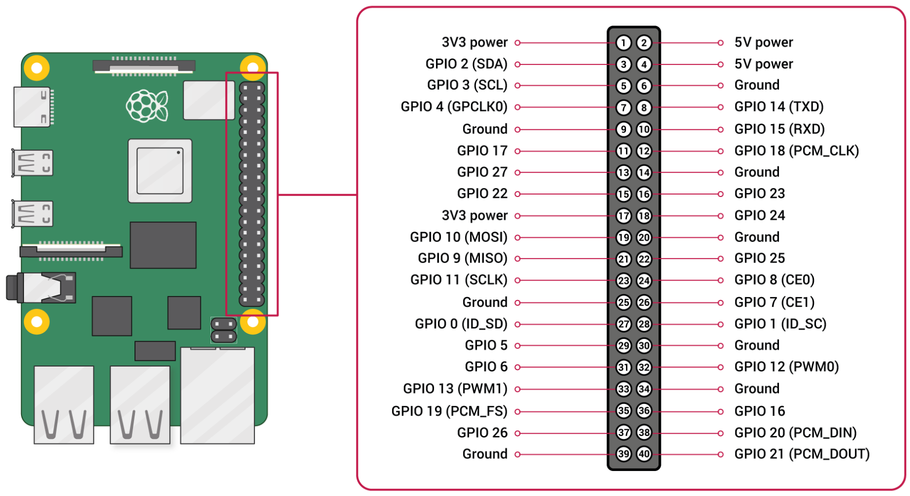
Raspberry Pi hardware

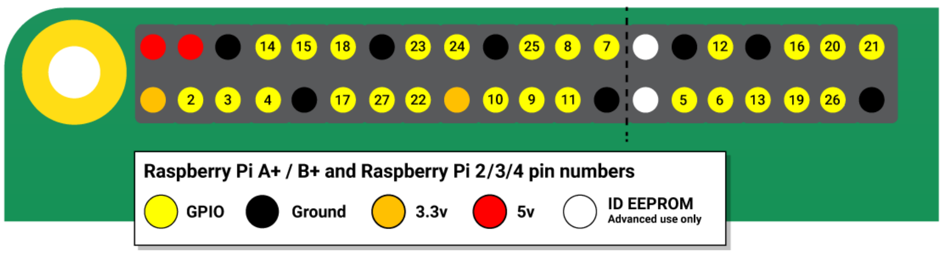
GPIO and the 40-pin Header

Edit this on GitHub

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards (unpopulated on Raspberry Pi Zero, Raspberry Pi Zero W and Raspberry Pi Zero 2 W). Prior to the Raspberry Pi 1 Model B+ (2014), boards comprised a shorter 26-pin header. The GPIO header on all boards (including the Raspberry Pi 400) have a 0.1" (2.54mm) pin pitch.


3V3 power 1 2 5V power
GPIO 2 (SDA) 3 4 5V power
GPIO 3 (SCL) 5 6 Ground
GPIO 4 (GPCLK0) 7 8 GPIO 14 (TXD)
Ground 9 10 GPIO 15 (RXD)
GPIO 17 11 12 GPIO 18 (PCM_CLK)
GPIO 27 13 14 Ground
GPIO 22 15 16 GPIO 23
3V3 power 17 18 GPIO 24
GPIO 10 (MOSI) 19 20 Ground
GPIO 9 (MISO) 21 22 GPIO 25
GPIO 11 (SCLK) 23 24 GPIO 8 (CE0)
Ground 25 26 GPIO 7 (CE1)
GPIO 0 (ID_SD) 27 28 GPIO 1 (ID_SC)
GPIO 5 29 30 Ground
GPIO 6 31 32 GPIO 12 (PWM0)
GPIO 13 (PWM1) 33 34 Ground
GPIO 19 (PCM_FS) 35 36 GPIO 16
GPIO 26 37 38 GPIO 20 (PCM_DIN)
Ground 39 40 GPIO 21 (PCM_DOUT)

Any of the GPIO pins can be designated (in software) as an input or output pin and used for a wide range of purposes.



NOTE

The numbering of the GPIO pins is not in numerical order; GPIO pins 0 and 1 are present on the board (physical pins 27 and 28) but are reserved for advanced use (see below).

Voltages

Two 5V pins and two 3.3V pins are present on the board, as well as a number of ground pins (0V), which are unconfigurable. The remaining pins are all general purpose 3.3V pins, meaning outputs are set to 3.3V and inputs are 3.3V-tolerant.

Outputs

A GPIO pin designated as an output pin can be set to high (3.3V) or low (0V).

Inputs

A GPIO pin designated as an input pin can be read as high (3.3V) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

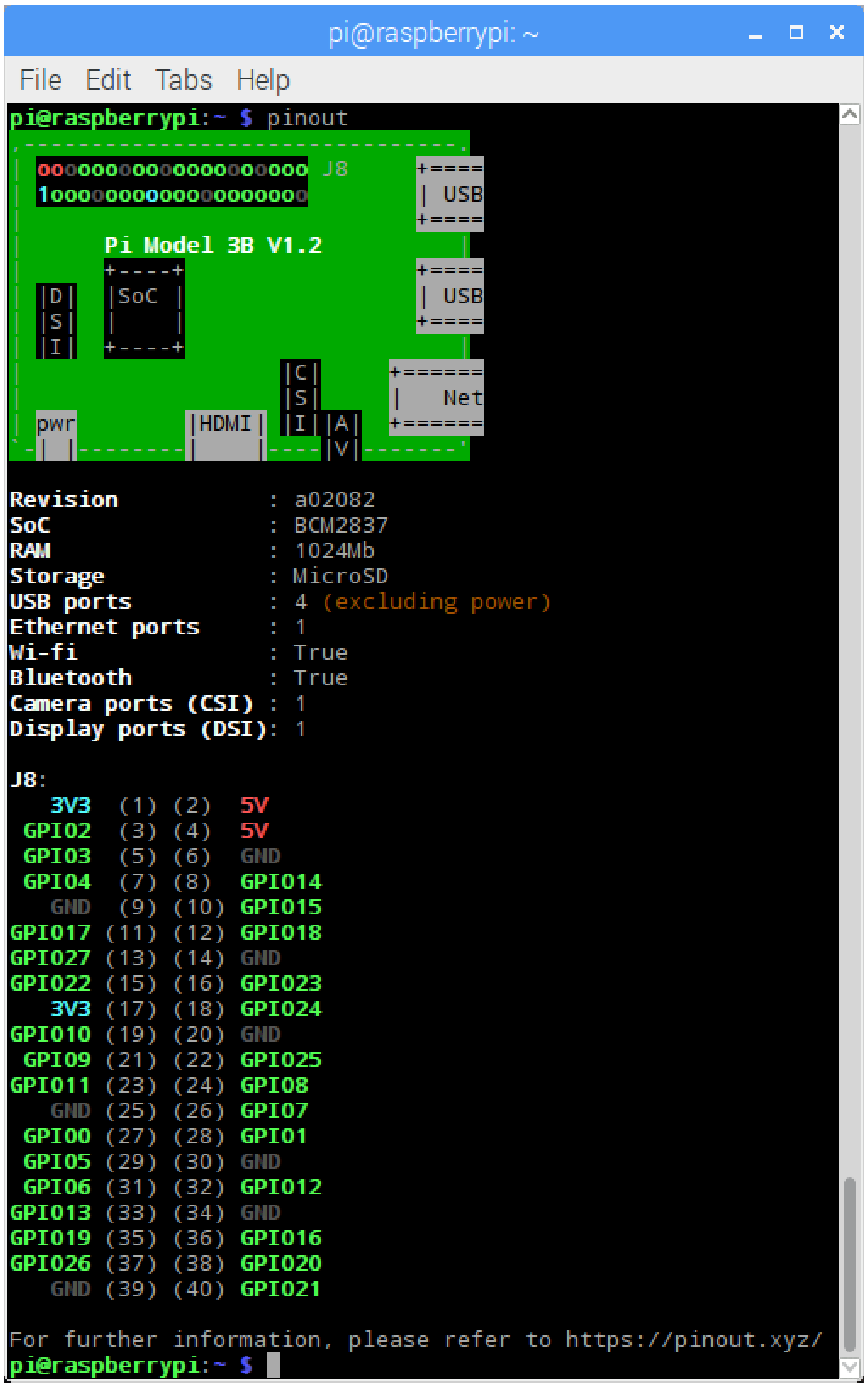
More

As well as simple input and output devices, the GPIO pins can be used with a variety of alternative functions, some are available on all pins, others on specific pins.

- PWM (pulse-width modulation)
  - Software PWM available on all pins
  - Hardware PWM available on GPIO12, GPIO13, GPIO18, GPIO19
- SPI
  - SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)
  - SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16)
- I2C
  - Data: (GPIO2); Clock (GPIO3)
  - EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
- Serial
  - TX (GPIO14); RX (GPIO15)

GPIO pinout

A handy reference can be accessed on the Raspberry Pi by opening a terminal window and running the command `pinout`. This tool is provided by the GPIO Zero Python library, which is installed by default in Raspberry Pi OS.



```
pi@raspberrypi:~ $ pinout
,--------------------------------.
| oooooooooooooooooooo J8     +====
| 1ooooooooooooooooooo        | USB
|                             +====
|      Pi Model 3B V1.2          |
|      +----+                 +====
| |D|  |SoC |                 | USB
| |S|  |    |                 +====
| |I|  +----+                    |
|                   |C|     +======
|                   |S|     |   Net
| pwr        |HDMI| |I||A|  +======
`-| |--------|    |----|V|-------'

Revision           : a02082
SoC                : BCM2837
RAM                : 1024Mb
Storage            : MicroSD
USB ports          : 4 (excluding power)
Ethernet ports     : 1
Wi-fi              : True
Bluetooth          : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
   3V3  (1) (2)  5V
 GPIO2  (3) (4)  5V
 GPIO3  (5) (6)  GND
 GPIO4  (7) (8)  GPIO14
   GND  (9) (10) GPIO15
GPIO17 (11) (12) GPIO18
GPIO27 (13) (14) GND
GPIO22 (15) (16) GPIO23
   3V3 (17) (18) GPIO24
GPIO10 (19) (20) GND
 GPIO9 (21) (22) GPIO25
GPIO11 (23) (24) GPIO8
   GND (25) (26) GPIO7
 GPIO0 (27) (28) GPIO1
 GPIO5 (29) (30) GND
 GPIO6 (31) (32) GPIO12
GPIO13 (33) (34) GND
GPIO19 (35) (36) GPIO16
GPIO26 (37) (38) GPIO20
   GND (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
pi@raspberrypi:~ $
```

For more details on the advanced capabilities of the GPIO pins see gadgetoid's interactive pinout diagram.

WARNING

While connecting up simple components to the GPIO pins is perfectly safe, it's important to be careful how you wire things up. LEDs should have resistors to limit the current passing through them. Do not use 5V for 3.3V components. Do not connect motors directly to the GPIO pins, instead use an H-bridge circuit or a motor controller board.

Permissions

In order to use the GPIO ports your user must be a member of the `gpio` group. The `pi` user is a member by default, other users need to be added manually.

```
sudo usermod -a -G gpio <username>
```

GPIO in Python

Using the GPIO Zero library makes it easy to get started with controlling GPIO devices with Python. The library is comprehensively documented at gpiozero.readthedocs.io.

LED

To control an LED connected to GPIO17, you can use this code:

```
from gpiozero import LED
from time import sleep

led = LED(17)

while True:
    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

Run this in an IDE like Thonny, and the LED will blink on and off repeatedly.

LED methods include `on()`, `off()`, `toggle()`, and `blink()`.

Button

To read the state of a button connected to GPIO2, you can use this code:

```
from gpiozero import Button
from time import sleep

button = Button(2)

while True:
    if button.is_pressed:
        print("Pressed")
    else:
        print("Released")
    sleep(1)
```

Button functionality includes the properties `is_pressed` and `is_held`; callbacks `when_pressed`, `when_released`, and `when_held`; and methods `wait_for_press()` and `wait_for_release`.

Button + LED

To connect the LED and button together, you can use this code:

```
from gpiozero import LED, Button

led = LED(17)
button = Button(2)

while True:
    if button.is_pressed:
        led.on()
    else:
        led.off()
```

Alternatively:

```
from gpiozero import LED, Button

led = LED(17)
button = Button(2)

while True:
    button.wait_for_press()
    led.on()
    button.wait_for_release()
    led.off()
```

or:

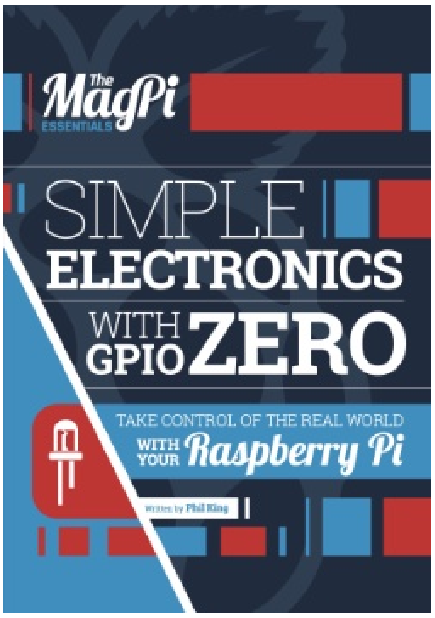
```
from gpiozero import LED, Button

led = LED(17)
button = Button(2)

button.when_pressed = led.on
button.when_released = led.off
```

Going further

You can find more information on how to program electronics connected to your Raspberry Pi with the GPIO Zero Python library in the Raspberry Pi Press book Simple Electronics with GPIO Zero. Written by Phil King, it is part of the MagPi Essentials series published by Raspberry Pi Press. The book gets you started with the GPIO Zero library, and walks you through how to use it by building a series of projects.


The MagPi ESSENTIALS
SIMPLE ELECTRONICS WITH GPIO ZERO
TAKE CONTROL OF THE REAL WORLD WITH YOUR Raspberry Pi
Written by Phil King

You can download this book as a PDF file for free, it has been released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY NC-SA) licence.

Schematics and Mechanical Drawings

Edit this on GitHub

Schematics for the various Raspberry Pi board versions:

Raspberry Pi 4 Model B

- Schematics, Revision 4.0
- Mechanical Drawings, PDF

- Mechanical Drawings, DXF

Raspberry Pi 3 Model B+

- Schematics, Revision 1.0
- Mechanical Drawings, PDF
- Mechanical Drawings, DXF
- Case Drawings, PDF

Raspberry Pi 3 Model B

- Schematics, Revision 1.2
- Mechanical Drawings, PDF
- Mechanical Drawings, DXF

Raspberry Pi 2 Model B

- Schematics, Revision 1.2

Raspberry Pi 1 Model B+

- Schematics, Revision 1.2
- Mechanical Drawings, PDF
- Mechanical Drawings, DXF

Raspberry Pi 3 Model A+

- Schematics, Revision 1.0
- Mechanical Drawings, PDF
- Case Drawings, PDF

NOTE

Mechanical drawings for the Raspberry Pi 3 Model A+ are also applicable to the Raspberry Pi 1 Model A+.

Raspberry Pi 1 Model A+

- Schematics, Revision 1.1

Raspberry Pi Zero

- Schematics, Revision 1.3

- Mechanical Drawings, PDF
- Case Drawings, PDF - Blank Lid
- Case Drawings, PDF - GPIO Lid
- Case Drawings, PDF - Camera Lid

Raspberry Pi Zero W

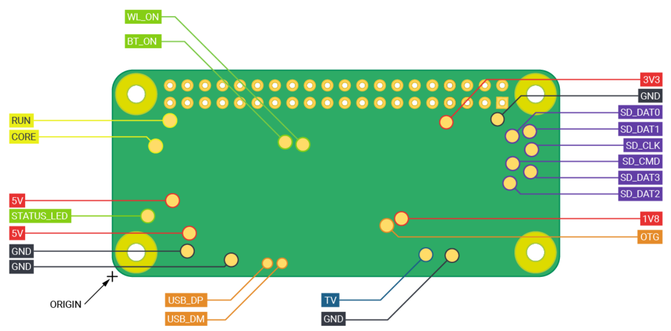
- Schematics, Revision 1.1
- Mechanical Drawings, PDF

Raspberry Pi Zero 2 W

- Schematics
- Mechanical Drawings, PDF
- Test Pad Positions

Test Pad Locations

The Raspberry Pi Zero 2 W has a number of test pad locations used during production of the board.



Label	Function	X (mm from origin)	Y (mm from origin)
STATUS_LED	Power state of LED (LOW = ON)	5.15	8.8
CORE	Processor power	6.3	18.98
RUN	Connect to GND to reset	8.37	22.69
5V	5V Input	8.75	11.05
5V	5V Input	11.21	6.3
GND	Ground pin	10.9	3.69

Label	Function	X (mm from origin)	Y (mm from origin)
GND	Ground pin	17.29	2.41
USB_DP	USB port	22.55	1.92
USB_DM	USB port	24.68	1.92
OTG	On-the-go ID pin	39.9	7.42
1V8	1.8V analog supply	42.03	8.42
TV	Composite TV out	45.58	3.17
GND	Ground pin	49.38	3.05
GND	Ground pin	55.99	22.87
3V3	3.3V I/O supply	48.55	22.44
SD_CLK	SD Card clock pin	60.95	18.45
SD_CMD	SD Card command pin	58.2	16.42
SD_DAT0	SD data pin	58.13	20.42
SD_DAT1	SD data pin	60.65	21.1
SD_DAT2	SD data pin	57.78	13.57
SD_DAT3	SD data pin	60.8	15.22
BT_ON	Bluetooth power status	25.13	19.55
WL_ON	Wireless LAN power status	27.7	19.2

Product compliance and safety

Edit this on GitHub

All Raspberry Pi products have undergone extensive compliance testing, for more information see the Product Information Portal

Flammability Rating

The PCBs used in Raspberry Pi devices adhere to UL94-V0.

NOTE

This applies to the PCBs **only**.

The Raspberry Pi Compliance Support

The Compliance Support programme is designed to eliminate the burden of navigating compliance issues and make it easier for companies to bring new products to consumers. It provides access to the same test engineers who worked on our Raspberry Pis during their compliance testing, connecting the user to a dedicated team at UL who assess and test the user’s product, facilitated by their in-depth knowledge of Raspberry Pi.

Find out more about the Raspberry Pi Compliance Support Programme.

Powered by Raspberry Pi

The Powered by Raspberry Pi progamme provides a process for companies wanting to use a form of the Raspberry Pi logo, and covers products with Raspberry Pi computers or silicon inside, and services provided by a Raspberry Pi. If you wish to start the process to apply you can do so online.

Approved Design Partners

Our list of approved design partners provide a set of consultancies that we work closely with and support so they can provide paid for design services across hardware, software, and mechanical.

Frequency Management and Thermal Control

Edit this on GitHub

All Raspberry Pi models perform a degree of thermal management to avoid overheating under heavy load. The SoCs have an internal temperature sensor, which software on the GPU polls to ensure that temperatures do not exceed a predefined limit; this is 85°C on all models. It is possible to set this to a lower value, but not to a higher one. As the device approaches the limit, various frequencies and sometimes voltages used on the chip (ARM, GPU) are reduced. This reduces the amount of heat generated, keeping the temperature under control.

When the core temperature is between 80°C and 85°C, a warning icon showing a red half-filled thermometer will be displayed, and the ARM cores will be progressively throttled back. If the temperature reaches 85°C, an icon showing a fully filled thermometer will be displayed, and both the ARM cores and the GPU will be throttled back. See the page on warning icons for images of the icons.

For Raspberry Pi 3 Model B+, the PCB technology has been changed to provide better heat dissipation and increased thermal mass. In addition, a soft temperature limit has been introduced, with the goal of maximising the time for which a device can "sprint" before reaching the hard limit at 85°C. When the soft limit is reached, the clock speed is reduced from 1.4GHz to 1.2GHz, and the operating voltage is reduced slightly. This reduces the rate of temperature increase: we trade a short period at 1.4GHz for a longer period at 1.2GHz. By default, the soft limit is 60°C, and this can be changed via the `temp_soft_limit` setting in config.txt.

The Raspberry Pi 4 Model B, continues with the same PCB technology as the Raspberry Pi 3 Model B+, to help dissipate excess heat. There is currently **no soft limit defined**.

Using DVFS

NOTE

Discussion of DVFS applies to Raspberry Pi 4 Model B, Raspberry Pi 400, and Compute Module 4 **only**.

Raspberry Pi 4 devices implement Dynamic Voltage and Frequency Scaling (DVFS). This technique allows Raspberry Pi 4 devices to run at lower temperatures whilst still providing the same performance.

Various clocks (e.g. ARM, Core, V3D, ISP, H264, HEVC) inside the SoC are monitored by the firmware, and whenever they are not running at full speed, the voltage supplied to the particular part of the chip driven by the clock is reduced relative to the reduction from full speed. In effect, only enough voltage is supplied to keep the block running correctly at the specific speed at which it is running. This can result in significant reductions in power used by the SoC, and therefore in the overall heat being produced.

Due to possible system stability problems involved with running an undervoltage, especially when using undervoltaged fixed clock peripherals (eg. PCIe), three DVFS modes are available and can be configured in `/boot/config.txt` with the below properties. Most systems should use `dvfs=3`, headless systems may benefit from a small power reduction with `dvfs=1` at the risk of PCIe stability issues.

property=value	Description
dvfs=1	allow undervoltage
dvfs=2	fixed voltage for default operating frequencies
dvfs=3	scale voltage up on demand for over clocking (default). If `over_voltage` is specified in `config.txt` then dynamic voltage scaling is disabled causing the system to revert to `dvfs=2`.

In addition, a more stepped CPU governor is also used to produce finer-grained control of ARM core frequencies, which means the DVFS is more effective. The steps are now 1500MHz, 1000MHz, 750MHz, and 600MHz. These steps can also help when the SoC is being throttled, and mean that throttling all the way back to 600MHz is much less likely, giving an overall increase in fully loaded performance.

The default CPU governor is `ondemand`, the governor can be manually changed with the `cpufreq-set` command (from the `cpufrequtils` package) to reduce idle power consumption:

