1` will set it to full speed (USB 1.0), which is slower than high speed (USB 2.0). This option should not be set except during troubleshooting of problems with USB devices.
- smsc95xx.turbo_mode: enables/disables the wired networking driver turbo mode. `smsc95xx.turbo_mode=N` turns turbo mode off.
- usbhid.mousepoll: specifies the mouse polling interval. If you have problems with a slow or erratic wireless mouse, setting this to 0 might help: `usbhid.mousepoll=0`.

Configuring UARTs

Edit this on GitHub

There are two types of UART available on the Raspberry Pi - PL011 and mini UART. The PL011 is a capable, broadly 16550-compatible UART, while the mini UART has a reduced feature set.

All UARTs on the Raspberry Pi are 3.3V only - damage will occur if they are connected to 5V systems. An adaptor can be used to connect to 5V systems. Alternatively, low-cost USB to 3.3V serial adaptors are available from various third parties.

Raspberry Pi Zero, 1, 2 and 3

The Raspberry Pi Zero, 1, 2, and 3 each contain two UARTs as follows:

Name	Type
UART0	PL011
UART1	mini UART

Raspberry Pi 4 and 400

The Raspberry Pi 4B and 400 have an additional four PL011s, which are disabled by default:

Name	Type
UART0	PL011
UART1	mini UART
UART2	PL011
UART3	PL011
UART4	PL011
UART5	PL011

CM1, CM3, CM3+ and CM4

The first generation Compute Module, together with Compute Module 3 and Compute Module 3+ each have two UARTs, while Compute Module 4 has six UARTs as described above.

On all models of Compute Module, the UARTs are disabled by default and can be explicitly enabled using a device tree overlay. You may also specify which GPIO pins to use, for example:

```
dtoverlay=uart1,txd1_pin=32,rxd1_pin=33
```

Primary UART

On the Raspberry Pi, one UART is selected to be present on GPIO 14 (transmit) and 15 (receive) - this is the primary UART. By default, this will also be the UART on which a Linux console may be present. Note that GPIO 14 is pin 8 on the GPIO header, while GPIO 15 is pin 10.

Secondary UART

The secondary UART is not normally present on the GPIO connector. By default, the secondary UART is connected to the Bluetooth side of the combined wireless LAN/Bluetooth controller, on models which contain this controller.

Primary and Secondary UART

The following table summarises the assignment of the first two UARTs:

Model	first PL011 (UART0)	mini UART
Raspberry Pi Zero	primary	secondary
Raspberry Pi Zero W	secondary (Bluetooth)	primary
Raspberry Pi 1	primary	secondary
Raspberry Pi 2	primary	secondary
Raspberry Pi 3	secondary (Bluetooth)	primary
Compute Module 3 & 3+	primary	secondary
Raspberry Pi 4	secondary (Bluetooth)	primary

NOTE

The mini UART is disabled by default, whether it is designated primary or secondary UART.

Linux devices on Raspberry Pi OS:

Linux device	Description
`/dev/ttyS0`	mini UART
`/dev/ttyAMA0`	first PL011 (UART0)
`/dev/serial0`	primary UART
`/dev/serial1`	secondary UART

NOTE

`/dev/serial0` and `/dev/serial1` are symbolic links which point to either `/dev/ttyS0` or `/dev/ttyAMA0`.

Mini-UART and CPU Core Frequency

In order to use the mini UART, you need to configure the Raspberry Pi to use a fixed VPU core clock frequency. This is because the mini UART clock is linked to the VPU core clock,

so that when the core clock frequency changes, the UART baud rate will also change. The `enable_uart` and `core_freq` settings can be added to `config.txt` to change the behaviour of the mini UART. The following table summarises the possible combinations:

Mini UART set to	core clock	Result
primary UART	variable	mini UART disabled
primary UART	fixed by setting `enable_uart=1`	mini UART enabled, core clock fixed to 250MHz, or if `force_turbo=1` is set, the VPU turbo frequency
secondary UART	variable	mini UART disabled
secondary UART	fixed by setting `core_freq=250`	mini UART enabled

The default state of the `enable_uart` flag depends on which UART is the primary UART:

Primary UART	Default state of enable_uart flag
mini UART	0
first PL011 (UART0)	1

Disabling the Linux Serial Console

By default, the primary UART is assigned to the Linux console. If you wish to use the primary UART for other purposes, you must reconfigure Raspberry Pi OS. This can be done by using raspi-config:

1. Start raspi-config: `sudo raspi-config`.
2. Select option 3 - Interface Options.
3. Select option P6 - Serial Port.
4. At the prompt `Would you like a login shell to be accessible over serial?` answer 'No'
5. At the prompt `Would you like the serial port hardware to be enabled?` answer 'Yes'
6. Exit raspi-config and reboot the Raspberry Pi for changes to take effect.