```
sudo apt install cpufrequtils
sudo cpufreq-set -g powersave
```

Measuring Temperatures

Due to the architecture of the SoCs used on the Raspberry Pi range, and the use of the upstream temperature monitoring code in the Raspberry Pi OS distribution, Linux-based temperature measurements can be inaccurate. However, the `vcgencmd` command provides an accurate and instantaneous reading of the current SoC temperature as it communicates with the GPU directly:

```
vcgencmd measure_temp
```

Adding Heatsinks

Whilst heatsinks are not necessary to prevent overheating damage to the SoC — the thermal throttling mechanism handles that — a heatsink or small fan will help if you wish to reduce the amount of thermal throttling that takes place. Depending on the exact circumstances, mounting the Raspberry Pi vertically can also help with heat dissipation, as doing so can improve air flow.

Raspberry Pi 4 Boot EEPROM

Edit this on GitHub

Raspberry Pi 4, 400 and Compute Module 4 computers use an EEPROM to boot the system. All other models of Raspberry Pi computer use the `bootcode.bin` file located in the boot filesystem.

NOTE

> The scripts and pre-compiled binaries used to create the `rpi-eeprom` package which is used to update the Raspberry Pi 4 bootloader and VLI USB controller EEPROMs is available on Github.

Boot Diagnostics

If an error occurs during boot then an error code will be displayed via the green LED. Newer versions of the bootloader will display a diagnostic message which will be shown on both HDMI displays.

Updating the Bootloader

Raspberry Pi 4 and Raspberry Pi 400

Raspberry Pi OS automatically updates the bootloader for critical bug fixes. The recommended methods for manually updating the bootloader or changing the boot modes are Raspberry Pi Imager and raspi-config

Using Raspberry Pi Imager to update the bootloader

IMPORTANT

> This is the recommended route to updating the bootloader.

Raspberry Pi Imager provides a GUI for updating the bootloader and selecting the boot mode.

1. Download Raspberry Pi Imager
2. Select a spare SD card. The contents will get overwritten!
3. Launch `Raspberry Pi Imager`
4. Select `Misc utility images` under `Operating System`
5. Select `Bootloader`

6. Select a boot-mode i.e. `SD` (recommended), `USB` or `Network`.
7. Select `SD card` and then `Write`
8. Boot the Raspberry Pi with the new image and wait for at least 10 seconds.
9. The green activity LED will blink with a steady pattern and the HDMI display will be green on success.
10. Power off the Raspberry Pi and remove the SD card.

Using raspi-config to update the bootloader

To change the boot-mode or bootloader version from within Raspberry Pi OS run raspi-config

1. Update Raspberry Pi OS to get the latest version of the `rpi-eeprom` package.
2. Run `sudo raspi-config`
3. Select `Advanced Options`
4. Select `Bootloader Version`
5. Select `Default` for factory default settings or `Latest` for the latest stable bootloader release.
6. Reboot

Updating the EEPROM Configuration

The boot behaviour (e.g. SD or USB boot) is controlled by a configuration file embedded in the EEPROM image and can be modified via the `rpi-eeprom-config` tool.

Please see the Bootloader Configuration sections for details of the configuration.

Reading the current EEPROM configuration

To view the configuration used by the current bootloader during the last boot run `rpi-eeprom-config` or `vcgencmd bootloader_config`.

Reading the configuration from an EEPROM image

To read the configuration from an EEPROM image:

```
rpi-eeprom-config pieeprom.bin
```

Editing the current bootloader configuration

The following command loads the current EEPROM configuration into a text editor. When the editor is closed, `rpi-eeprom-config` applies the updated configuration to latest available EEPROM release and uses `rpi-eeprom-update` to schedule an update when the system is rebooted:

```
sudo -E rpi-eeprom-config --edit
sudo reboot
```

If the updated configuration is identical or empty then no changes are made.

The editor is selected by the `EDITOR` environment variable.

Applying a saved configuration

The following command applies `boot.conf` to the latest available EEPROM image and uses `rpi-eeprom-update` to schedule an update when the system is rebooted.

```
sudo rpi-eeprom-config --apply boot.conf
sudo reboot
```

Automatic Updates

The `rpi-eeprom-update` `systemd` service runs at startup and applies an update if a new image is available, automatically migrating the current bootloader configuration.

To disable automatic updates:

```
sudo systemctl mask rpi-eeprom-update
```

To re-enable automatic updates:

```
sudo systemctl unmask rpi-eeprom-update
```

NOTE

If the FREEZE_VERSION bootloader EEPROM config is set then the EEPROM update service will skip any automatic updates. This removes the need to individually disable the EEPROM update service if there are multiple operating systems installed or when swapping SD-cards.

`rpi-eeprom-update`

Raspberry Pi OS uses the `rpi-eeprom-update` script to implement an automatic update service. The script can also be run interactively or wrapped to create a custom bootloader update service.

Reading the current EEPROM version:

```
vcgencmd bootloader_version
```

Check if an update is available:

```
sudo rpi-eeprom-update
```

Install the update:

```
sudo rpi-eeprom-update -a
sudo reboot
```

Cancel the pending update:

```
sudo rpi-eeprom-update -r
```

Installing a specific bootloader EEPROM image:

```
sudo rpi-eeprom-update -d -f pieeprom.bin
```

The `-d` flag instructs `rpi-eeprom-update` to use the configuration in the specified image file instead of automatically migrating the current configuration.

Display the built-in documentation:

```
rpi-eeprom-update -h
```

Bootloader Release Status

The firmware release status corresponds to a particular subdirectory of bootloader firmware images (`/lib/firmware/raspberrypi/bootloader/...`), and can be changed to select a different release stream.

- `default` - Updated for new hardware support, critical bug fixes and periodic update for new features that have been tested via the `latest` release.
- `latest` - Updated when new features have been successfully beta tested.
- `beta` - New or experimental features are tested here first.

Since the release status string is just a subdirectory name, then it is possible to create your own release streams e.g. a pinned release or custom network boot configuration.

N.B. `default` and `latest` are symbolic links to the older release names of `critical` and `stable`.

Changing the bootloader release

NOTE

You can change which release stream is to be used during an update by editing the `/etc/default/rpi-eeprom-update` file and changing the `FIRMWARE_RELEASE_STATUS` entry to the appropriate stream.

Updating the bootloader configuration in an EEPROM image file

The following command replaces the bootloader configuration in `pieeprom.bin` with `boot.conf` and writes the new image to `new.bin`:

```
rpi-eeprom-config --config boot.conf --out new.bin pieeprom.bin
```

recovery.bin

At power on, the BCM2711 ROM looks for a file called `recovery.bin` in the root directory of the boot partition on the SD card. If a valid `recovery.bin` is found then the ROM executes this instead of the contents of the EEPROM. This mechanism ensures that the bootloader EEPROM can always be reset to a valid image with factory default settings.

See also Raspberry Pi 4 boot-flow

EEPROM update files

Filename	Purpose
recovery.bin	bootloader EEPROM recovery executable
pieeprom.upd	Bootloader EEPROM image
pieeprom.bin	Bootloader EEPROM image - same as pieeprom.upd but changes recovery.bin behaviour
pieeprom.sig	The sha256 checksum of bootloader image (pieeprom.upd/pieeprom.bin)
vl805.bin	The VLI805 USB firmware EEPROM image - ignored on 1.4 and later board revisions which do not have a dedicated VLI EEPROM
vl805.sig	The sha256 checksum of vl805.bin

- If the bootloader update image is called `pieeprom.upd` then `recovery.bin` is renamed to `recovery.000` once the update has completed, then the system is rebooted. Since `recovery.bin` is no longer present the ROM loads the newly updated bootloader from EEPROM and the OS is booted as normal.
- If the bootloader update image is called `pieeprom.bin` then `recovery.bin` will stop after the update has completed. On success the HDMI output will be green and the green activity LED is flashed rapidly. If the update fails, the HDMI output will be red and an error code will be displayed via the activity LED.
- The `.sig` files contain the hexadecimal sha256 checksum of the corresponding image file; additional fields may be added in the future.
- The BCM2711 ROM does not support loading `recovery.bin` from USB mass storage or TFTP. Instead, newer versions of the bootloader support a self-update mechanism where the bootloader is able to reflash the EEPROM itself. See `ENABLE_SELF_UPDATE` on the bootloader configuration page.
- The temporary EEPROM update files are automatically deleted by the `rpi-eeprom-update` service at startup.

For more information about the `rpi-eeprom-update` configuration file see `rpi-eeprom-update -h`.

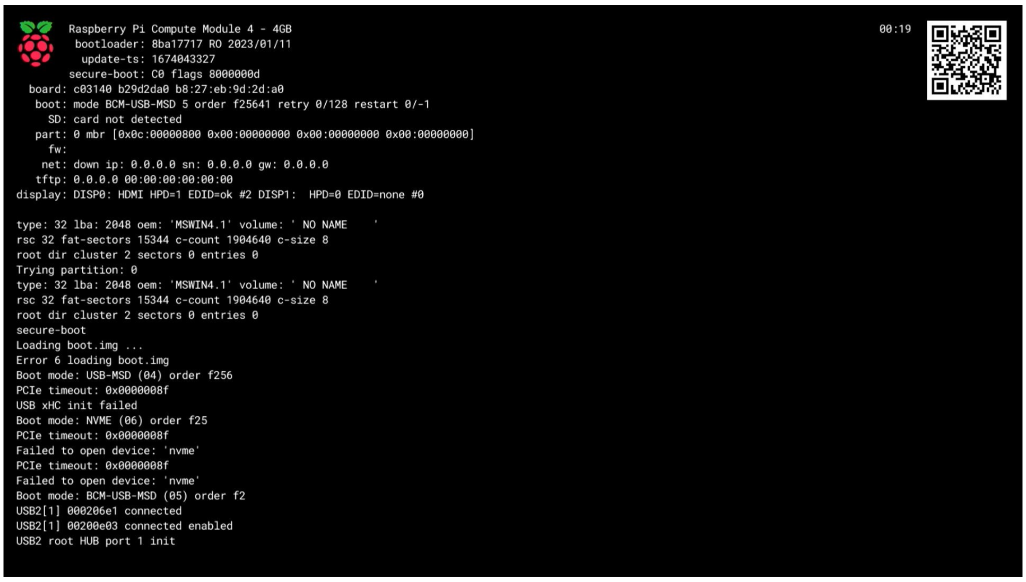
EEPROM write protect

Both the bootloader and VLI EEPROMs support hardware write protection. See the eeprom_write_protect option for more information about how to enable this when flashing the EEPROMs.

Boot Diagnostics on the Raspberry Pi 4

Edit this on GitHub

Starting with version 2020-04-16 of the Raspberry Pi 4 bootloader, diagnostic information can be displayed at boot time on an HDMI display. To see this diagnostic information, power down the Raspberry Pi 4, remove the SD card, then power back up. A diagnostic display similar to below should appear on the attached display.



```
        Raspberry Pi Compute Module 4 - 4GB
         bootloader: 8ba17717 RO 2023/01/11
          update-ts: 1674043327
        secure-boot: C0 flags 8000000d
  board: c03140 b29d2da0 b8:27:eb:9d:2d:a0
   boot: mode BCM-USB-MSD 5 order f25641 retry 0/128 restart 0/-1
     SD: card not detected
   part: 0 mbr [0x0c:00000800 0x00:00000000 0x00:00000000 0x00:00000000]
     fw:
    net: down ip: 0.0.0.0 sn: 0.0.0.0 gw: 0.0.0.0
   tftp: 0.0.0.0 00:00:00:00:00:00
display: DISP0: HDMI HPD=1 EDID=ok #2 DISP1:  HPD=0 EDID=none #0

type: 32 lba: 2048 oem: 'MSWIN4.1' volume: ' NO NAME    '
rsc 32 fat-sectors 15344 c-count 1904640 c-size 8
root dir cluster 2 sectors 0 entries 0
Trying partition: 0
type: 32 lba: 2048 oem: 'MSWIN4.1' volume: ' NO NAME    '
rsc 32 fat-sectors 15344 c-count 1904640 c-size 8
root dir cluster 2 sectors 0 entries 0
secure-boot
Loading boot.img ...
Error 6 loading boot.img
Boot mode: USB-MSD (04) order f256
PCIe timeout: 0x0000008f
USB xHC init failed
Boot mode: NVME (06) order f25
PCIe timeout: 0x0000008f
Failed to open device: 'nvme'
PCIe timeout: 0x0000008f
Failed to open device: 'nvme'
Boot mode: BCM-USB-MSD (05) order f2
USB2[1] 000206e1 connected
USB2[1] 00200e03 connected enabled
USB2 root HUB port 1 init
```

This diagnostics page will also appear if the bootloader is unable to boot from an inserted SD card, or is unable to network boot; for example, if there is no bootable image on the card, or it is defective, or the network boot parameters are incorrect.

Once the diagnostics page is displayed, a reboot is only possible by power cycling the device (i.e. unplug then re-plug the power supply).

The top line describes the model of Raspberry Pi and its memory capacity. The QR code is a link to the Downloads Page.

The diagnostic information is as follows:

Line:	**Information**
bootloader	Bootloader git version - RO (if EEPROM is write protected) - software build date
update-ts	The timestamp corresponding to when the EEPROM configuration was updated. This timestamp is checked in self-update mode to avoid updating to an old configuration.

Line:	Information
secure-boot	If secure-boot is enabled then the processor revision (B0/C0) and signed-boot status flags are displayed. Otherwise, this line is blank.
board	Board revision - Serial Number - Ethernet MAC address
boot	**mode** (current boot mode name and number) **order** (the BOOT ORDER configuration) **retry** (retry count in the current boot mode) **restart** (number of cycles through the list of boot modes).
SD	The SD card detect status (detected / not detected).
part	Master Boot Record primary partitions type:LBA.
fw	Filename for start.elf and fixup.dat if present (e.g. start4x.elf, fixup4x.dat).
net	Network boot: - Link status (up/down) client IP address (ip), Subnet (sn), Default gateway (gw)
tftp	Network boot: TFTP server IP address
display	Indicates whether hotplug was detected (`HPD=1`) and if so whether the EDID was read successfully (`EDID=ok`) for each HDMI output.

This display can be disabled using the `DISABLE_HDMI` option, see Bootloader Configuration.

NOTE

This is purely for diagnosing boot failures; it is not an interactive bootloader. If you require an interactive bootloader, consider using a tool such as U-Boot.

Raspberry Pi Boot Modes

Edit this on GitHub

The Raspberry Pi has a number of different stages of booting. This document explains how the boot modes work, and which ones are supported for Linux booting.

Special `bootcode.bin`-only boot mode

USB host and Ethernet boot can be performed by BCM2837-based Raspberry Pis - that is, Raspberry Pi 2B version 1.2, Raspberry Pi 3B, and Raspberry Pi 3B+ (Raspberry Pi 3A+ cannot net boot since it does not have a built-in Ethernet interface). In addition, all Raspberry Pi models **except Raspberry Pi 4B** can use a new `bootcode.bin`-only method to enable USB host boot.

NOTE

The Raspberry Pi 4B does not use the bootcode.bin file - instead the bootloader is located in an on-board EEPROM chip. See Raspberry Pi 4 Bootflow and SPI Boot EEPROM.

Format an SD card as FAT32 and copy on the latest `bootcode.bin`. The SD card must be present in the Raspberry Pi for it to boot. Once bootcode.bin is loaded from the SD card, the Raspberry Pi continues booting using USB host mode.

This is useful for the Raspberry Pi 1, 2, and Zero models, which are based on the BCM2835 and BCM2836 chips, and in situations where a Raspberry Pi 3 fails to boot (the latest bootcode.bin includes additional bugfixes for the Raspberry Pi 3B, compared to the boot code burned into the BCM2837A0).

If you have a problem with a mass storage device still not working, even with this bootcode.bin, then please add a new file 'timeout' to the SD card. This will extend to six seconds the time for which it waits for the mass storage device to initialise.

`bootcode.bin` UART Enable

NOTE

For boards pre-Raspberry Pi 4 Model B.

For information on enabling the UART on the Raspberry Pi 4 bootloader, please see this page.

It is possible to enable an early stage UART to debug booting issues (useful with the above bootcode.bin only boot mode). To do this, make sure you've got a recent version of the firmware (including bootcode.bin). To check if UART is supported in your current firmware:

```
strings bootcode.bin | grep BOOT_UART
```

To enable UART from bootcode.bin use:

```
sed -i -e "s/BOOT_UART=0/BOOT_UART=1/" bootcode.bin
```

Next, connect a suitable USB serial cable to your host computer (a Raspberry Pi will work, although I find the easiest path is to use a USB serial cable since it'll work out the box without any pesky config.txt settings). Use the standard pins 6, 8 and 10 (GND, GPIO14, GPIO15) on a Raspberry Pi or Compute Module board.

Then use `screen` on linux or a Mac or `putty` on windows to connect to the serial.

Setup your serial to receive at 115200-8-N-1, and then boot your Raspberry Pi / Compute Module. You should get an immediate serial output from the device as bootcode.bin runs.

Boot sequence

Edit this on GitHub

IMPORTANT

The following boot sequence applies to the BCM2837 and BCM2837B0 based models of Raspberry Pi only. On models prior to this, the Raspberry Pi will try SD card boot, followed by USB device mode boot. For the Raspberry Pi 4 boot sequence please see the Raspberry Pi 4 boot flow section.

USB boot defaults on the Raspberry Pi 3 will depend on which version is being used. See this page for information on enabling USB boot modes when not enabled by default.

When the BCM2837 boots, it uses two different sources to determine which boot modes to enable. Firstly, the OTP (one-time programmable) memory block is checked to see which boot modes are enabled. If the GPIO boot mode setting is enabled, then the relevant GPIO lines are tested to select which of the OTP-enabled boot modes should be attempted. Note that GPIO boot mode can only be used to select boot modes that are already enabled in the OTP. See GPIO boot mode for details on configuring GPIO boot mode. GPIO boot mode is disabled by default.

Next, the boot ROM checks each of the boot sources for a file called bootcode.bin; if it is successful it will load the code into the local 128K cache and jump to it. The overall boot mode process is as follows:

- BCM2837 boots
- Read OTP to determine which boot modes to enable
- If GPIO boot mode enabled, use GPIO boot mode to refine list of enabled boot modes
- If enabled: check primary SD for bootcode.bin on GPIO 48-53
  - Success - Boot
  - Fail - timeout (five seconds)
- If enabled: check secondary SD
  - Success - Boot
  - Fail - timeout (five seconds)
- If enabled: check NAND
- If enabled: check SPI
- If enabled: check USB
  - If OTG pin == 0
    - Enable USB, wait for valid USB 2.0 devices (two seconds)
      - Device found:
        - If device type == hub
          - Recurse for each port
        - If device type == (mass storage or LAN951x)
          - Store in list of devices
    - Recurse through each MSD
      - If bootcode.bin found boot
    - Recurse through each LAN951x

        - DHCP / TFTP boot
    - else (Device mode boot)
        - Enable device mode and wait for host PC to enumerate
        - We reply to PC with VID: 0a5c PID: 0x2763 (Raspberry Pi 1 or Raspberry Pi 2) or 0x2764 (Raspberry Pi 3)

NOTE

- If there is no SD card inserted, the SD boot mode takes five seconds to fail. To reduce this and fall back to USB more quickly, you can either insert an SD card with nothing on it or use the GPIO bootmode OTP setting described above to only enable USB.
- The default pull for the GPIOs is defined on page 102 of the ARM Peripherals datasheet. If the value at boot time does not equal the default pull, then that boot mode is enabled.
- USB enumeration is a means of enabling power to the downstream devices on a hub, then waiting for the device to pull the D+ and D- lines to indicate if it is either USB 1 or USB 2. This can take time: on some devices it can take up to three seconds for a hard disk drive to spin up and start the enumeration process. Because this is the only way of detecting that the hardware is attached, we have to wait for a minimum amount of time (two seconds). If the device fails to respond after this maximum timeout, it is possible to increase the timeout to five seconds using `program_usb_boot_timeout=1` in `config.txt`.
- MSD boot takes precedence over Ethernet boot.
- It is no longer necessary for the first partition to be the FAT partition, as the MSD boot will continue to search for a FAT partition beyond the first one.
- The boot ROM also now supports GUID partitioning and has been tested with hard drives partitioned using Mac, Windows, and Linux.
- The LAN951x is detected using the Vendor ID 0x0424 and Product ID 0xec00: this is different to the standalone LAN9500 device, which has a product ID of 0x9500 or 0x9e00. To use the standalone LAN9500, an I2C EEPROM would need to be added to change these IDs to match the LAN951x.

The primary SD card boot mode is, as standard, set to be GPIOs 49-53. It is possible to boot from the secondary SD card on a second set of pins, i.e. to add a secondary SD card to the GPIO pins. However, we have not yet enabled this ability.

NAND boot and SPI boot modes do work, although they do not yet have full GPU support.

The USB device boot mode is enabled by default at the time of manufacture, but the USB host boot mode is only enabled with `program_usb_boot_mode=1`. Once enabled, the processor will use the value of the OTGID pin on the processor to decide between the two modes. On any Raspberry Pi Model B / B+, the OTGID pin is driven to '0' and therefore will only boot via host mode once enabled (it is not possible to boot through device mode because the LAN951x device is in the way).

The USB will boot as a USB device on the Raspberry Pi Zero or Compute Module if the OTGID pin is left floating (when plugged into a PC for example), so you can 'squirt' the bootcode.bin into the device. The `usbboot` code for doing this is available on Github.

Raspberry Pi 4 Boot Flow

Edit this on GitHub

The main difference between this and previous products is that the second stage bootloader is loaded from an SPI flash EEPROM instead of the `bootcode.bin` file on previous products.

First Stage Bootloader

The boot flow for the ROM (first stage) is as follows:-

- BCM2711 SoC powers up
- Read OTP to determine if the `nRPIBOOT` GPIO is configured
- If `nRPIBOOT` GPIO is high or OTP does NOT define `nRPIBOOT` GPIO
  - Check OTP to see if `recovery.bin` can be loaded from SD/EMMC
    - If SD recovery.bin is enabled then check primary SD/EMMC for `recovery.bin`
      - Success - run `recovery.bin` and update the SPI EEPROM
      - Fail - continue
  - Check SPI EEPROM for second stage loader
    - Success - run second stage bootloader
    - Fail - continue
- While True
  - Attempt to load recovery.bin from USB device boot
    - Success - run `recovery.bin` and update the SPI EEPROM or switch to USB mass storage device mode
    - Fail - retry USB device boot

NOTE

Currently only CM4 reserves a GPIO for `nRPIBOOT`.

NOTE

`recovery.bin` is a minimal second stage program used to reflash the bootloader SPI EEPROM image.

Second Stage Bootloader

This section describes the high-level flow of the second stage bootloader.

Please see the bootloader configuration page for more information about each boot mode and the boot folder page for a description of the GPU firmware files loaded by this stage.

- Initialise clocks and SDRAM
- Read the EEPROM configuration file
- Check `PM_RSTS` register to determine if HALT is requested

  - Check `POWER_OFF_ON_HALT` and `WAKE_ON_GPIO` EEPROM configuration settings.
  - If `POWER_OFF_ON_HALT` is `1` and `WAKE_ON_GPIO` is `0` then
    - Use PMIC to power off system
  - else if `WAKE_ON_GPIO` is `1`
    - Enable fall-edge interrupts on GPIO3 to wake-up if GPIO3 is pulled low
  - sleep
- While True
  - Read the next boot-mode from the BOOT_ORDER parameter in the EEPROM config file.
  - If boot-mode == `RESTART`
    - Jump back to the first boot-mode in the `BOOT_ORDER` field
  - else if boot-mode == `STOP`
    - Display start.elf not found error pattern and wait forever.
  - else if boot-mode == `SD CARD`
    - Attempt to load firmware from the SD card
      - Success - run the firmware
      - Failure - continue
  - else if boot-mode == `NETWORK` then
    - Use DHCP protocol to request IP address
    - Load firmware from the DHCP or statically defined TFTP server
    - If the firmware is not found or a timeout or network error occurs then continue
  - else if boot-mode == `USB-MSD` or boot-mode == `BCM-USB-MSD` then
    - While USB discover has not timed out
      - Check for USB mass storage devices
      - If a new mass storage device is found then
        - For each drive (LUN)
          - Attempt to load firmware
            - Success - run the firmware
            - Failed - advance to next LUN
  - else if boot-mode == `NVME` then
    - Scan PCIe for an NVMe device and if found
      - Attempt to load firmware from the NVMe device

        - Success - run the firmware
        - Failure - continue
    - else if boot-mode == `RPIBOOT` then
        - Attempt to load firmware using USB device mode from the USB OTG port - see usbboot. There is no timeout for `RPIBOOT` mode.

Bootloader Updates

The bootloader may also be updated before the firmware is started if a `pieeprom.upd` file is found. Please see the bootloader EEPROM page for more information about bootloader updates.

Fail-safe OS updates (TRYBOOT)

The bootloader/firmware provide a one-shot flag which, if set, is cleared but causes `tryboot.txt` to be loaded instead of `config.txt`. This alternate config would specify the pending OS update firmware, cmdline, kernel and os_prefix parameters. Since the flag is cleared before starting the firmware, a crash or reset will cause the original `config.txt` file to be loaded on the next reboot.

To set the `tryboot` flag add `tryboot` after the partition number in the `reboot` command. Normally, the partition number defaults to zero but it must be specified if extra arguments are added.

```
# Quotes are important. Reboot only accepts a single argument.
sudo reboot '0 tryboot'
```

`tryboot` is supported on all Raspberry Pi models, however, on Raspberry Pi 4 Model B revision 1.0 and 1.1 the EEPROM must not be write protected. This is because older Raspberry Pi 4B devices have to reset the power supply (losing the tryboot state) so this is stored inside the EEPROM instead.

If `secure-boot` is enabled then `tryboot` mode will cause `tryboot.img` to be loaded instead of `boot.img`.

TRYBOOT_A_B mode

If the `tryboot_a_b` property in autoboot.txt is set to `1` then `config.txt` is loaded instead of `tryboot.txt`. This is because the `tryboot` switch has already been made at a higher level (the partition) and so it's unnecessary to have a `tryboot.txt` file within alternate partition itself.

N.B. The `tryboot_a_b` property is implicitly set to `1` when loading files from within a `boot.img` ramdisk.

Raspberry Pi 4 Bootloader Configuration

Edit this on GitHub

Editing the Configuration

Before editing the bootloader configuration, update your system to get the latest version of the `rpi-eeprom` package.

To view the current EEPROM configuration:
`rpi-eeprom-config`

To edit it and apply the updates to latest EEPROM release:
`sudo -E rpi-eeprom-config --edit`

Please see the boot EEPROM page for more information about the EEPROM update process.

Configuration Properties

This section describes all the configuration items available in the bootloader. The syntax is the same as config.txt but the properties are specific to the bootloader. Conditional filters are also supported except for EDID.

BOOT_UART

If `1` then enable UART debug output on GPIO 14 and 15. Configure the receiving debug terminal at 115200bps, 8 bits, no parity bits, 1 stop bit.

Default: `0`

WAKE_ON_GPIO

If `1` then `sudo halt` will run in a lower power mode until either GPIO3 or GLOBAL_EN are shorted to ground.

Default: `1`

POWER_OFF_ON_HALT

If `1` and `WAKE_ON_GPIO=0` then `sudo halt` will switch off all PMIC outputs. This is lowest possible power state for halt but may cause problems with some HATs because 5V will still be on. `GLOBAL_EN` must be shorted to ground to boot.

Raspberry Pi 400 has a dedicated power button which operates even if the processor is switched off. This behaviour is enabled by default, however, `WAKE_ON_GPIO=2` may be set to use an external GPIO power button instead of the dedicated power button.

Default: `0`

BOOT_ORDER

The `BOOT_ORDER` setting allows flexible configuration for the priority of different boot modes. It is represented as a 32-bit unsigned integer where each nibble represents a boot-mode. The boot modes are attempted in lowest significant nibble to highest significant nibble order.

`BOOT_ORDER` fields

The BOOT_ORDER property defines the sequence for the different boot modes. It is read right to left and up to 8 digits may be defined.

Value	Mode	Description
0x0	SD CARD DETECT	Try SD then wait for card-detect to indicate that the card has changed - deprecated now that 0xf (RESTART) is available.
0x1	SD CARD	SD card (or eMMC on Compute Module 4).
0x2	NETWORK	Network boot - See Network boot server tutorial
0x3	RPIBOOT	RPIBOOT - See usbboot
0x4	USB-MSD	USB mass storage boot - See USB mass storage boot
0x5	BCM-USB-MSD	USB 2.0 boot from USB Type C socket (CM4: USB type A socket on CM4IO board).
0x6	NVME	CM4 only: boot from an NVMe SSD connected to the PCIe interface. See NVMe boot for more details.
0x7	HTTP	HTTP boot over ethernet. See HTTP boot for more details.
0xe	STOP	Stop and display error pattern. A power cycle is required to exit this state.
0xf	RESTART	Restart from the first boot-mode in the BOOT_ORDER field i.e. loop

`RPIBOOT` is intended for use with Compute Module 4 to load a custom debug image (e.g. a Linux RAM-disk) instead of the normal boot. This should be the last boot option because it does not currently support timeouts or retries.

`BOOT_ORDER` examples

BOOT_ORDER	Description
0xf41	Try SD first, followed by USB-MSD then repeat (default if BOOT_ORDER is empty)
0xf14	Try USB first, followed by SD then repeat
0xf21	Try SD first, followed by NETWORK then repeat

MAX_RESTARTS

If the RESTART (`0xf`) boot-mode is encountered more than MAX_RESTARTS times then a watchdog reset is triggered. This isn't recommended for general use but may be useful for test or remote systems where a full reset is needed to resolve issues with hardware or network interfaces.

Default: `-1` (infinite)

SD_BOOT_MAX_RETRIES

The number of times that SD boot will be retried after failure before moving to the next boot-mode defined by `BOOT_ORDER`.
`-1` means infinite retries.

Default: `0`

NET_BOOT_MAX_RETRIES

The number of times that network boot will be retried after failure before moving to the next boot-mode defined by `BOOT_ORDER`.
`-1` means infinite retries.

Default: `0`

DHCP_TIMEOUT

The timeout in milliseconds for the entire DHCP sequence before failing the current iteration.

Minimum: `5000`
Default: `45000`

DHCP_REQ_TIMEOUT

The timeout in milliseconds before retrying DHCP DISCOVER or DHCP REQ.

Minimum: `500`
Default: `4000`

TFTP_FILE_TIMEOUT

The timeout in milliseconds for an individual file download via TFTP.

Minimum: `5000`
Default: `30000`

TFTP_IP

Optional dotted decimal ip address (e.g. `192.168.1.99`) for the TFTP server which overrides the server-ip from the DHCP request.
This may be useful on home networks because tftpd-hpa can be used instead of dnsmasq where broadband router is the DHCP server.

Default: ""

TFTP_PREFIX

In order to support unique TFTP boot directories for each Raspberry Pi the bootloader prefixes the filenames with a device specific directory. If neither start4.elf nor start.elf are found in the prefixed directory then the prefix is cleared. On earlier models the serial number is used as the prefix, however, on Raspberry Pi 4 the MAC address is no longer generated from the serial number making it difficult to automatically create tftpboot directories on the server by inspecting DHCPDISCOVER packets. To support this the TFTP_PREFIX may be customized to either be the MAC address, a fixed value or the serial number (default).

Value	**Description**
0	Use the serial number e.g. `9ffefdef/`
1	Use the string specified by TFTP_PREFIX_STR
2	Use the MAC address e.g. `dc-a6-32-01-36-c2/`

Default: 0

TFTP_PREFIX_STR

Specify the custom directory prefix string used when `TFTP_PREFIX` is set to 1. For example:- `TFTP_PREFIX_STR=tftp_test/`

Default: ""
Max length: 32 characters

PXE_OPTION43

Overrides the PXE Option43 match string with a different string. It's normally better to apply customisations to the DHCP server than change the client behaviour but this option is provided in case that's not possible.

Default: `Raspberry Pi Boot`

DHCP_OPTION97

In earlier releases the client GUID (Option97) was just the serial number repeated 4 times. By default, the new GUID format is the concatenation of the fourcc for `RPi4` (0x34695052 - little endian), the board revision (e.g. 0x00c03111) (4-bytes), the least significant 4 bytes of the mac address and the 4-byte serial number. This is intended to be unique but also provide structured information to the DHCP server, allowing Raspberry Pi 4 computers to be identified without relying upon the Ethernet MAC OUID.

Specify DHCP_OPTION97=0 to revert the old behaviour or a non-zero hex-value to specify a custom 4-byte prefix.

Default: `0x34695052`

MAC_ADDRESS

Overrides the Raspberry Pi Ethernet MAC address with the given value. e.g. `dc:a6:32:01:36:c2`

Default: ""

MAC_ADDRESS_OTP

Overrides the Raspberry Pi Ethernet MAC address with a value stored in the Customer OTP registers.

For example, to use a MAC address stored in rows 0 and 1 of the `Customer OTP`.

```
MAC_ADDRESS_OTP=0,1
```

The first value (row 0 in the example) contains the OUI and the most significant 8 bits of the MAC address. The second value (row 1 in the example) stores the remaining 16-bits of the MAC address. This is the same format as used for the Raspberry Pi MAC address programmed at manufacture.

Any two customer rows may be selected and combined in either order.

The `Customer OTP` rows are OTP registers 36 to 43 in the `vcgencmd otp_dump` output so if the first two rows are programmed as follows then `MAC_ADDRESS_OTP=0,1` would give a MAC address of `e4:5f:01:20:24:7e`.

```
36:247e0000
37:e45f0120
```

Default: ""

Static IP address configuration

If TFTP_IP and the following options are set then DHCP is skipped and the static IP configuration is applied. If the TFTP server is on the same subnet as the client then GATEWAY may be omitted.

CLIENT_IP

The IP address of the client e.g. `192.168.0.32`

Default: ""

SUBNET

The subnet address mask e.g. `255.255.255.0`

Default: ""

GATEWAY

The gateway address to use if the TFTP server is on a different subnet e.g. `192.168.0.1`

Default: ""

DISABLE_HDMI

The HDMI boot diagnostics display is disabled if `DISABLE_HDMI=1`. Other non-zero values are reserved for future use.

Default: `0`

HDMI_DELAY

Skip rendering of the HDMI diagnostics display for up to N seconds (default 5) unless a fatal error occurs. The default behaviour is designed to avoid the bootloader diagnostics screen from briefly appearing during a normal SD / USB boot.

Default: `5`

ENABLE_SELF_UPDATE

Enables the bootloader to update itself from a TFTP or USB mass storage device (MSD) boot filesystem.

If self update is enabled then the bootloader will look for the update files (.sig/.upd) in the boot file system. If the update image differs from the current image then the update is applied and system is reset. Otherwise, if the EEPROM images are byte-for-byte identical then boot continues as normal.

Notes:-

- Self-update is not enabled in SD boot; the ROM can already load recovery.bin from the SD card.
- Bootloader releases prior to 2021 do not support `self-update`.
- For network boot make sure that the TFTP `boot` directory can be mounted via NFS and that `rpi-eeprom-update` can write to it.

Default: `1`

FREEZE_VERSION

Previously this property was only checked by the `rpi-eeprom-update` script. However, now that self-update is enabled the bootloader will also check this property. If set to 1, this overrides `ENABLE_SELF_UPDATE` to stop automatic updates. To disable `FREEZE_VERSION` you will have to use an SD card boot with recovery.bin.

Custom EEPROM update scripts must also check this flag.

Default: `0`

HTTP_HOST

If network install or HTTP boot is initiated, `boot.img` and `boot.sig` are downloaded from this server.

Invalid host names will be ignored. They should only contain lower case alphanumeric characters and `-` or `.`. If `HTTP_HOST` is set then HTTPS is disabled and plain HTTP used instead. You can specify an IP address to avoid the need for a DNS lookup. Don`t include the HTTP scheme or any forward slashes in the hostname.

Default: `fw-download-alias1.raspberrypi.com`

HTTP_PORT

You can use this property to change the port used for network install and HTTP boot. HTTPS is enabled when using the default host `fw-download-alias1.raspberrypi.com`. If `HTTP_HOST` is changed then HTTPS is disabled and plain HTTP will be used instead.

When HTTPS is disabled, plain HTTP will still be used even if `HTTP_PORT` is changed to `443`.

Default: `443` if HTTPS is enabled otherwise `80`

HTTP_PATH

The path used for network install and HTTP boot.

The case of the path **is** significant. Use forward (Linux) slashes for the path separator. Leading and trailing forward slashes are not required.

If `HTTP_HOST` is not set, `HTTP_PATH` is ignored and the URL will be `https://fw-download-alias1.raspberrypi.com:443/net_install/boot.img`. If `HTTP_HOST` is set the URL will be `http://<HTTP_HOST>:<HTTP_PORT>/<HTTP_PATH>/boot.img`

Default: `net_install`

IMAGER_REPO_URL

The embedded Raspberry Pi Imager application is configured with a json file downloaded at startup.

You can change the URL of the json file used by the embedded Raspberry Pi Imager application to get it to offer your own images. You can test this with the standard Raspberry Pi Imager application by passing the URL via the `--repo` argument.

Default: `http://downloads.raspberrypi.org/os_list_imagingutility_v3.json`

NET_INSTALL_ENABLED

When network install is enabled, the bootloader displays the network install screen on boot if it detects a keyboard.

To enable network install, add `NET_INSTALL_ENABLED=1`, or to disable network install add `NET_INSTALL_ENABLED=0`.

This setting is ignored and network install is disabled if `DISABLE_HDMI=1` is set.

In order to detect the keyboard, network install must initialise the USB controller and enumerate devices. This increases boot time by approximately 1 second so it may be advantageous to disable network install in some embedded applications.

Default: `1` on Raspberry Pi 4 and Raspberry Pi 400, and `0` on Compute Module 4.

NET_INSTALL_KEYBOARD_WAIT

If network install is enabled, the bootloader attempts to detect a keyboard and the `SHIFT` key to initiate network install. You can change the length of this wait in milliseconds with this property.

Setting this to `0` disables the keyboard wait, although network install can still be initiated if no boot files are found and USB boot-mode `4` is in `BOOT_ORDER`.

NOTE

Testing suggests keyboard and SHIFT detection takes at least 750ms.

Default: `900`

NETCONSOLE - advanced logging

`NETCONSOLE` duplicates debug messages to the network interface. The IP addresses and ports are defined by the `NETCONSOLE` string.

NOTE

NETCONSOLE blocks until the ethernet link is established or a timeout occurs. The timeout value is `DHCP_TIMEOUT` although DHCP is not attempted unless network boot is requested.

Format

See https://wiki.archlinux.org/index.php/Netconsole

```
src_port@src_ip/dev_name,dst_port@dst_ip/dst_mac
E.g. 6665@169.254.1.1/,6666@/
```

In order to simplify parsing, the bootloader requires every field separator to be present. The source ip address must be specified but the following fields may be left blank and assigned default values.

- src_port - 6665
- dev_name - "" (the device name is always ignored)
- dst_port - 6666
- dst_ip - 255.255.255.255
- dst_mac - 00:00:00:00:00

One way to view the data is to connect the test Raspberry Pi 4 to another Raspberry Pi running WireShark and select “udp.srcport == 6665” as a filter and select `Analyze -> Follow -> UDP stream` to view as an ASCII log.

`NETCONSOLE` should not be enabled by default because it may cause network problems. It can be enabled on demand via a GPIO filter e.g.

```
# Enable debug if GPIO 7 is pulled low
[gpio7=0]
NETCONSOLE=6665@169.254.1.1/,6666@/
```

Default: "" (not enabled)
Max length: 32 characters

PARTITION

The `PARTITION` option may be used to specify the boot partition number, if it has not explicitly been set by the `reboot` command (e.g. `sudo reboot N`) or by `boot_partition=N` in `autoboot.txt`. This could be used to boot from a rescue partition if the user presses a button.

```
# Boot from partition 2 if GPIO 7 is pulled low
[gpio7=0]
PARTITION=2
```

Default: 0

USB_MSD_EXCLUDE_VID_PID

A list of up to 4 VID/PID pairs specifying devices which the bootloader should ignore. If this matches a HUB then the HUB won't be enumerated, causing all downstream devices to be excluded. This is intended to allow problematic (e.g. very slow to enumerate) devices to be ignored during boot enumeration. This is specific to the bootloader and is not passed to the OS.

The format is a comma-separated list of hexadecimal values with the VID as most significant nibble. Spaces are not allowed. E.g. `034700a0,a4231234`

Default: ""

USB_MSD_DISCOVER_TIMEOUT

If no USB mass storage devices are found within this timeout then USB-MSD is stopped and the next boot-mode is selected

Minimum: `5000` (5 seconds)
Default: `20000` (20 seconds)

USB_MSD_LUN_TIMEOUT

How long to wait in milliseconds before advancing to the next LUN e.g. a multi-slot SD-CARD reader. This is still being tweaked but may help speed up boot if old/slow devices are connected as well as a fast USB-MSD device containing the OS.

Minimum: `100`
Default: `2000` (2 seconds)

USB_MSD_PWR_OFF_TIME

During USB mass storage boot, power to the USB ports is switched off for a short time to ensure the correct operation of USB mass storage devices. Most devices work correctly using the default setting: change this only if you have problems booting from a particular device. Setting `USB_MSD_PWR_OFF_TIME=0` will prevent power to the USB ports being switched off during USB mass storage boot.

Minimum: `250`
Maximum: `5000`
Default: `1000` (1 second)

USB_MSD_STARTUP_DELAY

If defined, delays USB enumeration for the given timeout after the USB host controller has initialised. If a USB hard disk drive takes a long time to initialise and triggers USB timeouts then this delay can be used to give the driver additional time to initialise. It may also be necessary to increase the overall USB timeout (`USB_MSD_DISCOVER_TIMEOUT`).

Minimum: `0`
Maximum: `30000` (30 seconds)
Default: `0`

VL805

Compute Module 4 only.

If the `VL805` property is set to `1` then the bootloader will search for a VL805 PCIe XHCI controller and attempt to initialise it with VL805 firmware embedded in the bootloader EEPROM. This enables industrial designs to use VL805 XHCI controllers without providing a dedicated SPI EEPROM for the VL805 firmware.

- On Compute Module 4 the bootloader never writes to the dedicated VL805 SPI EEPROM. This option just configures the controller to load the firmware from SDRAM.
- Do not use this option if the VL805 XHCI controller has a dedicated EEPROM. It will fail to initialise because the VL805 ROM will attempt to use a dedicated SPI EEPROM if fitted.
- The embedded VL805 firmware assumes the same USB configuration as Raspberry Pi 4B (2 USB 3.0 ports and 4 USB 2.0 ports). There is no support for loading alternate VL805 firmware images, a dedicated VL805 SPI EEPROM should be used instead for such configurations.

Default: `0`

XHCI_DEBUG

This property is a bit-field which controls the verbosity of USB debug messages for mass storage boot-mode. Enabling all of these messages generates a huge amount of log data which will slow down booting and may even cause boot to fail. For verbose logs it’s best to use `NETCONSOLE`.

Value	Log
0x1	USB descriptors
0x2	Mass storage mode state machine
0x4	Mass storage mode state machine - verbose
0x8	All USB requests
0x10	Device and hub state machines

Value	Log
0x20	All xHCI TRBs (VERY VERBOSE)
0x40	All xHCI events (VERY VERBOSE)

To combine values, add them together. For example:

```
# Enable mass storage and USB descriptor logging
XHCI_DEBUG=0x3
```

Default: `0x0` (no USB debug messages enabled)

config.txt section

After reading `config.txt` the GPU firmware `start4.elf` reads the bootloader EEPROM config and checks for a section called `[config.txt]`. If the `[config.txt]` section exists then the contents from the start of this section to the end of the file is appended in memory, to the contents of the `config.txt` file read from the boot partition. This can be used to automatically apply settings to every operating system, for example, dtoverlays.

WARNING

If an invalid configuration which causes boot to fail is specified then the bootloader EEPROM will have to be re-flashed.

Configuration Properties in `config.txt`

boot_ramdisk

If this property is set to `1` then the bootloader will attempt load a ramdisk file called `boot.img` containing the boot boot file-system. Subsequent files (e.g. `start4.elf`) are read from the ramdisk instead of the original boot file-system.

The primary purpose of `boot_ramdisk` is to support `secure-boot`, however, unsigned `boot.img` files can also be useful to Network Boot or `RPIBOOT` configurations.

- The maximum size for a ramdisk file is 96MB.
- `boot.img` files are raw disk `.img` files. The recommended format is a plain FAT32 partition with no MBR.
- The memory for the ramdisk filesystem is released before the operating system is started.
- If TRYBOOT is selected then the bootloader will search for `tryboot.img` instead of `boot.img`.
- See also autoboot.txt

For more information about `secure-boot` and creating `boot.img` files please see USBBOOT

Default: `0`

boot_load_flags

Experimental property for custom firmware (bare metal).

Bit 0 (0x1) indicates that the .elf file is custom firmware. This disables any compatibility checks (e.g. is USB MSD boot supported) and resets PCIe before starting the executable.

Default: `0x0`

uart_2ndstage

If `uart_2ndstage` is `1` then enable debug logging to the UART. This option also automatically enables UART logging in `start.elf`. This is also described on the Boot options page.

The `BOOT_UART` property also enables bootloader UART logging but does not enable UART logging in `start.elf` unless `uart_2ndstage=1` is also set.

Default: `0`

erase_eeprom

If `erase_eeprom` is set to `1` then `recovery.bin` will erase the entire SPI EEPROM instead of flashing the bootloader image. This property has no effect during a normal boot.

Default: `0`

eeprom_write_protect

Configures the EEPROM `Write Status Register`. This can be set to either mark the entire EEPROM as write-protected or clear write-protection.

This option must be used in conjunction with the EEPROM `/WP` pin which controls updates to the EEPROM `Write Status Register`. Pulling `/WP` low (CM4 `EEPROM_nEP` or Pi4B `TP5`) does NOT write-protect the EEPROM unless the `Write Status Register` has also been configured.

See the Winbond W25x40cl datasheet for further details.

`eeprom_write_protect` settings in `config.txt` for `recovery.bin`.

Value	Description
1	Configures the write protect regions to cover the entire EEPROM.
0	Clears the write protect regions.
-1	Do nothing.

NOTE

`flashrom` does not support clearing of the write-protect regions and will fail to update the EEPROM if write-protect regions are defined.

Default: `-1`

bootloader_update

This option may be set to 0 to block self-update without requiring the EEPROM configuration to be updated. This is sometimes useful when updating multiple Raspberry Pis via network boot because this option can be controlled per Raspberry Pi (e.g. via a serial number filter in `config.txt`).

Default: `1`

Secure Boot configuration properties in `config.txt`

The following `config.txt` properties are used to program the `secure-boot` OTP settings. These changes are irreversible and can only be programmed via `RPIBOOT` when flashing the bootloader EEPROM image. This ensures that `secure-boot` cannot be set remotely or by accidentally inserting a stale SD card image.

For more information about enabling `secure-boot` please see the secure-boot readme and the secure-boot tutorial in the USBBOOT repo.

program_pubkey

If this property is set to `1` then `recovery.bin` will write the hash of the public key in the EEPROM image to OTP. Once set, the bootloader will reject EEPROM images signed with different RSA keys or unsigned images.

Default: `0`

revoke_devkey

If this property is set to `1` then `recovery.bin` will write a value to OTP that prevents the ROM from loading old versions of the second stage bootloader which do not support `secure-boot`. This prevents `secure-boot` from being turned off by reverting to an older release of the bootloader.

Default: `0`

program_rpiboot_gpio

Since there is no dedicated `nRPIBOOT` jumper on Raspberry Pi 4B or Raspberry Pi 400, an alternative GPIO must be used to select `RPIBOOT` mode by pulling the GPIO low. Only one GPIO may be selected and the available options are `2, 4, 5, 7, 8`. This property does not depend on `secure-boot` but please verify that this GPIO configuration does not conflict with any HATs which might pull the GPIO low during boot.

Since for safety this property can only be programmed via `RPIBOOT`, the bootloader EEPROM must first be cleared using `erase_eeprom`. This causes the BCM2711 ROM to failover to `RPIBOOT` mode, which then allows this option to be set.

Default: ``

program_jtag_lock

If this property is set to `1` then `recovery.bin` will program an OTP value that prevents VideoCore JTAG from being used. This option requires that `program_pubkey` and `revoke_devkey` are also set. This option can prevent failure-analysis and should only be set after the device has been fully tested.

Default: `0`

Updating to the LATEST / STABLE bootloader

The DEFAULT version of the bootloader is only updated for CRITICAL fixes and major releases. The LATEST / STABLE bootloader is updated more often to include the latest fixes and improvements.

Advanced users can switch to the LATEST / STABLE bootloader to get the latest functionality. Open a command prompt and start `raspi-config`.

```
sudo raspi-config
```

Navigate to `Advanced Options` and then `Bootloader Version`. Select `Latest` and choose `Yes` to confirm. Select `Finish` and confirm you want to reboot. After the reboot, open a command prompt again and update your system.

```
sudo apt update
sudo apt install rpi-eeprom  # Update rpi-eeprom to the latest version
```

If you run `rpi-eeprom-update`, you should see that a more recent version of the bootloader is available and it's the `stable` release.

```
*** UPDATE AVAILABLE ***
BOOTLOADER: update available
   CURRENT: Tue 25 Jan 14:30:41 UTC 2022 (1643121041)
    LATEST: Thu 10 Mar 11:57:12 UTC 2022 (1646913432)
   RELEASE: stable (/lib/firmware/raspberrypi/bootloader/stable)
            Use raspi-config to change the release.
```

Now you can update your bootloader.

```
sudo rpi-eeprom-update -a
sudo reboot
```

If you run `rpi-eeprom-update` again after your Raspberry Pi has rebooted, you should now see that the `CURRENT` date has updated to indicate that you are using the latest version of the bootloader.