Enabling Early Console for Linux

Although the Linux kernel starts the UARTs relatively early in the boot process, it is still long after some critical bits of infrastructure have been set up. A failure in those early stages

can be hard to diagnose without access to the kernel log messages from that time. To enable `earlycon` support for one of the UARTs, add one of the following options to `cmdline.txt`, depending on which UART is the primary:

For Raspberry Pi 4, 400 and Compute Module 4:

```
earlycon=uart8250,mmio32,0xfe215040
earlycon=pl011,mmio32,0xfe201000
```

For Raspberry Pi 2, Pi 3 and Compute Module 3:

```
earlycon=uart8250,mmio32,0x3f215040
earlycon=pl011,mmio32,0x3f201000
```

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

```
earlycon=uart8250,mmio32,0x20215040
earlycon=pl011,mmio32,0x20201000
```

The baudrate defaults to 115200bps.

NOTE

> Selecting the wrong early console can prevent the Raspberry Pi from booting.

UARTs and Device Tree

Various UART Device Tree overlay definitions can be found in the kernel GitHub tree. The two most useful overlays are `disable-bt` and `miniuart-bt`.

`disable-bt` disables the Bluetooth device and makes the first PL011 (UART0) the primary UART. You must also disable the system service that initialises the modem, so it does not connect to the UART, using `sudo systemctl disable hciuart`.

`miniuart-bt` switches the Bluetooth function to use the mini UART, and makes the first PL011 (UART0) the primary UART. Note that this may reduce the maximum usable baud rate (see mini UART limitations below). You must also set the VPU core clock to a fixed frequency using either `force_turbo=1` or `core_freq=250`.

The overlays `uart2`, `uart3`, `uart4`, and `uart5` are used to enable the four additional UARTs on the Raspberry Pi 4. There are other UART-specific overlays in the folder. Refer to `/boot/overlays/README` for details on Device Tree overlays, or run `dtoverlay -h overlay-name` for descriptions and usage information.

You add a line to the `config.txt` file to apply a Device Tree overlay. Note that the `-overlay.dts` part of the filename is removed. For example:

```
dtoverlay=disable-bt
```

PL011 and mini-UART

There are some differences between PL011 UARTs and mini-UART.

The mini-UART has smaller FIFOs. Combined with the lack of flow control, this makes it more prone to losing characters at higher baudrates. It is also generally less capable than a PL011, mainly due to its baud rate link to the VPU clock speed.

The particular deficiencies of the mini UART compared to a PL011 are :

- No break detection
- No framing errors detection
- No parity bit
- No receive timeout interrupt
- No DCD, DSR, DTR or RI signals

Further documentation on the mini UART can be found in the SoC peripherals document.

Firmware Warning Icons

Edit this on GitHub

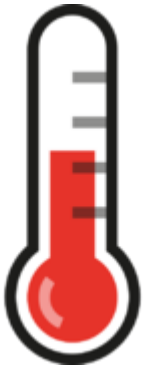
Under certain circumstances, the Raspberry Pi firmware will display a warning icon on the display, to indicate an issue. There are currently three icons that can be displayed.

Undervoltage Warning

If the power supply to the Raspberry Pi drops below 4.63V (±5%), the following icon is displayed.

Over Temperature Warning (80-85°C)

If the temperature of the SoC is between 80C and 85C, the following icon is displayed. The ARM core(s) will be throttled back in an attempt to reduce the core temperature.



Over Temperature Warning (over 85°C)

If the temperature of the SoC is over 85°C, the following icon is displayed. The ARM core(s) and the GPU will be throttled back in an attempt to reduce the core temperature.

LED Warning Flash Codes

Edit this on GitHub

If a Raspberry Pi fails to boot for some reason, or has to shut down, in many cases an LED will be flashed a specific number of times to indicate what happened. The LED will blink for a number of long flashes (0 or more), then short flashes, to indicate the exact status. In most cases, the pattern will repeat after a 2 second gap.