```
BOOTLOADER: up to date
   CURRENT: Thu 10 Mar 11:57:12 UTC 2022 (1646913432)
    LATEST: Thu 10 Mar 11:57:12 UTC 2022 (1646913432)
   RELEASE: stable (/lib/firmware/raspberrypi/bootloader/stable)
            Use raspi-config to change the release.
```

USB Boot Modes

Edit this on GitHub

WARNING

The default way of using a Raspberry Pi is to boot it using an SD card. This is the recommended method for new and inexperienced users.

There are two separate boot modes for USB; USB device boot and USB host boot.

The choice between the two boot modes is made by the firmware at boot time when it reads the OTP bits. There are two bits to control USB boot: the first enables USB device boot and is enabled by default. The second enables USB host boot; if the USB host boot mode bit is set, then the processor reads the OTGID pin to decide whether to boot as a host (driven to zero as on any Raspberry Pi Model B / B+) or as a device (left floating). The Raspberry Pi Zero has access to this pin through the OTGID pin on the USB connector, and the Compute Module has access to this pin on the edge connector.

There are also OTP bits that allow certain GPIO pins to be used for selecting which boot modes the Raspberry Pi should attempt to use.

NOTE

USB boot modes only available on certain models.

USB Device Boot Mode

NOTE

Device boot is available on Raspberry Pi Compute Module, Compute Module 3, Raspberry Pi Zero, Zero W, A, A+, and 3A+ only.

When this boot mode is activated (usually after a failure to boot from the SD card), the Raspberry Pi puts its USB port into device mode and awaits a USB reset from the host. Example code showing how the host needs to talk to the Raspberry Pi can be found on Github.

The host first sends a structure to the device down control endpoint 0. This contains the size and signature for the boot (security is not enabled, so no signature is required). Secondly, code is transmitted down endpoint 1 (bootcode.bin). Finally, the device will reply with a success code of:

- 0 - Success
- 0x80 - Failed

USB Host Boot Mode

NOTE

Host boot is available on Raspberry Pi 3B, 3B+, 3A+, and 2B v1.2 only. Raspberry Pi 3A+ only supports mass storage boot, not network boot.

The USB host boot mode follows this sequence:

- Enable the USB port and wait for D+ line to be pulled high indicating a USB 2.0 device (we only support USB2.0)
- If the device is a hub:

    - Enable power to all downstream ports of the hub
    - For each port, loop for a maximum of two seconds (or five seconds if `program_usb_boot_timeout=1` has been set)
        - Release from reset and wait for D+ to be driven high to indicate that a device is connected
        - If a device is detected:
            - Send "Get Device Descriptor"
                - If VID == SMSC && PID == 9500
                    - Add device to Ethernet device list
            - If class interface == mass storage class
                - Add device to mass storage device list
- Else
    - Enumerate single device
- Go through mass storage device list
    - Boot from mass storage device
- Go through Ethernet device list
    - Boot from Ethernet

USB Mass Storage Boot

Edit this on GitHub

NOTE

Available on Raspberry Pi 2B v1.2, 3A+, 3B, 3B+, 4B, 400 and Zero 2 W, and Raspberry Pi Compute Module 3, 3+ and 4 only.

This page explains how to boot your Raspberry Pi from a USB mass storage device such as a flash drive or a USB hard disk. When attaching USB devices, particularly hard disks and SSDs, be mindful of their power requirements. If you wish to attach more than one SSD or hard disk to the Raspberry Pi, this normally requires external power - either a powered hard disk enclosure, or a powered USB hub. Note that models prior to the Raspberry Pi 4B have known issues which prevent booting with some USB devices.

Raspberry Pi 4B and Raspberry Pi 400

The bootloader in Raspberry Pi 400 and newer Raspberry Pi 4B boards support USB boot by default, although the `BOOT_ORDER` bootloader configuration may need to be modified. On earlier Raspberry Pi 4B boards, or to select alternate boot modes, the bootloader must be updated.

See:-

- Instructions for changing the boot mode via the Raspberry Pi Imager.
- Instructions for changing the boot mode via the raspi-config.

- The bootloader configuration page for other boot configuration options.

Compute Module 4

Please see the Flashing the Compute Module eMMC for bootloader update instructions.

Raspberry Pi 3B+

The Raspberry Pi 3B+ supports USB mass storage boot out of the box.

Raspberry Pi 2B, 3A+, 3B, CM3, CM3+, Zero 2 W

On the Raspberry Pi 2B v1.2, 3A+, 3B, Zero 2 W, and Compute Module 3, 3+ you must first enable USB host boot mode. This is to allow USB mass storage boot, and network boot. Note that network boot is not supported on the Raspberry Pi 3A+ or Zero 2 W.

To enable USB host boot mode, the Raspberry Pi needs to be booted from an SD card with a special option to set the USB host boot mode bit in the one-time programmable (OTP) memory. Once this bit has been set, the SD card is no longer required.

IMPORTANT

Any change you make to the OTP is permanent and cannot be undone.

NOTE

On the Raspberry Pi 3A+, setting the OTP bit to enable USB host boot mode will permanently prevent that Raspberry Pi from booting in USB device mode.

You can use any SD card running Raspberry Pi OS to program the OTP bit.

Enable USB host boot mode with this code:

```
echo program_usb_boot_mode=1 | sudo tee -a /boot/config.txt
```

This adds `program_usb_boot_mode=1` to the end of `/boot/config.txt`.

Note that although the option is named `program_usb_boot_mode`, it only enables USB *host* boot mode. USB *device* boot mode is only available on certain models of Raspberry Pi - see USB device boot mode.

The next step is to reboot the Raspberry Pi with `sudo reboot` and check that the OTP has been programmed with:

```
vcgencmd otp_dump | grep 17:
17:3020000a
```

Check that the output `0x3020000a` is shown. If it is not, then the OTP bit has not been successfully programmed. In this case, go through the programming procedure again. If the bit is still not set, this may indicate a fault in the Raspberry Pi hardware itself.

If you wish, you can remove the `program_usb_boot_mode` line from `config.txt`, so that if you put the SD card into another Raspberry Pi, it won't program USB host boot mode.

Make sure there is no blank line at the end of `config.txt`.
You can now boot from a USB mass storage device in the same way as booting from an SD card - see the following section for further information.

Booting from USB Mass Storage

The procedure is the same as for SD cards - simply image the USB storage device with the operating system image.

After preparing the storage device, connect the drive to the Raspberry Pi and power up the Raspberry Pi, being aware of the extra USB power requirements of the external drive. After five to ten seconds, the Raspberry Pi should begin booting and show the rainbow splash screen on an attached display. Make sure that you do not have an SD card inserted in the Raspberry Pi, since if you do, it will boot from that first.

See the bootmodes documentation for the boot sequence and alternative boot modes (network, USB device, GPIO or SD boot).

Known Issues

IMPORTANT

These do **not** apply to Raspberry Pi 4 Model B.

- The default timeout for checking bootable USB devices is 2 seconds. Some flash drives and hard disks power up too slowly. It is possible to extend this timeout to five seconds (add a new file `timeout` to the SD card), but note that some devices take even longer to respond.
- Some flash drives have a very specific protocol requirement that is not handled by the bootcode and may thus be incompatible.

Special `bootcode.bin`-only Boot Mode

IMPORTANT

This does **not** apply to Raspberry Pi 4 Model B.

If you are unable to use a particular USB device to boot your Raspberry Pi, an alternative for the Raspberry Pi 2B v1.2, 3A+, 3B and 3B+ is to use the special bootcode.bin-only boot mode. The Raspberry Pi will still boot from the SD card, but `bootcode.bin` is the only file read from it.

Hardware Compatibility

Before attempting to boot from a USB mass storage device it is advisable to verify that the device works correctly under Linux. Boot using an SD card and plug in the USB mass storage device. This should appears as a removable drive. This is especially important with USB SATA adapters which may be supported by the bootloader in mass storage mode but fail if Linux selects USB Attached SCSI - UAS mode. See this forum thread about UAS and how to add usb-storage.quirks to workaround this issue.

Spinning hard disk drives nearly always require a powered USB hub. Even if it appears to work, you are likely to encounter intermittent failures without a powered USB HUB.

Multiple Bootable Drives

When searching for a bootable partition, the bootloader scans all USB mass storage devices in parallel and will select the first to respond. If the boot partition does not contain a suitable `start.elf` file, the next available device is selected. There is no method for specifying the boot device according to the USB topology because this would slow down boot and adds unnecessary and hard to support configuration complexity.

NOTE

The `config.txt` file conditional filters can be used to select alternate firmware in complex device configurations.

Network Booting

Edit this on GitHub

This section describes how network booting works on the Raspberry Pi 3B, 3B+ and 2B v1.2. On the Raspberry Pi 4 network booting is implemented in the second stage bootloader in the EEPROM. Please see the Raspberry Pi 4 Bootloader Configuration page for more information. We also have a tutorial about setting up a network boot system. Network booting works only for the wired adapter built into the above models of Raspberry Pi. Booting over wireless LAN is not supported, nor is booting from any other wired network device.

To network boot, the boot ROM does the following:

- Initialise on-board Ethernet device (Microchip LAN9500 or LAN7500)
- Send DHCP request (with Vendor Class identifier DHCP option 60 set to 'PXEClient:Arch:00000:UNDI:002001')
- Receive DHCP reply
- (optional) Receive DHCP proxy reply
- ARP to tftpboot server
- ARP reply includes tftpboot server ethernet address
- TFTP RRQ 'bootcode.bin'
    - File not found: Server replies with TFTP error response with textual error message
    - File exists: Server will reply with the first block (512 bytes) of data for the file with a block number in the header
        - Raspberry Pi replies with TFTP ACK packet containing the block number, and repeats until the last block which is not 512 bytes
- TFTP RRQ 'bootsig.bin'
    - This will normally result in an error `file not found`. This is to be expected, and TFTP boot servers should be able to handle it.

From this point the `bootcode.bin` code continues to load the system. The first file it will try to access is [`serial_number`]/start.elf. If this does not result in an error then any other files to be read will be pre-pended with the `serial_number`. This is useful because it enables you to create separate directories with separate start.elf / kernels for your Raspberry Pis. To get the serial number for the device you can either try this boot mode and see what file is accessed using tcpdump / wireshark, or you can run a standard Raspberry Pi OS SD card and `cat /proc/cpuinfo`.

If you put all your files into the root of your tftp directory then all following files will be accessed from there.

Debugging Network Boot Mode

The first thing to check is that the OTP bit is correctly programmed. To do this, you need to add `program_usb_boot_mode=1` to config.txt and reboot (with a standard SD card that boots correctly into Raspberry Pi OS). Once you've done this, you should be able to do:

```
 vcgencmd otp_dump | grep 17:
```

If row 17 contains `3020000a` then the OTP is correctly programmed. You should now be able to remove the SD card, plug in Ethernet, and then the Ethernet LEDs should light up around 5 seconds after the Raspberry Pi powers up.

To capture the ethernet packets on the server, use tcpdump on the tftpboot server (or DHCP server if they are different). You will need to capture the packets there otherwise you will not be able to see packets that get sent directly because network switches are not hubs!

```
sudo tcpdump -i eth0 -w dump.pcap
```

This will write everything from eth0 to a file dump.pcap you can then post process it or upload it to cloudshark.com for communication

DHCP Request / Reply

As a minimum you should see a DHCP request and reply which looks like the following:

```
6:44:38.717115 IP (tos 0x0, ttl 128, id 0, offset 0, flags [none], proto
UDP (17), length 348)
    0.0.0.0.68 > 255.255.255.255.67: [no cksum] BOOTP/DHCP, Request from
b8:27:eb:28:f6:6d, length 320, xid 0x26f30339, Flags [none] (0x0000)
          Client-Ethernet-Address b8:27:eb:28:f6:6d
          Vendor-rfc1048 Extensions
            Magic Cookie 0x63825363
            DHCP-Message Option 53, length 1: Discover
            Parameter-Request Option 55, length 12:
              Vendor-Option, Vendor-Class, BF, Option 128
              Option 129, Option 130, Option 131, Option 132
              Option 133, Option 134, Option 135, TFTP
            ARCH Option 93, length 2: 0
            NDI Option 94, length 3: 1.2.1
            GUID Option 97, length 17: 0.68.68.68.68.68.68.68.68.68.68.6
8.68.68.68.68.68
            Vendor-Class Option 60, length 32: "PXEClient:Arch:00000:UND
I:002001"
            END Option 255, length 0
16:44:41.224619 IP (tos 0x0, ttl 64, id 57713, offset 0, flags [none], pr
oto UDP (17), length 372)
    192.168.1.1.67 > 192.168.1.139.68: [udp sum ok] BOOTP/DHCP, Reply, le
```

```
ngth 344, xid 0x26f30339, Flags [none] (0x0000)
          Your-IP 192.168.1.139
          Server-IP 192.168.1.1
          Client-Ethernet-Address b8:27:eb:28:f6:6d
          Vendor-rfc1048 Extensions
            Magic Cookie 0x63825363
            DHCP-Message Option 53, length 1: Offer
            Server-ID Option 54, length 4: 192.168.1.1
            Lease-Time Option 51, length 4: 43200
            RN Option 58, length 4: 21600
            RB Option 59, length 4: 37800
            Subnet-Mask Option 1, length 4: 255.255.255.0
            BR Option 28, length 4: 192.168.1.255
            Vendor-Class Option 60, length 9: "PXEClient"
            GUID Option 97, length 17: 0.68.68.68.68.68.68.68.68.68.68.6
8.68.68.68.68.68
            Vendor-Option Option 43, length 32: 6.1.3.10.4.0.80.88.69.9.2
0.0.0.17.82.97.115.112.98.101.114.114.121.32.80.105.32.66.111.111.116.255
            END Option 255, length 0
```

The important part of the reply is the Vendor-Option Option 43. This needs to contain the string "Raspberry Pi Boot", although, due to a bug in the boot ROM, you may need to add three spaces to the end of the string.

TFTP file read

You will know whether the Vendor Option is correctly specified: if it is, you'll see a subsequent TFTP RRQ packet being sent. RRQs can be replied to by either the first block of data or an error saying file not found. In a couple of cases they even receive the first packet and then the transmission is aborted by the Raspberry Pi (this happens when checking whether a file exists). The example below is just three packets: the original read request, the first data block (which is always 516 bytes containing a header and 512 bytes of data, although the last block is always less than 512 bytes and may be zero length), and the third packet (the ACK which contains a frame number to match the frame number in the data block).

```
16:44:41.224964 IP (tos 0x0, ttl 128, id 0, offset 0, flags [none], proto
UDP (17), length 49)
    192.168.1.139.49152 > 192.168.1.1.69: [no cksum]  21 RRQ "bootcode.bi
n" octet
16:44:41.227223 IP (tos 0x0, ttl 64, id 57714, offset 0, flags [none], pr
oto UDP (17), length 544)
    192.168.1.1.55985 > 192.168.1.139.49152: [udp sum ok] UDP, length 516
16:44:41.227418 IP (tos 0x0, ttl 128, id 0, offset 0, flags [none], proto
UDP (17), length 32)
    192.168.1.139.49152 > 192.168.1.1.55985: [no cksum] UDP, length 4
```

Known Problems

There are a number of known problems with the Ethernet boot mode. Since the implementation of the boot modes is in the chip itself, there are no workarounds other than to use an SD card with just the bootcode.bin file.

DHCP requests time out after five tries

The Raspberry Pi will attempt a DHCP request five times with five seconds in between, for a total period of 25 seconds. If the server is not available to respond in this time, then the Raspberry Pi will drop into a low-power state. There is no workaround for this other than bootcode.bin on an SD card.

TFTP server on separate subnet not supported

Fixed in Raspberry Pi 3 Model B+ (BCM2837B0).

DHCP relay broken

The DHCP check also checked if the hops value was `1`, which it wouldn't be with DHCP relay.

Fixed in Raspberry Pi 3 Model B+.

Raspberry Pi Boot string

The "Raspberry Pi Boot " string in the DHCP reply requires the extra three spaces due to an error calculating the string length.

Fixed in Raspberry Pi 3 Model B+

DHCP UUID constant

The DHCP UUID is set to be a constant value.

Fixed in Raspberry Pi 3 Model B+; the value is set to the 32-bit serial number.

ARP check can fail to respond in the middle of TFTP transaction

The Raspberry Pi will only respond to ARP requests when it is in the initialisation phase; once it has begun transferring data, it'll fail to continue responding.

Fixed in Raspberry Pi 3 Model B+.

DHCP request/reply/ack sequence not correctly implemented

At boot time, Raspberry Pi broadcasts a DHCPDISCOVER packet. The DHCP server replies with a DHCPOFFER packet. The Raspberry Pi then continues booting without doing a DHCPREQUEST or waiting for DHCPACK. This may result in two separate devices being offered the same IP address and using it without it being properly assigned to the client.

Different DHCP servers have different behaviours in this situation. dnsmasq (depending upon settings) will hash the MAC address to determine the IP address, and ping the IP address to make sure it isn't already in use. This reduces the chances of this happening because it requires a collision in the hash.

GPIO Boot Mode

Edit this on GitHub

NOTE

GPIO boot mode is only available on the Raspberry Pi 3A+, 3B, 3B+, Compute Module 3 and 3+.

The Raspberry Pi can be configured to allow the boot mode to be selected at power on using hardware attached to the GPIO connector: this is GPIO boot mode. This is done by setting bits in the OTP memory of the SoC. Once the bits are set, they permanently allocate 5 GPIOs to allow this selection to be made. Once the OTP bits are set they cannot be unset so you should think carefully about enabling this, since those 5 GPIO lines will always control booting. Although you can use the GPIOs for some other function once the Raspberry Pi has booted, you must set them up so that they enable the desired boot modes when the Raspberry Pi boots.

To enable GPIO boot mode, add the following line to the `config.txt` file:

```
program_gpio_bootmode=n
```

Where n is the bank of GPIOs which you wish to use. Then reboot the Raspberry Pi once to program the OTP with this setting. Bank 1 is GPIOs 22-26, bank 2 is GPIOs 39-43. Unless you have a Compute Module, you must use bank 1: the GPIOs in bank 2 are only available on the Compute Module. Because of the way the OTP bits are arranged, if you first program GPIO boot mode for bank 1, you then have the option of selecting bank 2 later. The reverse is not true: once bank 2 has been selected for GPIO boot mode, you cannot select bank 1.

Once GPIO boot mode is enabled, the Raspberry Pi will no longer boot. You must pull up at least one boot mode GPIO pin in order for the Raspberry Pi to boot.

Pin Assignments

Raspberry Pi 3B and Compute Module 3

Bank 1	Bank 2	boot type
22	39	SD0
23	40	SD1
24	41	NAND (no Linux support at present)
25	42	SPI (no Linux support at present)
26	43	USB

USB in the table above selects both USB device boot mode and USB host boot mode. In order to use a USB boot mode, it must be enabled in the OTP memory. For more information, see USB device boot and USB host boot.

Newer Raspberry Pi 3B (BCM2837B0 with the metal lid), Raspberry Pi 3A+, 3B+ and Compute Module 3+

Bank 1	Bank 2	boot type
20	37	SD0
21	38	SD1
22	39	NAND (no Linux support at present)

Bank 1	Bank 2	boot type
23	40	SPI (no Linux support at present)
24	41	USB device
25	42	USB host - mass storage device
26	43	USB host - ethernet

NOTE

The various boot modes are attempted in the numerical order of the GPIO lines, i.e. SD0, then SD1, then NAND and so on.

Boot Flow

SD0 is the Broadcom SD card / MMC interface. When the boot ROM within the SoC runs, it always connects SD0 to the built-in microSD card slot. On Compute Modules with an eMMC device, SD0 is connected to that; on the Compute Module Lite SD0 is available on the edge connector and connects to the microSD card slot in the CMIO carrier board. SD1 is the Arasan SD card / MMC interface which is also capable of SDIO. All Raspberry Pi models with built-in wireless LAN use SD1 to connect to the wireless chip via SDIO.

The default pull resistance on the GPIO lines is 50K ohm, as documented on page 102 of the BCM2835 ARM peripherals datasheet. A pull resistance of 5K ohm is recommended to pull a GPIO line up: this will allow the GPIO to function but not consume too much power.

NVMe SSD Boot

Edit this on GitHub

NVMe (non-volatile memory express) is a standard for accessing solid state drives (SSDs) via a PCIe bus. You can connect these drives via the PCIe slot on a Compute Module 4 (CM4) IO board, allowing a CM4 to boot from SSD.

Required Hardware

You need an NVMe M.2 SSD. You cannot plug an M.2 SSD directly into the PCIe slot on the IO board - an adaptor is needed. Be careful to get the correct type: a suitable adaptor can be found online by searching for 'PCI-E 3.0 x1 Lane to M.2 NGFF M-Key SSD Nvme PCI Express Adapter Card'.

The latest version of Raspberry Pi OS supports booting from NVMe drives. To check that your NVMe drive is connected correctly, boot Raspberry Pi OS from another drive and run `ls -l /dev/nvme*`; example output is shown below.