Long flashes	Short flashes	Status
0	3	Generic failure to boot
0	4	start*.elf not found
0	7	Kernel image not found
0	8	SDRAM failure
0	9	Insufficient SDRAM
0	10	In HALT state
2	1	Partition not FAT
2	2	Failed to read from partition
2	3	Extended partition not FAT
2	4	File signature/hash mismatch - Pi 4
3	1	SPI EEPROM error - Pi 4
3	2	SPI EEPROM is write protected - Pi 4
3	3	I2C error - Pi 4
3	4	Secure-boot configuration is not valid
4	4	Unsupported board type
4	5	Fatal firmware error
4	6	Power failure type A
4	7	Power failure type B

Securing your Raspberry Pi

Edit this on GitHub

The security of your Raspberry Pi is important. Gaps in security leave your Raspberry Pi open to hackers who can then use it without your permission.

What level of security you need depends on how you wish to use your Raspberry Pi. For example, if you are simply using your Raspberry Pi on your home network, behind a router with a firewall, then it is already quite secure by default.

However, if you wish to expose your Raspberry Pi directly to the internet, either with a direct connection (unlikely) or by letting certain protocols through your router firewall (e.g. SSH), then you need to make some basic security changes.

Even if you are hidden behind a firewall, it is sensible to take security seriously. This documentation will describe some ways of improving the security of your Raspberry Pi. Please note, though, that it is not exhaustive.

Change the Default Password

The default username and password is used for every single Raspberry Pi running Raspberry Pi OS. So, if you can get access to a Raspberry Pi, and these settings have not been changed, you have `root` access to that Raspberry Pi.

So the first thing to do is change the password. This can be done via the `raspi-config` application, or from the command line.

```
sudo raspi-config
```

Select option 2, and follow the instructions to change the password.

However, all `raspi-config` does is start up the command line `passwd` application, which you can do from the command line. So instead you can type in your new password and confirm it.

```
passwd
```

Changing your Username

You can, of course, make your Raspberry Pi even more secure by also changing your username. All Raspberry Pis come with the default username `pi`, so changing this will immediately make your Raspberry Pi more secure.

To add a new user, enter:

```
sudo adduser alice
```

You will be prompted to create a password for the new user.

The new user will have a home directory at `/home/alice/`.

To add them to the `sudo` group to give them `sudo` permissions as well as all of the other necessary permissions:

```
sudo usermod -a -G adm,dialout,cdrom,sudo,audio,video,plugdev,games,users,inpu
t,netdev,gpio,i2c,spi alice
```

You can check your permissions are in place (i.e. you can use `sudo`) by trying the following:

```
sudo su - alice
```

If it runs successfully, then you can be sure that the new account is in the `sudo` group.

Once you have confirmed that the new account is working, you can delete the `pi` user. In order to do this, you'll need to first change the autologin user to your new user `alice`, with the following:

```
sudo raspi-config
```

Select option 1, S5 `Boot / Auto login`, and say yes to reboot. Please note that with the current Raspberry Pi OS distribution, there are some aspects that require the `pi` user to be present. If you are unsure whether you will be affected by this, then leave the `pi` user in place. Work is being done to reduce the dependency on the `pi` user.

To delete the `pi` user, type the following:

```
sudo deluser pi
```

This command will delete the `pi` user but will leave the `/home/pi` folder. If necessary, you can use the command below to remove the home folder for the `pi` user at the same time. Note the data in this folder will be permanently deleted, so make sure any required data is stored elsewhere.

```
sudo deluser -remove-home pi
```

This command will result in a warning that the group `pi` has no more members. The `deluser` command removes both the `pi` user and the `pi` group though, so the warning can be safely ignored.

Make sudo Require a Password

Placing `sudo` in front of a command runs it as a superuser, and by default, that does not need a password. In general, this is not a problem. However, if your Raspberry Pi is exposed to the internet and somehow becomes exploited (perhaps via a webpage exploit for example), the attacker will be able to change things that require superuser credentials, unless you have set `sudo` to require a password.

To force `sudo` to require a password, enter:

```
sudo visudo /etc/sudoers.d/010_pi-nopasswd
```

and change the `pi` entry (or whichever usernames have superuser rights) to:

```
pi ALL=(ALL) PASSWD: ALL
```

Then save the file: it will be checked for any syntax errors. If no errors were detected, the file will be saved and you will be returned to the shell prompt. If errors were detected, you will be asked 'what now?' Press the 'enter' key on your keyboard: this will bring up a list of options. You will probably want to use 'e' for '(e)dit sudoers file again', so you can edit the file and fix the problem.

NOTE

> Choosing option 'Q' will save the file with any syntax errors still in place, which makes it impossible for *any* user to use the sudo command.

Updating Raspberry Pi OS

An up-to-date distribution contains all the latest security fixes, so you should go ahead and update your version of Raspberry Pi OS to the latest version.

If you are using SSH to connect to your Raspberry Pi, it can be worthwhile to add a `cron` job that specifically updates the ssh-server. The following command, perhaps as a daily cron job, will ensure you have the latest SSH security fixes promptly, independent of your normal update process.

```
apt install openssh-server
```

Improving SSH Security

SSH is a common way of accessing a Raspberry Pi remotely. By default, logging in with SSH requires a username/password pair, and there are ways to make this more secure. An even more secure method is to use key based authentication.

Improving username/password security

The most important thing to do is ensure you have a very robust password. If your Raspberry Pi is exposed to the internet, the password needs to be very secure. This will help to avoid dictionary attacks or the like.

You can also **allow** or **deny** specific users by altering the `sshd` configuration.

```
sudo nano /etc/ssh/sshd_config
```

Add, edit, or append to the end of the file the following line, which contains the usernames you wish to allow to log in:

```
AllowUsers alice bob
```

You can also use `DenyUsers` to specifically stop some usernames from logging in:

```
DenyUsers jane john
```

After the change you will need to restart the `sshd` service using `sudo systemctl restart ssh` or reboot so the changes take effect.

Using key-based authentication.

Key pairs are two cryptographically secure keys. One is private, and one is public. They can be used to authenticate a client to an SSH server (in this case the Raspberry Pi).

The client generates two keys, which are cryptographically linked to each other. The private key should never be released, but the public key can be freely shared. The SSH server takes a copy of the public key, and, when a link is requested, uses this key to send the client a challenge message, which the client will encrypt using the private key. If the server can use the public key to decrypt this message back to the original challenge message, then the identity of the client can be confirmed.

Generating a key pair in Linux is done using the `ssh-keygen` command on the **client**; the keys are stored by default in the `.ssh` folder in the user's home directory. The private key will be called `id_rsa` and the associated public key will be called `id_rsa.pub`. The key will be 2048 bits long: breaking the encryption on a key of that length would take an extremely long time, so it is very secure. You can make longer keys if the situation demands it. Note that you should only do the generation process once: if repeated, it will overwrite any previous generated keys. Anything relying on those old keys will need to be updated to the new keys.

You will be prompted for a passphrase during key generation: this is an extra level of security. For the moment, leave this blank.

The public key now needs to be moved on to the server: see Copy your public key to your Raspberry Pi.

Finally, we need to disable password logins, so that all authentication is done by the key pairs.

```
sudo nano /etc/ssh/sshd_config
```

There are three lines that need to be changed to `no`, if they are not set that way already:

```
ChallengeResponseAuthentication no
PasswordAuthentication no
UsePAM no
```

Save the file and either restart the ssh system with `sudo service ssh reload` or reboot.

Install a Firewall

There are many firewall solutions available for Linux. Most use the underlying iptables project to provide packet filtering. This project sits over the Linux netfiltering system. `iptables` is installed by default on Raspberry Pi OS, but is not set up. Setting it up can be a complicated task, and one project that provides a simpler interface than `iptables` is ufw, which stands for 'Uncomplicated Fire Wall'. This is the default firewall tool in Ubuntu, and can be easily installed on your Raspberry Pi:

```
sudo apt install ufw
```

`ufw` is a fairly straightforward command line tool, although there are some GUIs available for it. This document will describe a few of the basic command line options. Note that `ufw` needs to be run with superuser privileges, so all commands are preceded with `sudo`. It is also possible to use the option `--dry-run` any `ufw` commands, which indicates the results of the command without actually making any changes.