```
crw------- 1 root root 245, 0 Mar  9 14:58 /dev/nvme0
brw-rw---- 1 root disk 259, 0 Mar  9 14:58 /dev/nvme0n1
```

If you need to connect the NVMe drive to a PC or Mac you can use a USB adaptor: search for 'NVME PCI-E M-Key Solid State Drive External Enclosure'. The enclosure must support M key SSDs.

Required Software

To boot from NVMe you need a recent version of the bootloader (after July 2021), and a recent version of the VideoCore firmware and Raspberry Pi OS Linux kernel. The latest Raspberry Pi OS release has everything you need, so you can use the Raspberry Pi Imager to install the software to your SSD.

Bootloader

You might need to use `rpiboot` to update the CM4 bootloader. Instructions for building `rpiboot` and configuring the IO board to switch the ROM to usbboot mode are in the usbboot Github repository.

Remember to add the NVMe boot mode `6` to BOOT_ORDER in `recovery/boot.conf`.

Firmware and kernel

You must have the latest versions of the VideoCore firmware and Raspberry Pi OS Linux kernel to boot directly from an NVMe SSD disk. The Raspberry Pi Bullseye and Buster Legacy releases have everything needed.

If you are using CM4 lite, remove the SD card and the board will boot from the NVMe disk. For versions of CM4 with an eMMC, make sure you have set NVMe first in the boot order.

NVMe BOOT_ORDER

This boot behaviour is controlled via the `BOOT_ORDER` setting in the EEPROM configuration: we have added a new boot mode `6` for NVMe. See Raspberry Pi 4 Bootloader Configuration.

Below is an example of UART output when the bootloader detects the NVMe drive:

```
Boot mode: SD (01) order f64
Boot mode: USB-MSD (04) order f6
Boot mode: NVME (06) order f
VID 0x144d MN Samsung SSD 970 EVO Plus 250GB
NVME on
```

It will then find a FAT partition and load `start4.elf`:

```
Read start4.elf bytes  2937840 hnd 0x00050287 hash ''
```

It will then load the kernel and boot the OS:

```
MESS:00:00:07.096119:0: brfs: File read: /mfs/sd/kernel8.img
MESS:00:00:07.098682:0: Loading 'kernel8.img' to 0x80000 size 0x1441a00
MESS:00:00:07.146055:0:[    0.000000] Booting Linux on physical CPU 0x000
0000000 [0x410fd083]
```

In Linux the SSD appears as `/dev/nvme0` and the "namespace" as `/dev/nvme0n1`. There will be two partitions `/dev/nvme0n1p1` (FAT) and `/dev/nvme0n1p2` (EXT4). Use `lsblk` to check the partition assignments:

```
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
nvme0n1     259:0    0 232.9G  0 disk
├─nvme0n1p1 259:1    0   256M  0 part /boot
└─nvme0n1p2 259:2    0 232.6G  0 part /
```

Troubleshooting

If the boot process fails, please file an issue on the rpi-eeprom Github repository, including a copy of the console and anything displayed on the screen during boot.

HTTP Boot

Edit this on GitHub

The network install feature uses HTTP over ethernet to boot the Raspberry Pi into the embedded Raspberry Pi Imager application.

In addition to network install, you can explicitly boot your device with files downloaded via HTTP with boot-mode `7`. You can still use this even if network install on boot is disabled.

You could for example add this to your `BOOT_ORDER` as a fall-back boot method, or put it behind a GPIO conditional to initiate HTTP boot from your own server when a GPIO pin is pulled low.

For example, if you added the following to your eeprom config and GPIO 8 (which has a default state of 1 or HIGH) were to be pulled low, the files `http://downloads.raspberrypi.org:80/net_install/boot.img` and `http://downloads.raspberrypi.org:80/net_install/boot.sig` would be downloaded. If network install on boot were enabled it would use the same URL. If GPIO 8 were not pulled low the behaviour would be unchanged.

```
[gpio8=0]
BOOT_ORDER=0xf7
HTTP_HOST=downloads.raspberrypi.org
NET_INSTALL_ENABLED=0
```

HTTP in the `BOOT_ORDER` will be ignored if secure boot is enabled and HTTP_HOST is not set.

Requirements

To use HTTP boot you need to use the LATEST / STABLE bootloader configuration and update to a bootloader dated 10th March 2022 or later. HTTP boot only works over ethernet, so you need to connect your Raspberry Pi to your network via an Ethernet cable, e.g. to a socket on the back of your router.

Keys

All HTTP downloads must be signed. The bootloader includes a public key for the files on the default host `fw-download-alias1.raspberrypi.com`. This key will be used to verify the network install image **unless** you set HTTP_HOST **and** include a public key in the eeprom. This allows you to host the Raspberry Pi network install images on your own server.

WARNING

Using your own network install image will require you to sign the image and add your public key to the eeprom. At the moment, if you then apply a public eeprom update, your key will be lost and will need to be re-added.

USBBOOT has all the tools needed to program public keys. You would do something like this.

```
# Add your PUBLIC key to the eeprom. boot.conf contains your modifications
rpi-eeprom-config -c boot.conf -p mypubkey.pem -o pieeprom.upd pieeprom.original.bin

# Generate signature for your eeprom
rpi-eeprom-digest -i pieeprom.upd -o pieeprom.sig

# Sign the network install image with your PRIVATE key
# Put boot.img and boot.sig on your web server
rpi-eeprom-digest -i boot.img -o boot.sig -k myprivkey.pem
```

Secure Boot

If secure boot is enabled then the Raspberry Pi can only run code signed by the customer's private key. So if you want to use network install or HTTP boot mode with secure boot you must sign `boot.img` and generate `boot.sig` with your own key and host these files somewhere for download. The public key in the eeprom will be used to verify the image.

If secure boot is enabled and HTTP_HOST is not set, then network install and HTTP boot will be disabled.

For more information about secure boot see USBBOOT.

Parallel Display Interface (DPI)

Edit this on GitHub

An up-to-24-bit parallel RGB interface is available on all Raspberry Pi boards with the 40 way header and the Compute Modules. This interface allows parallel RGB displays to be attached to the Raspberry Pi GPIO either in RGB24 (8 bits for red, green and blue) or RGB666 (6 bits per colour) or RGB565 (5 bits red, 6 green, and 5 blue).

This interface is controlled by the GPU firmware and can be programmed by a user via special config.txt parameters and by enabling the correct Linux Device Tree overlay.

GPIO Pins

One of the alternate functions selectable on bank 0 of the Raspberry Pi GPIO is DPI (Display Parallel Interface) which is a simple clocked parallel interface (up to 8 bits of R, G and B; clock, enable, hsync, and vsync). This interface is available as alternate function 2 (ALT2) on GPIO bank 0:

GPIO	ALT2
GPIO0	PCLK

GPIO	ALT2
GPIO1	DE
GPIO2	LCD_VSYNC
GPIO3	LCD_HSYNC
GPIO4	DPI_D0
GPIO5	DPI_D1
GPIO6	DPI_D2
GPIO7	DPI_D3
GPIO8	DPI_D4
GPIO9	DPI_D5
GPIO10	DPI_D6
GPIO11	DPI_D7
GPIO12	DPI_D8
GPIO13	DPI_D9
GPIO14	DPI_D10
GPIO15	DPI_D11
GPIO16	DPI_D12
GPIO17	DPI_D13
GPIO18	DPI_D14
GPIO19	DPI_D15
GPIO20	DPI_D16
GPIO21	DPI_D17
GPIO22	DPI_D18
GPIO23	DPI_D19
GPIO24	DPI_D20
GPIO25	DPI_D21
GPIO26	DPI_D22
GPIO27	DPI_D23

NOTE

There are various ways that the colour values can be presented on the DPI output pins in either 565, 666, or 24-bit modes (see the following table and the `output_format` part of the `dpi_output_format` parameter below):

Mode	RGB bits	GPIO																		
		27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	565	-	-	-	-	-	-	-	-	7	6	5	4	3	7	6	5	4	3	2	7	6

3	565	-	-	-	7	6	5	4	3	-	-	7	6	5	4	3	2	-	-	-	7	6
4	565	-	-	7	6	5	4	3	-	-	-	7	6	5	4	3	2	-	-	7	6	5
5	666	-	-	-	-	-	-	7	6	5	4	3	2	7	6	5	4	3	2	7	6	5
6	666	-	-	7	6	5	4	3	2	-	-	7	6	5	4	3	2	-	-	7	6	5
7	888	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3

Disable Other GPIO Peripherals

Note that all other peripheral overlays that use conflicting GPIO pins must be disabled. In config.txt, take care to comment out or invert any dtparams that enable I2C or SPI:

```
dtparam=i2c_arm=off
dtparam=spi=off
```

Controlling Output Format

The output format (clock, colour format, sync polarity, enable) can be controlled with a magic number (unsigned integer or hex value prefixed with 0x) passed to the `dpi_output_format` parameter in config.txt created from the following fields:

```
output_format          = (dpi_output_format >>  0) & 0xf;
rgb_order              = (dpi_output_format >>  4) & 0xf;

output_enable_mode     = (dpi_output_format >>  8) & 0x1;
invert_pixel_clock     = (dpi_output_format >>  9) & 0x1;

hsync_disable          = (dpi_output_format >> 12) & 0x1;
vsync_disable          = (dpi_output_format >> 13) & 0x1;
output_enable_disable  = (dpi_output_format >> 14) & 0x1;

hsync_polarity         = (dpi_output_format >> 16) & 0x1;
vsync_polarity         = (dpi_output_format >> 17) & 0x1;
output_enable_polarity = (dpi_output_format >> 18) & 0x1;

hsync_phase            = (dpi_output_format >> 20) & 0x1;
vsync_phase            = (dpi_output_format >> 21) & 0x1;
output_enable_phase    = (dpi_output_format >> 22) & 0x1;

output_format:
   1: DPI_OUTPUT_FORMAT_9BIT_666
   2: DPI_OUTPUT_FORMAT_16BIT_565_CFG1
   3: DPI_OUTPUT_FORMAT_16BIT_565_CFG2
   4: DPI_OUTPUT_FORMAT_16BIT_565_CFG3
   5: DPI_OUTPUT_FORMAT_18BIT_666_CFG1
   6: DPI_OUTPUT_FORMAT_18BIT_666_CFG2
   7: DPI_OUTPUT_FORMAT_24BIT_888

rgb_order:
   1: DPI_RGB_ORDER_RGB
   2: DPI_RGB_ORDER_BGR
   3: DPI_RGB_ORDER_GRB
   4: DPI_RGB_ORDER_BRG

output_enable_mode:
   0: DPI_OUTPUT_ENABLE_MODE_DATA_VALID
   1: DPI_OUTPUT_ENABLE_MODE_COMBINED_SYNCS

invert_pixel_clock:
   0: RGB Data changes on rising edge and is stable at falling edge
   1: RGB Data changes on falling edge and is stable at rising edge.

hsync/vsync/output_enable_polarity:
   0: default for HDMI mode
   1: inverted
```

```
hsync/vsync/oe phases:
   0: DPI_PHASE_POSEDGE
   1: DPI_PHASE_NEGEDGE
```

NB the single bit fields all act as an "invert default behaviour".

Controlling Timings and Resolutions

In firmware dated August 2018 or later, the `hdmi_timings` config.txt entry that was previously used to set up the DPI timings has be superseded by a new `dpi_timings` parameter. If the `dpi_timings` parameter is not present, the system will fall back to using the `hdmi_timings` parameter to ensure backwards compatibility. If neither are present and a custom mode is requested, then a default set of parameters for VGAp60 is used.

The `dpi_group` and `dpi_mode` config.txt parameters are used to set either predetermined modes (DMT or CEA modes as used by HDMI) or a user can generate custom modes.

If you set up a custom DPI mode, then in config.txt use:

```
dpi_group=2
dpi_mode=87
```

This will tell the driver to use the custom `dpi_timings` (older firmware uses `hdmi_timings`) timings for the DPI panel.

The `dpi_timings` parameters are specified as a space-delimited set of parameters:

```
dpi_timings=<h_active_pixels> <h_sync_polarity> <h_front_porch> <h_sync_p
ulse> <h_back_porch> <v_active_lines> <v_sync_polarity> <v_front_porch> <
v_sync_pulse> <v_back_porch> <v_sync_offset_a> <v_sync_offset_b> <pixel_r
ep> <frame_rate> <interlaced> <pixel_freq> <aspect_ratio>

<h_active_pixels> = horizontal pixels (width)
<h_sync_polarity> = invert hsync polarity
<h_front_porch>   = horizontal forward padding from DE active edge
<h_sync_pulse>    = hsync pulse width in pixel clocks
<h_back_porch>    = vertical back padding from DE active edge
<v_active_lines>  = vertical pixels height (lines)
<v_sync_polarity> = invert vsync polarity
<v_front_porch>   = vertical forward padding from DE active edge
<v_sync_pulse>    = vsync pulse width in pixel clocks
<v_back_porch>    = vertical back padding from DE active edge
<v_sync_offset_a> = leave at zero
<v_sync_offset_b> = leave at zero
<pixel_rep>       = leave at zero
<frame_rate>      = screen refresh rate in Hz
<interlaced>      = leave at zero
<pixel_freq>      = clock frequency (width*height*framerate)
<aspect_ratio>    = *

* The aspect ratio can be set to one of eight values (choose closest for
your screen):

HDMI_ASPECT_4_3 = 1
HDMI_ASPECT_14_9 = 2
HDMI_ASPECT_16_9 = 3
HDMI_ASPECT_5_4 = 4
HDMI_ASPECT_16_10 = 5
HDMI_ASPECT_15_9 = 6
HDMI_ASPECT_21_9 = 7
HDMI_ASPECT_64_27 = 8
```

Overlays

A Linux Device Tree overlay is used to switch the GPIO pins into the correct mode (alt function 2). As previously mentioned, the GPU is responsible for driving the DPI display. Hence there is no Linux driver; the overlay simply sets the GPIO alt functions correctly.

A 'full fat' DPI overlay (dpi24.dtb) is provided which sets all 28 GPIOs to ALT2 mode, providing the full 24 bits of colour bus as well as the h and v-sync, enable and pixel clock. Note this uses **all** of the bank 0 GPIO pins.

A second overlay (vga666.dtb) is provided for driving VGA monitor signals in 666 mode which don't need the clock and DE pins (GPIO 0 and 1) and only require GPIOs 4-21 for colour (using mode 5).

These overlays are fairly trivial and a user can edit them to create a custom overlay to enable just the pins required for their specific use case. For example, if one was using a DPI display using vsync, hsync, pclk, and de but in RGB565 mode (mode 2), then the dpi24.dtb overlay could be edited so that GPIOs 20-27 were not switched to DPI mode and hence could be used for other purposes.

Example `config.txt` Settings

Gert VGA666 adaptor

This setup is for the Gert VGA adaptor.

Note that the instructions provided in the documentation in the above GitHub link are somewhat out of date, so please use the settings below.

```
dtoverlay=vga666
enable_dpi_lcd=1
display_default_lcd=1
dpi_group=2
dpi_mode=82
```

800x480 LCD panel

NOTE

This was tested with Adafruit's DPI add-on board and an 800x480 LCD panel.