To enable the firewall, which will also ensure it starts up on boot, use:

```
sudo ufw enable
```

To disable the firewall, and disable start up on boot, use:

```
sudo ufw disable
```

Allow a particular port to have access (we have used port 22 in our example):

```
sudo ufw allow 22
```

Denying access on a port is also very simple (again, we have used port 22 as an example):

```
sudo ufw deny 22
```

You can also specify which service you are allowing or denying on a port. In this example, we are denying tcp on port 22:

```
sudo ufw deny 22/tcp
```

You can specify the service even if you do not know which port it uses. This example allows the ssh service access through the firewall:

```
sudo ufw allow ssh
```

The status command lists all current settings for the firewall:

```
sudo ufw status
```

The rules can be quite complicated, allowing specific IP addresses to be blocked, specifying in which direction traffic is allowed, or limiting the number of attempts to connect, for example to help defeat a Denial of Service (DoS) attack. You can also specify the device rules are to be applied to (e.g. eth0, wlan0). Please refer to the `ufw` man page (`man ufw`) for full details, but here are some examples of more sophisticated commands.

Limit login attempts on ssh port using tcp: this denies connection if an IP address has attempted to connect six or more times in the last 30 seconds:

```
sudo ufw limit ssh/tcp
```

Deny access to port 30 from IP address 192.168.2.1

```
sudo ufw deny from 192.168.2.1 port 30
```

Installing `fail2ban`

If you are using your Raspberry Pi as some sort of server, for example an `ssh` or a webserver, your firewall will have deliberate 'holes' in it to let the server traffic through. In

these cases, Fail2ban can be useful. Fail2ban, written in Python, is a scanner that examines the log files produced by the Raspberry Pi, and checks them for suspicious activity. It catches things like multiple brute-force attempts to log in, and can inform any installed firewall to stop further login attempts from suspicious IP addresses. It saves you having to manually check log files for intrusion attempts and then update the firewall (via `iptables`) to prevent them.

Install `fail2ban` using the following command:

```
sudo apt install fail2ban
```

On installation, Fail2ban creates a folder `/etc/fail2ban` in which there is a configuration file called `jail.conf`. This needs to be copied to `jail.local` to enable it. Inside this configuration file are a set of default options, together with options for checking specific services for abnormalities. Do the following to examine/change the rules that are used for `ssh`:

```
sudo cp /etc/fail2ban/jail.conf /etc/fail2ban/jail.local
sudo nano /etc/fail2ban/jail.local
```

Add the following section to the `jail.local` file. On some versions of fail2ban this section may already exist, so update this pre-existing section if it is there.

```
[ssh]
enabled  = true
port     = ssh
filter   = sshd
logpath  = /var/log/auth.log
maxretry = 6
```

As you can see, this section is named ssh, is enabled, examines the ssh port, filters using the `sshd` parameters, parses the `/var/log/auth.log` for malicious activity, and allows six retries before the detection threshold is reached. Checking the default section, we can see that the default banning action is:

```
# Default banning action (e.g. iptables, iptables-new,
# iptables-multiport, shorewall, etc) It is used to define
# action_* variables. Can be overridden globally or per
# section within jail.local file
banaction = iptables-multiport
```

`iptables-multiport` means that the Fail2ban system will run the `/etc/fail2ban/action.d/iptables-multiport.conf` file when the detection threshold is reached. There are a number of different action configuration files that can be used. Multiport bans all access on all ports.

If you want to permanently ban an IP address after three failed attempts, you can change the maxretry value in the `[ssh]` section, and set the bantime to a negative number:

```
[ssh]
enabled  = true
port     = ssh
filter   = sshd
logpath  = /var/log/auth.log
maxretry = 3
bantime = -1
```

Configuring Screen Blanking

Edit this on GitHub

You can configure your Raspberry Pi to use a screen saver or to blank the screen.

On Console

When running without a graphical desktop, Raspberry Pi OS will blank the screen after 10 minutes without user input, e.g. mouse movement or key presses.

The current setting, in seconds, can be displayed using:

```
cat /sys/module/kernel/parameters/consoleblank
```

To change the `consoleblank` setting, edit the kernel command line:

```
sudo nano /boot/cmdline.txt
```

The file `/boot/cmdline.txt` contains a single line of text. Add `consoleblank=n` to have the console blank after `n` seconds of inactivity. For example `consoleblank=300` will cause the console to blank after 300 seconds, 5 minutes, of inactivity. Make sure that you add your `consoleblank` option to the single line of text already in the `cmdline.txt` file. To disable screen blanking, set `consoleblank=0`.

You can also use the `raspi-config` tool to disable screen blanking. Note that the screen blanking setting in `raspi-config` also controls screen blanking when the graphical desktop is running.

On the Desktop

Raspberry Pi OS will blank the graphical desktop after 10 minutes without user input. You can disable this by changing the 'Screen Blanking' option in the Raspberry Pi Configuration

tool, which is available on the Preferences menu. Note that the 'Screen Blanking' option also controls screen blanking when the graphical desktop is not running.

There is also a graphical screensaver available, which can be installed as follows:

```
sudo apt install xscreensaver
```

This may take a few minutes.

Once this has been installed, you can find the Screensaver application on the Preferences menu: it provides many options for setting up the screensaver, including disabling it completely.

Switching off HDMI

If you want to switch off the video display entirely, you can use the vcgencmd command,

```
vcgencmd display_power 0
```

Video will not come back on until you reboot or switch it back on:

```
vcgencmd display_power 1
```

The `boot` Folder

Edit this on GitHub

In a basic Raspberry Pi OS install, the boot files are stored on the first partition of the SD card, which is formatted with the FAT file system. This means that it can be read on Windows, macOS, and Linux devices.

When the Raspberry Pi is powered on, it loads various files from the boot partition/folder in order to start up the various processors, then it boots the Linux kernel.

Once Linux has booted, the boot partition is mounted as `/boot`.

Boot Folder Contents

bootcode.bin

This is the bootloader, which is loaded by the SoC on boot, does some very basic setup, and then loads one of the `start*.elf` files. `bootcode.bin` is not used on the Raspberry Pi 4,

because it has been replaced by boot code in the onboard EEPROM.

start.elf, start_x.elf, start_db.elf, start_cd.elf, start4.elf, start4x.elf, start4cd.elf, start4db.elf

These are binary blobs (firmware) that are loaded on to the VideoCore in the SoC, which then take over the boot process. `start.elf` is the basic firmware, `start_x.elf` includes camera drivers and codec, `start_db.elf` is a debug version of the firmware, and `start_cd.elf` is a cut-down version with no support hardware blocks like codecs and 3D, and for use when `gpu_mem=16` is specified in `config.txt`. More information on how to use these can be found in the `config.txt` section.

`start4.elf`, `start4x.elf`, `start4cd.elf`, and `start4db.elf` are firmware files specific to the Raspberry Pi 4.

fixup*.dat

These are linker files and are matched pairs with the `start*.elf` files listed in the previous section.

cmdline.txt

The kernel command line passed in to the kernel when it boots.

config.txt

Contains many configuration parameters for setting up the Raspberry Pi. See the `config.txt` section.

issue.txt

Some text-based housekeeping information containing the date and git commit ID of the distribution.

ssh or ssh.txt

When this file is present, SSH will be enabled on boot. The contents don't matter, it can be empty. SSH is otherwise disabled by default.

wpa_supplicant.conf

This is the file to configure wireless network settings (if the hardware is capable of it). Edit the country code and the network part to fit your case. More information on how to use this file can be found in the `wireless/headless` section.

Device Tree files

There are various Device Tree blob files, which have the extension `.dtb`. These contain the hardware definitions of the various models of Raspberry Pi, and are used on boot to set up the kernel according to which Raspberry Pi model is detected.

Kernel Files

The boot folder will contain various kernel image files, used for the different Raspberry Pi models:

Filename	Processor	Raspberry Pi model	Notes
kernel.img	BCM2835	Pi Zero, Pi 1	
kernel7.img	BCM2836, BCM2837	Pi Zero 2 W, Pi 2, Pi 3	Later Pi 2 uses the BCM2837
kernel7l.img	BCM2711	Pi 4, Pi 400	Large Physical Address Extension (LPAE)
kernel8.img	BCM2837, BCM2711	Pi Zero 2 W, Pi 2, Pi 3, Pi 4, Pi 400	64-bit kernel. Raspberry Pi 2 with BCM2836 does not support 64-bit kernels.

NOTE

The architecture reported by `lscpu` is `armv7l` for systems running a 32-bit kernel (i.e. everything except `kernel8.img`), and `aarch64` for systems running a 64-bit kernel. The `l` in the `armv7l` case refers to the architecture being little-endian, not `LPAE` as is indicated by the `l` in the `kernel7l.img` filename.

The Overlays Folder

The `overlays` sub-folder contains Device Tree overlays. These are used to configure various hardware devices that may be attached to the system, for example the Raspberry Pi Touch Display or third-party sound boards. These overlays are selected using entries in `config.txt` — see 'Device Trees, overlays and parameters, part 2' for more info.