```
dtoverlay=dpi24
overscan_left=0
overscan_right=0
overscan_top=0
overscan_bottom=0
framebuffer_width=800
framebuffer_height=480
enable_dpi_lcd=1
display_default_lcd=1
dpi_group=2
dpi_mode=87
dpi_output_format=0x6f005
dpi_timings=800 0 40 48 88 480 0 13 3 32 0 0 0 60 0 32000000 6
```

General Purpose I/O (GPIO)

Edit this on GitHub

General Purpose I/O (GPIO) pins can be configured as either general-purpose input, general-purpose output, or as one of up to six special alternate settings, the functions of which are pin-dependent.

There are three GPIO banks on BCM2835. Each of the three banks has its own VDD input pin. On Raspberry Pi, all GPIO banks are supplied from 3.3V.

WARNING

Connection of a GPIO to a voltage higher than 3.3V will likely destroy the GPIO block within the SoC.

A selection of pins from Bank 0 is available on the P1 header on Raspberry Pi.

GPIO Pads

The GPIO connections on the BCM2835 package are sometimes referred to in the peripherals data sheet as "pads" — a semiconductor design term meaning 'chip connection to outside world'.

The pads are configurable CMOS push-pull output drivers/input buffers. Register-based control settings are available for:

- Internal pull-up / pull-down enable/disable
- Output drive strength
- Input Schmitt-trigger filtering

Power-on states

All GPIO pins revert to general-purpose inputs on power-on reset. The default pull states are also applied, which are detailed in the alternate function table in the ARM peripherals datasheet. Most GPIOs have a default pull applied.

Interrupts

Each GPIO pin, when configured as a general-purpose input, can be configured as an interrupt source to the ARM. Several interrupt generation sources are configurable:

- Level-sensitive (high/low)
- Rising/falling edge
- Asynchronous rising/falling edge

Level interrupts maintain the interrupt status until the level has been cleared by system software (e.g. by servicing the attached peripheral generating the interrupt).

The normal rising/falling edge detection has a small amount of synchronisation built into the detection. At the system clock frequency, the pin is sampled with the criteria for generation of an interrupt being a stable transition within a three-cycle window, i.e. a record of '1 0 0' or '0 1 1'. Asynchronous detection bypasses this synchronisation to enable the detection of very narrow events.

Alternative Functions

Almost all of the GPIO pins have alternative functions. Peripheral blocks internal to the SoC can be selected to appear on one or more of a set of GPIO pins, for example the I2C buses can be configured to at least 3 separate locations. Pad control, such as drive strength or Schmitt filtering, still applies when the pin is configured as an alternate function.

Voltage Specifications

The table below gives the various voltage specifications for the GPIO pins for BCM2835, BCM2836, BCM2837 and RP3A0-based products (e.g. Raspberry Pi Zero or Raspberry Pi 3+). For information about Compute Modules you should see the relevant datasheets.

Symbol	**Parameter**	**Conditions**	**Min**	**Typical**	**Max**	**Unit**
V_{IL}	Input Low Voltage	-	-	-	0.9	V
V_{IH}	Input high voltage[a]	-	1.6	-	-	V
I_{IL}	Input leakage current	TA = +85○C	-	-	5	µA
C_{IN}	Input capacitance	-	-	5	-	pF
V_{OL}	Output low voltage[b]	IOL = -2mA	-	-	0.14	V
V_{OH}	Output high voltage[b]	IOH = 2mA	3.0	-	-	V
I_{OL}	Output low current[c]	VO = 0.4V	18	-	-	mA
I_{OH}	Output high current[c]	VO = 2.3V	17	-	-	mA
R_{PU}	Pullup resistor	-	50	-	65	kΩ
R_{PD}	Pulldown resistor	-	50	-	65	kΩ

[a] Hysteresis enabled

[b] Default drive strength (8mA)

[c] Maximum drive strength (16mA)

The table below gives the various voltage specifications for the GPIO pins for BCM2711-based products (e.g. Raspberry Pi 4 and Raspberry Pi 400). For information about Compute Modules you should see the relevant datasheets.

Symbol	Parameter	Conditions	Min	Typical	Max	Unit
V_{IL}	Input Low Voltage	-	-	-	0.8	V
V_{IH}	Input high voltage[a]	-	2.0	-	-	V
I_{IL}	Input leakage current	TA = +85○C	-	-	10	μA
V_{OL}	Output low voltage[b]	IOL = -4mA	-	-	0.4	V
V_{OH}	Output high voltage[b]	IOH = 4mA	2.6	-	-	V
I_{OL}	Output low current[c]	VO = 0.4V	7	-	-	mA
I_{OH}	Output high current[c]	VO = 2.6V	7	-	-	mA
R_{PU}	Pullup resistor	-	33	-	73	kΩ
R_{PD}	Pulldown resistor	-	33	-	73	kΩ

[a] Hysteresis enabled

[b] Default drive strength (4mA)

[c] Maximum drive strength (8mA)

GPIO Pads Control

Edit this on GitHub

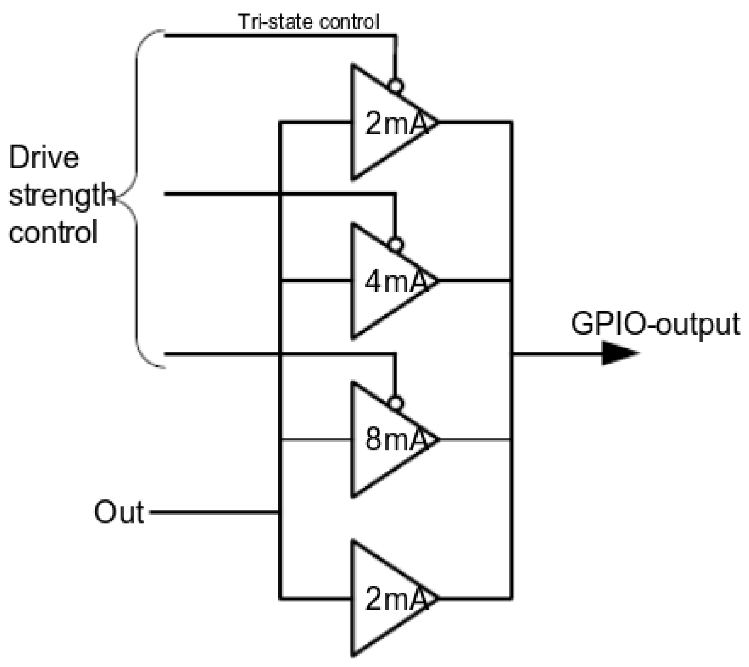
GPIO drive strengths do not indicate a maximum current, but a maximum current under which the pad will still meet the specification. You should set the GPIO drive strengths to match the device being attached in order for the device to work correctly.

How Drive Strength is Controlled

Inside the pad are a number of drivers in parallel. If the drive strength is set low (0b000) most of these are tri-stated so they do not add anything to the output current. If the drive strength is increased, more and more drivers are put in parallel. The following diagram shows that behaviour.

WARNING

For Raspberry Pi 4, Raspberry Pi 400 and Compute Module 4 the current level is half the value shown in the diagram.



What does the current value mean?

NOTE

The current value specifies the maximum current under which the pad will still meet the specification.

1. It is **not** the current that the pad will deliver
2. It is **not** a current limit so the pad will not blow up

The pad output is a voltage source:

- If set high, the pad will try to drive the output to the rail voltage (3.3 volts)
- If set low, the pad will try to drive the output to ground (0 volts)

The pad will try to drive the output high or low. Success will depend on the requirements of what is connected. If the pad is shorted to ground, it will not be able to drive high. It will actually try to deliver as much current as it can, and the current is only limited by the internal resistance.

If the pad is driven high and it is shorted to ground, in due time it will fail. The same holds true if you connect it to 3.3V and drive it low.

Meeting the specification is determined by the guaranteed voltage levels. Because the pads are digital, there are two voltage levels, high and low. The I/O ports have two parameters which deal with the output level:

- V_{OL}, the maximum low-level voltage (0.14V at 3.3V VDD IO)
- V_{OH}, the minimum high-level voltage (3.0V at 3.3V VDD IO)

V_{OL}=0.14V means that if the output is Low, it will be <= 0.14V. V_{OH}=3.0V means that if the output is High, it will be >= 3.0V.

Thus a drive strength of 16mA means:

If you set the pad high, you can draw up to 16mA, and the output voltage is guaranteed to be >=V_{OH}. This also means that if you set a drive strength of 2mA and you draw 16mA, the voltage will **not** be V_{OH} but lower. In fact, it may not be high enough to be seen as high by an external device.

There is more information on the physical characteristics of the GPIO pins.

NOTE

On the Compute Module devices, it is possible to change the VDD IO from the standard 3.3V. In this case, V_{OL} and V_{OH} will change according to the table on the linked page.

Why don't I set all my pads to the maximum current?

Two reasons:

1. The Raspberry Pi 3.3V supply was designed with a maximum current of ~3mA per GPIO pin. If you load each pin with 16mA, the total current is 272mA. The 3.3V supply will collapse under that level of load.
2. Big current spikes will happen, especially if you have a capacitive load. That will "bounce" around all the other pins near it. It is likely to cause interference with the SD card or even the SDRAM behaviour.

What is a safe current?

All the electronics of the pads are designed for 16mA. That is a safe value under which you will not damage the device. Even if you set the drive strength to 2mA and then load it so 16mA comes out, this will not damage the device. Other than that, there is no guaranteed maximum safe current.

GPIO Addresses

- 0x 7e10 002c PADS (GPIO 0-27)
- 0x 7e10 0030 PADS (GPIO 28-45)
- 0x 7e10 0034 PADS (GPIO 46-53)

Bits	Field name	Description	Type	Reset
31:24	PASSWRD	Must be 0x5A when writing; accidental write protect password	W	0
23:5		**Reserved** - Write as 0, read as don't care		
4	SLEW	Slew rate; 0 = slew rate	RW	0x1

Bits	Field name	Description	Type	Reset
		limited; 1 = slew rate not limited		
3	HYST	Enable input hysteresis; 0 = disabled; 1 = enabled	RW	0x1
2:0	DRIVE	Drive strength, see breakdown list below	RW	0x3

Beware of SSO (Simultaneous Switching Outputs) limitations which are device-dependent as well as dependent on the quality and layout of the PCB, the amount and quality of the decoupling capacitors, the type of load on the pads (resistance, capacitance), and other factors beyond the control of Raspberry Pi.

Drive strength list

- 0 = 2mA
- 1 = 4mA
- 2 = 6mA
- 3 = 8mA
- 4 = 10mA
- 5 = 12mA
- 6 = 14mA
- 7 = 16mA

Peripheral Addresses

Edit this on GitHub

If there is no kernel driver available, and a program needs to access a peripheral address directly with mmap, it needs to know where in the virtual memory map the peripheral bus segment has been placed. This varies according to which model of Raspberry Pi is being used, so there are three helper functions in bcm_host.c to help provide platform independence.

NOTE

You should use these functions rather than hardcoded values, as this will ensure future compatibility.

```
unsigned bcm_host_get_peripheral_address()
```

This returns the ARM-side physical address where peripherals are mapped.

```
unsigned bcm_host_get_peripheral_size()
```

This returns the size of the peripheral's space.

```
unsigned bcm_host_get_sdram_address()
```

This returns the bus address of the SDRAM.

Here are the current values as of this writing, in table form:

SoC	Peripheral Address	Peripheral Size	SDRAM Address	Source
BCM2835	0x20000000	0x01000000	0x40000000	bcm2835.dtsi
BCM2836	0x3f000000	0x01000000	0xC0000000	bcm2836.dtsi
BCM2837	0x3f000000	0x01000000	0xC0000000	bcm2837.dtsi
BCM2711	0xfe000000	0x01800000	0xc0000000	bcm2711.dtsi

Building a C program using these functions

The include file and library are installed by default on a Raspberry Pi OS system. Just add the following line to your C program:

```
#include <bcm_host.h>
```

Example:

```
#include <stdio.h>
#include <bcm_host.h>

int main(void) {
    printf("bcm_host_get_peripheral_address -> 0x%08x\n", bcm_host_get_pe
ripheral_address());
    printf("bcm_host_get_peripheral_size -> 0x%08x\n", bcm_host_get_perip
heral_size());
    printf("bcm_host_get_sdram_address -> 0x%08x\n", bcm_host_get_sdram_a
ddress());

    return 0;
}
```

Link with:

```
-lbcm_host
```

So a simple command line compile might be:

```
cc myfile.c -I/opt/vc/include -L/opt/vc/lib -lbcm_host -o myfile
```

Industrial use of the Raspberry Pi

Edit this on GitHub

The Raspberry Pi is often used as part of another product. This documentation describes some extra facilities available to use other capabilities of the Raspberry Pi.

One-Time Programmable Settings

There are a number of OTP values that can be used. To see a list of all the OTP values, you can use:

```
vcgencmd otp_dump
```

Some interesting lines from this dump are:

- 28 - Serial number
- 29 - Ones complement of serial number
- 30 - Revision number

Also, from 36 to 43 (inclusive), there are eight rows of 32 bits available for the customer

To program these bits, you will need to use the vcmailbox. This is a Linux driver interface to the firmware which will handle the programming of the rows. To do this, please refer to the documentation, and the vcmailbox example application.

The vcmailbox application can be used directly from the command line on Raspberry Pi OS. An example usage would be:

```
vcmailbox 0x00010004 8 8 0 0
```

which will return something like:

```
0x00000020 0x80000000 0x00010004 0x00000008 0x800000008 0xnnnnnnnn 0x0000
0000 0x00000000
```

The above uses the mailbox property interface `GET_BOARD_SERIAL` with a request size of 8 bytes and response size of 8 bytes (sending two integers for the request 0, 0). The response to this will be two integers (0x00000020 and 0x80000000) followed by the tag code, the request length, the response length (with the 31st bit set to indicate that it is a response) then the 64-bit serial number (where the MS 32 bits are always 0).

Write and Read Customer OTP Values

WARNING

The OTP values are One-Time Programmable, once a bit has been changed from 0 to 1, it can't be changed back

To set the customer OTP values you will need to use the `SET_CUSTOMER_OTP` (0x38021) tag as follows:

```
vcmailbox 0x00038021 [8 + number * 4] [8 + number * 4] [start_num] [numbe
r] [value] [value] [value] ...
```

- `start_num` = the first row to program from 0-7
- `number` = number of rows to program
- `value` = each value to program

So, to program OTP customer rows 4, 5, and 6 to 0x11111111, 0x22222222, 0x33333333 respectively, you would use:

```
vcmailbox 0x00038021 20 20 4 3 0x11111111 0x22222222 0x33333333
```

This will then program rows 40, 41, and 42.

To read the values back, you can use:

```
vcmailbox 0x00030021 20 20 4 3 0 0 0
```

which should display:

```
0x0000002c 0x80000000 0x00030021 0x00000014 0x80000014 0x00000000 0x00000
003 0x11111111 0x22222222 0x33333333
```

If you'd like to integrate this functionality into your own code, you should be able to achieve this by using the vcmailbox.c code as an example.

Locking the OTP Changes

It is possible to lock the OTP changes to avoid them being edited again. This can be done using a special argument with the OTP write mailbox:

```
vcmailbox 0x00038021 8 8 0xffffffff 0xaffe0000
```

Once locked, the customer OTP values can no longer be altered. Note that this locking operation is irreversible.

Making Customer OTP bits Unreadable

It is possible to prevent the customer OTP bits from being read at all. This can be done using a special argument with the OTP write mailbox:

```
vcmailbox 0x00038021 8 8 0xffffffff 0xaffebabe
```

This operation is unlikely to be useful for the vast majority of users, and is irreversible.

Device specific private key

Eight rows of OTP (256 bits) are available for use as a device-specific private key. This is intended to support file-system encryption.

These rows can be programmed and read using similar `vcmailbox` commands to those used for managing customer OTP rows. If secure-boot / file-system encryption is not required then the device private key rows can be used to store general purpose information.

- The device private key rows can only be read via the `vcmailbox` command which requires access to `/dev/vcio` which is restricted to the `video` group on Raspberry Pi OS.
- Raspberry Pi computers do not have a hardware protected key store. It is recommended that this feature is used in conjunction with secure-boot in order to restrict access to this data.
- Raspberry Pi OS does not support an encrypted root-filesystem.

See cryptsetup for more information about open-source disk encryption.

Key programming script `rpi-otp-private-key`

The rpi-otp-private-key script wraps the device private key `vcmailbox` APIs in order to make it easier to read/write a key in the same format as OpenSSL.

Read the key as a 64-byte hex number

```
rpi-otp-private-key
```

Example output

```
f8dbc7b0a4fcfb1d706e298ac9d0485c2226ce8df7f7596ac77337bd09fbe160
```

Writes a 64-byte randomly generated number to the device private key.
Warning: This operation cannot be undone.

```
# rpi-otp-private-key -w $(openssl rand -hex 32)
```

Mailbox API for reading/writing the key.

Read all of the rows.

```
vcmailbox 0x00030081 40 40 0 8 0 0 0 0 0 0 0 0
```

Example output

```
0x00000040 0x80000000 0x00030081 0x00000028 0x80000028 0x00000000 0x00000
008 0xf8dbc7b0 0xa4fcfb1d 0x706e298a 0xc9d0485c 0x2226ce8d 0xf7f7596a 0xc
77337bd 0x09fbe160 0x00000000
```

Write all of the row (replace the trailing eight zeros with the key data)

```
vcmailbox 0x00038081 40 40 0 8 0 0 0 0 0 0 0 0
```

Write the key shown in the previous example

```
vcmailbox 0x38081 40 40 0 8 0xf8dbc7b0 0xa4fcfb1d 0x706e298a 0xc9d0485c 0
x2226ce8d 0xf7f7596a 0xc77337bd 0x09fbe160
```

OTP Register and Bit Definitions

Edit this on GitHub

All SoCs used by the Raspberry Pi range have a inbuilt One-Time Programmable (OTP) memory block.

It is 66 32-bit values long, although only a few locations have factory-programmed data.

The `vcgencmd` to display the contents of the OTP is:

```
vcgencmd otp_dump
```

OTP Registers

This list contains the publicly available information on the registers. If a register or bit is not defined here, then it is not public.

17 — bootmode register

- Bit 1: sets the oscillator frequency to 19.2MHz
- Bit 3: enables pull ups on the SDIO pins
- Bit 19: enables GPIO bootmode
- Bit 20: sets the bank to check for GPIO bootmode
- Bit 21: enables booting from SD card
- Bit 22: sets the bank to boot from
- Bit 28: enables USB device booting
- Bit 29: enables USB host booting (ethernet and mass storage)

NOTE

On BCM2711 the bootmode is defined by the bootloader EEPROM configuration instead of OTP.

18 — copy of bootmode register
28 — serial number
29 — ~(serial number)
30 — revision code [1]
33 — board revision extended - the meaning depends on the board model.
This is available via device-tree in `/proc/device-tree/chosen/rpi-boardrev-ext` and for testing purposes this OTP value can be temporarily overridden by setting `board_rev_ext` in `config.txt`.

- Compute Module 4
    - Bit 30: Whether the Compute Module has a WiFi module fitted
        - 0 - WiFi
        - 1 - No WiFi
    - Bit 31: Whether the Compute Module has an EMMC module fitted
        - 0 - EMMC
        - 1 - No EMMC (Lite)
- Raspberry Pi 400
    - Bits 0-7: The default keyboard country code used by piwiz

36-43 — customer OTP values
45 — MPG2 decode key
46 — WVC1 decode key
47-54 — SHA256 of RSA public key for secure-boot
55 — secure-boot flags (reserved for use by the bootloader)
56-63 — 256-bit device-specific private key
64-65 — MAC address; if set, system will use this in preference to the automatically generated address based on the serial number
66 — advanced boot register (not BCM2711)

- Bits 0-6: GPIO for ETH_CLK output pin
- Bit 7: enables ETH_CLK output
- Bits 8-14: GPIO for LAN_RUN output pin
- Bit 15: enables LAN_RUN output
- Bit 24: extends USB HUB timeout parameter
- Bit 25: ETH_CLK frequency:
    - 0 - 25MHz
    - 1 - 24MHz

^{1}Also contains bits to disable overvoltage, OTP programming, and OTP reading.

Power Supply

Edit this on GitHub

The power supply requirements differ by Raspberry Pi model. All models require a 5.1V supply, but the current required generally increases according to model. All models up to the Raspberry Pi 3 require a micro USB power connector, whilst the Raspberry Pi 4 and Raspberry Pi 400 use a USB-C connector.

Exactly how much current (mA) the Raspberry Pi requires is dependent on what you connect to it. The following table gives various current requirements.

Product	Recommended PSU current capacity	Maximum total USB peripheral current draw	Typical bare-board active current consumption
Raspberry Pi 1 Model A	700mA	500mA	200mA
Raspberry Pi 1 Model B	1.2A	500mA	500mA
Raspberry Pi 1 Model A+	700mA	500mA	180mA
Raspberry Pi 1 Model B+	1.8A	1.2A	330mA
Raspberry Pi 2 Model B	1.8A	1.2A	350mA
Raspberry Pi 3 Model B	2.5A	1.2A	400mA
Raspberry Pi 3 Model A+	2.5A	Limited by PSU, board, and connector ratings only.	350mA
Raspberry Pi 3 Model B+	2.5A	1.2A	500mA
Raspberry Pi 4 Model B	3.0A	1.2A	600mA
Raspberry Pi 400	3.0A	1.2A	800mA
Raspberry Pi Zero	1.2A	Limited by PSU, board, and connector ratings only	100mA
Raspberry Pi Zero W	1.2A	Limited by PSU, board, and connector ratings only.	150mA
Raspberry Pi Zero 2 W	2A	Limited by PSU, board, and connector ratings only.	350mA

Raspberry Pi have developed their own power supplies for use with all models. These are reliable, use heavy gauge wires and are reasonably priced.

For Raspberry Pi 0-3, we recommend our 2.5A micro USB Supply. For Raspberry Pi 4 and Raspberry Pi 400, we recommend our 3A USB-C Supply.

If you need to connect a USB device that will take the power requirements above the values specified in the table above, then you must connect it using an externally-powered USB hub.

Typical Power Requirements

The specific power requirements of each model are shown below.

Product	Recommended PSU current capacity	Maximum total USB peripheral current draw	Typical bare-board active current consumption
Raspberry Pi 1 Model A	700mA	500mA	200mA
Raspberry Pi 1 Model B	1.2A	500mA	500mA
Raspberry Pi 1 Model A+	700mA	500mA	180mA
Raspberry Pi 1 Model B+	1.8A	1.2A	330mA
Raspberry Pi 2 Model B	1.8A	1.2A	350mA
Raspberry Pi 3 Model B	2.5A	1.2A	400mA
Raspberry Pi 3 Model A+	2.5A	Limited by PSU, board, and connector ratings only.	350mA
Raspberry Pi 3 Model B+	2.5A	1.2A	500mA
Raspberry Pi 4 Model B	3.0A	1.2A	600mA
Raspberry Pi 400	3.0A	1.2A	800mA
Raspberry Pi Zero	1.2A	Limited by PSU, board, and connector ratings only	100mA
Raspberry Pi Zero W	1.2A	Limited by PSU, board, and connector ratings only.	150mA
Raspberry Pi Zero 2 W	2A	Limited by PSU, board, and connector ratings only	350mA

From the Raspberry Pi B+ onwards, 1.2A is supplied to downstream USB peripherals. This allows the vast majority of USB devices to be connected directly to these models, assuming the upstream power supply has sufficient available current.

Very high-current devices, or devices which can draw a surge current such as certain modems and USB hard disks, will still require an external powered USB hub. The power requirements of the Raspberry Pi increase as you make use of the various interfaces on the Raspberry Pi. The GPIO pins can draw 50mA safely (note that that means 50mA distributed across all the pins: an individual GPIO pin can only safely draw 16mA), the HDMI port uses 50mA, the Camera Module requires 250mA, and keyboards and mice can take as little as 100mA or as much as 1000mA! Check the power rating of the devices you plan to connect to the Raspberry Pi and purchase a power supply accordingly. If you're not sure, we would advise you to buy a powered USB hub.

This is the typical amount of power (in Ampere) drawn by different Raspberry Pi models during standard processes:

		Raspberry Pi 1B+	Raspberry Pi 2B	Raspberry Pi 3B	Raspberry Pi Zero	Raspberry Pi 4B
Boot	Max	0.26	0.40	0.75	0.20	0.85
	Avg	0.22	0.22	0.35	0.15	0.7
Idle	Avg	0.20	0.22	0.30	0.10	0.6
Video playback (H.264)	Max	0.30	0.36	0.55	0.23	0.85
	Avg	0.22	0.28	0.33	0.16	0.78
Stress	Max	0.35	0.82	1.34	0.35	1.25
	Avg	0.32	0.75	0.85	0.23	1.2
Halt current				0.10	0.055	0.023

NOTE

For these measurements we used a standard Raspberry Pi OS image (current as of 26 Feb 2016, or June 2019 for the Raspberry Pi 4), at room temperature, with the Raspberry Pi connected to a HDMI monitor, USB keyboard, and USB mouse. The Raspberry Pi 3 Model B was connected to a wireless LAN access point, the Raspberry Pi 4 was connected to Ethernet. All these power measurements are approximate and do not take into account power consumption from additional USB devices; power consumption can easily exceed these measurements if multiple additional USB devices or a HAT are connected to the Raspberry Pi.

Power Supply Warnings

On all models of Raspberry Pi since the Raspberry Pi B+ (2014) except the Zero range, there is low-voltage detection circuitry that will detect if the supply voltage drops below 4.63V (+/- 5%). This will result in a warning icon being displayed on all attached displays and an entry being added to the kernel log.

If you are seeing warnings, you should improve the power supply and/or cable, as low power can cause problems with corruption of SD cards, or erratic behaviour of the Raspberry Pi itself; for example, unexplained crashes.

Voltages can drop for a variety of reasons, for example if the power supply itself is inadequate, the power supply cable is made of too thin wires, or you have plugged in high demand USB devices.

Back-powering

The USB specification requires that USB devices must not supply current to upstream devices. If a USB device does supply current to an upstream device then this is called back-powering. Often this happens when a badly-made powered USB hub is connected, and will result in the powered USB hub supplying power to the host Raspberry Pi. This is not recommended since the power being supplied to the Raspberry Pi via the hub will bypass the protection circuitry built into the Raspberry Pi, leaving it vulnerable to damage in the event of a power surge.

Serial Peripheral Interface (SPI)

Edit this on GitHub

Raspberry Pi computers are equipped with a number of SPI buses. SPI can be used to connect a wide variety of peripherals - displays, network controllers (Ethernet, CAN bus), UARTs, etc. These devices are best supported by kernel device drivers, but the `spidev` API allows userspace drivers to be written in a wide array of languages.

SPI Hardware

Raspberry Pi Zero, 1, 2 and 3 have three SPI controllers:

- SPI0, with two hardware chip selects, is available on the header of all Raspberry Pis; there is also an alternate mapping that is only available on Compute Modules.
- SPI1, with three hardware chip selects, is available on all Raspberry Pi models except the original Raspberry Pi 1 Model A and Model B.
- SPI2, also with three hardware chip selects, is only available on Compute Module 1, 3 and 3+.

On the Raspberry Pi 4, 400 and Compute Module 4 there are four additional SPI buses: SPI3 to SPI6, each with 2 hardware chip selects. These extra SPI buses are available via alternate function assignments on certain GPIO pins - see the BCM2711 ARM Peripherals datasheet.

Chapter 10 in the BCM2835 ARM Peripherals datasheet describes the main controller. Chapter 2.3 describes the auxiliary controller.

Pin/GPIO mappings

SPI0

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	19	GPIO10	SPI0_MOSI
MISO	21	GPIO09	SPI0_MISO
SCLK	23	GPIO11	SPI0_SCLK
CE0	24	GPIO08	SPI0_CE0_N
CE1	26	GPIO07	SPI0_CE1_N

SPI0 alternate mapping (Compute Modules only, except CM4)

SPI Function	Broadcom Pin Name	Broadcom Pin Function
MOSI	GPIO38	SPI0_MOSI
MISO	GPIO37	SPI0_MISO
SCLK	GPIO39	SPI0_SCLK
CE0	GPIO36	SPI0_CE0_N
CE1	GPIO35	SPI0_CE1_N

SPI1

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	38	GPIO20	SPI1_MOSI
MISO	35	GPIO19	SPI1_MISO
SCLK	40	GPIO21	SPI1_SCLK
CE0	12	GPIO18	SPI1_CE0_N
CE1	11	GPIO17	SPI1_CE1_N
CE2	36	GPIO16	SPI1_CE2_N

SPI2 (Compute Modules only, except CM4)

SPI Function	Broadcom Pin Name	Broadcom Pin Function
MOSI	GPIO41	SPI2_MOSI
MISO	GPIO40	SPI2_MISO
SCLK	GPIO42	SPI2_SCLK
CE0	GPIO43	SPI2_CE0_N
CE1	GPIO44	SPI2_CE1_N
CE2	GPIO45	SPI2_CE2_N

SPI3 (BCM2711 only)

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	03	GPIO02	SPI3_MOSI
MISO	28	GPIO01	SPI3_MISO
SCLK	05	GPIO03	SPI3_SCLK
CE0	27	GPIO00	SPI3_CE0_N
CE1	18	GPIO24	SPI3_CE1_N

SPI4 (BCM2711 only)

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	31	GPIO06	SPI4_MOSI
MISO	29	GPIO05	SPI4_MISO
SCLK	26	GPIO07	SPI4_SCLK
CE0	07	GPIO04	SPI4_CE0_N
CE1	22	GPIO25	SPI4_CE1_N

SPI5 (BCM2711 only)

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	08	GPIO14	SPI5_MOSI
MISO	33	GPIO13	SPI5_MISO
SCLK	10	GPIO15	SPI5_SCLK
CE0	32	GPIO12	SPI5_CE0_N
CE1	37	GPIO26	SPI5_CE1_N

SPI6 (BCM2711 only)

SPI Function	Header Pin	Broadcom Pin Name	Broadcom Pin Function
MOSI	38	GPIO20	SPI6_MOSI
MISO	35	GPIO19	SPI6_MISO
SCLK	40	GPIO21	SPI6_SCLK
CE0	12	GPIO18	SPI6_CE0_N
CE1	13	GPIO27	SPI6_CE1_N

Master modes

Signal name abbreviations

```
SCLK - Serial CLocK
CE   - Chip Enable (often called Chip Select)
MOSI - Master Out Slave In
MISO - Master In Slave Out
MOMI - Master Out Master In
```

Standard mode

In Standard SPI mode the peripheral implements the standard 3 wire serial protocol (SCLK, MOSI and MISO).

Bidirectional mode

In bidirectional SPI mode the same SPI standard is implemented, except that a single wire is used for data (MOMI) instead of the two used in standard mode (MISO and MOSI). In this mode, the MOSI pin serves as MOMI pin.

LoSSI mode (Low Speed Serial Interface)

The LoSSI standard allows issuing of commands to peripherals (LCD) and to transfer data to and from them. LoSSI commands and parameters are 8 bits long, but an extra bit is used to indicate whether the byte is a command or parameter/data. This extra bit is set high for a data and low for a command. The resulting 9-bit value is serialized to the output. LoSSI is commonly used with MIPI DBI type C compatible LCD controllers.

NOTE

Some commands trigger an automatic read by the SPI controller, so this mode cannot be used as a multipurpose 9-bit SPI.

Transfer modes

- Polled
- Interrupt
- DMA

Speed

The CDIV (Clock Divider) field of the CLK register sets the SPI clock speed:

```
SCLK = Core Clock / CDIV
```

If CDIV is set to 0, the divisor is 65536. The divisor must be a multiple of 2, with odd numbers rounded down. Note that not all possible clock rates are usable because of analogue electrical issues (rise times, drive strengths, etc).

See the Linux driver section for more info.

Chip Selects

Setup and hold times related to the automatic assertion and de-assertion of the CS lines when operating in **DMA** mode are as follows:

- The CS line will be asserted at least 3 core clock cycles before the msb of the first byte of the transfer.
- The CS line will be de-asserted no earlier than 1 core clock cycle after the trailing edge of the final clock pulse.

SPI Software

Linux driver

The default Linux driver is `spi-bcm2835`.

SPI0 is disabled by default. To enable it, use raspi-config, or ensure the line `dtparam=spi=on` is not commented out in `/boot/config.txt`. By default it uses 2 chip select lines, but this can be reduced to 1 using `dtoverlay=spi0-1cs`. `dtoverlay=spi0-2cs` also exists, and without any parameters it is equivalent to `dtparam=spi=on`.

To enable SPI1, you can use 1, 2 or 3 chip select lines, adding in each case:

```
dtoverlay=spi1-1cs  #1 chip select
dtoverlay=spi1-2cs  #2 chip select
dtoverlay=spi1-3cs  #3 chip select
```

to the `/boot/config.txt` file. Similar overlays exist for SPI2, SPI3, SPI4, SPI5 and SPI6.

The driver does not make use of the hardware chip select lines because of some limitations - instead it can use an arbitrary number of GPIOs as software/GPIO chip selects. This means you are free to choose any spare GPIO as a CS line, and all of these SPI overlays include that control - see `/boot/overlays/README` for details, or run (for example) `dtoverlay -h spi0-2cs` (`dtoverlay -a | grep spi` might be helpful to list them all).

Speed

The driver supports all speeds which are even integer divisors of the core clock, although as said above not all of these speeds will support data transfer due to limits in the GPIOs and in the devices attached. As a rule of thumb, anything over 50MHz is unlikely to work, but your mileage may vary.

Supported Mode bits

```
SPI_CPOL    - Clock polarity
SPI_CPHA    - Clock phase
SPI_CS_HIGH - Chip Select active high
SPI_NO_CS   - 1 device per bus, no Chip Select
SPI_3WIRE   - Bidirectional mode, data in and out pin shared
```

Bidirectional or "3-wire" mode is supported by the `spi-bcm2835` kernel module. Please note that in this mode, either the tx or rx field of the spi_transfer struct must be a NULL pointer, since only half-duplex communication is possible. Otherwise, the transfer will fail. The spidev_test.c source code does not consider this correctly, and therefore does not work at all in 3-wire mode.

Supported bits per word

- 8 - Normal
- 9 - This is supported using LoSSI mode.

Transfer modes

Interrupt mode is supported on all SPI buses. SPI0, and SPI3-6 also support DMA transfers.

SPI driver latency

This thread discusses latency problems.

spidev

`spidev` presents an ioctl-based userspace interface to individual SPI CS lines. Device Tree is used to indicate whether a CS line is going to be driven by a kernel driver module or managed by spidev on behalf of the user; it is not possible to do both at the same time. Note that Raspberry Pi's own kernels are more relaxed about the use of Device Tree to enable `spidev` - the upstream kernels print warnings about such usage, and ultimately may prevent it altogether.

Using spidev from C

There is a loopback test program in the Linux documentation that can be used as a starting point. See the Troubleshooting section.

Using spidev from Python

There are several Python libraries that provide access to `spidev`, including `spidev` (`pip install spidev` - see https://pypi.org/project/spidev/) and `SPI-Py` (https://github.com/lthiery/SPI-Py).

Using spidev from a shell such as bash

```
# Write binary 1, 2 and 3
echo -ne "\x01\x02\x03" > /dev/spidev0.0
```

Other SPI libraries

There are other userspace libraries that provide SPI control by directly manipulating the hardware: this is not recommended.

Troubleshooting

Loopback test

This can be used to test SPI send and receive. Put a wire between MOSI and MISO. It does not test CE0 and CE1.

```
wget https://raw.githubusercontent.com/raspberrypi/linux/rpi-3.10.y/Docum
entation/spi/spidev_test.c
gcc -o spidev_test spidev_test.c
./spidev_test -D /dev/spidev0.0
spi mode: 0
bits per word: 8
max speed: 500000 Hz (500 KHz)

FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

Some of the content above has been copied from the elinux SPI page, which also borrows from here. Both are covered by the CC-SA license.

Universal Serial Bus (USB)

Edit this on GitHub

In general, every device supported by Linux can be used with a Raspberry Pi, although there are some limitations for models prior to Raspberry Pi 4.

Maximum Power Output

As with all computers, the USB ports on the Raspberry Pi supply a limited amount of power. Often problems with USB devices are caused by power issues. To rule out insufficient power as the cause of the problem, connect your USB devices to the Raspberry Pi using a powered hub.

Model	**Max power output of USB ports**
Pi Zero, 1	500mA per port[1]
Pi 2, 3, 4	1200mA total across all ports

1. For the original Raspberry Pi 1 Model B the limit is 100mA per port.

Raspberry Pi 4

The Raspberry Pi 4 contains two USB 3.0 ports and two USB 2.0 ports which are connected to a VL805 USB controller. The USB 2.0 lines on all four ports are connected to a single USB 2.0 hub within the VL805: this limits the total available bandwidth for USB 1.1 and USB 2.0 devices to that of a single USB 2.0 port.

On the Raspberry Pi 4, the USB controller used on previous models is located on the USB type C port and is disabled by default.

Raspberry Pi Zero, 1, 2 and 3

The USB controller on models prior to Raspberry Pi 4 has only a basic level of support for certain devices, which presents a higher software processing overhead. It also supports only one root USB port: all traffic from connected devices is funnelled down this single bus, which operates at a maximum speed of 480Mbps.

The USB 2.0 specification defines three device speeds - low, full and high. Most mice and keyboards are low speed, most USB sound devices are full speed and most video devices (webcams or video capture) are high speed.

Generally, there are no issues with connecting multiple high speed USB devices to a Raspberry Pi.

The software overhead incurred when talking to low and full speed devices means that there are limitations on the number of simultaneously active low and full speed devices. Small numbers of these types of devices connected to a Raspberry Pi will cause no issues.

Known USB Issues

Interoperability with USB 3.0 hubs

There is an issue with USB 3.0 hubs in conjunction with the use of full or low speed devices, including most mice and keyboards. A bug in most USB 3.0 hub hardware means that the models prior to Raspberry Pi 4 cannot talk to full or low speed devices connected to a USB 3.0 hub.

USB 2.0 high speed devices, including USB 2.0 hubs, operate correctly when connected via a USB 3.0 hub.

Avoid connecting low or full speed devices into a USB 3.0 hub. As a workaround, plug a USB 2.0 hub into the downstream port of the USB 3.0 hub and connect the low speed device, or use a USB 2.0 hub between the Raspberry Pi and the USB 3.0 hub, then plug low speed devices into the USB 2.0 hub.

USB 1.1 webcams

Old webcams may be full speed devices. Because these devices transfer a lot of data and incur additional software overhead, reliable operation is not guaranteed. As a workaround, try to use the camera at a lower resolution.

Esoteric USB sound cards

Expensive audiophile sound cards typically use large amounts of USB bandwidth: reliable operation with 96kHz/192kHz DACs is not guaranteed. As a workaround, forcing the output stream to be CD quality (44.1kHz/48kHz 16-bit) will reduce the stream bandwidth to reliable levels.

Single TT USB hubs

USB 2.0 and 3.0 hubs have a mechanism for talking to full or low speed devices connected to their downstream ports called a transaction translator (TT). This device buffers high speed requests from the host and transmits them at full or low speed to the downstream device. Two configurations of hub are allowed by the USB specification: Single TT (one TT for all ports) and Multi TT (one TT per port). Because of a hardware limitation, if too many full or low speed devices are plugged into a single TT hub, the devices may behave unreliably. It is recommended to use a Multi TT hub to interface with multiple full and low speed devices. As a workaround, spread full and low speed devices out between the Raspberry Pi's own USB port and the single TT hub.

Raspberry Pi revision codes

Edit this on GitHub

Each distinct Raspberry Pi model revision has a unique revision code. You can look up a Raspberry Pi's revision code by running:

```
cat /proc/cpuinfo
```

The last three lines show the hardware type, the revision code, and the Raspberry Pi's unique serial number. For example:

```
Hardware    : BCM2835
Revision    : a02082
Serial      : 00000000765fc593
```

NOTE

As of the 4.9 kernel, all Raspberry Pi computers report `BCM2835`, even those with BCM2836, BCM2837 and BCM2711 processors. You should not use this string to detect the processor. Decode the revision code using the information below, or `cat /sys/firmware/devicetree/base/model`.

Old-style revision codes

The first set of Raspberry Pi models were given sequential hex revision codes from `0002` to `0015`:

Code	Model	Revision	RAM	Manufacturer
0002	B	1.0	256MB	Egoman
0003	B	1.0	256MB	Egoman
0004	B	2.0	256MB	Sony UK
0005	B	2.0	256MB	Qisda
0006	B	2.0	256MB	Egoman
0007	A	2.0	256MB	Egoman
0008	A	2.0	256MB	Sony UK
0009	A	2.0	256MB	Qisda
000d	B	2.0	512MB	Egoman
000e	B	2.0	512MB	Sony UK
000f	B	2.0	512MB	Egoman
0010	B+	1.2	512MB	Sony UK
0011	CM1	1.0	512MB	Sony UK
0012	A+	1.1	256MB	Sony UK
0013	B+	1.2	512MB	Embest
0014	CM1	1.0	512MB	Embest
0015	A+	1.1	256MB/512MB	Embest

New-style revision codes

With the launch of the Raspberry Pi 2, new-style revision codes were introduced. Rather than being sequential, each bit of the hex code represents a piece of information about the revision:

```
NOQuuuWuFMMMCCCCPPPPTTTTTTTTRRRR
```

Part	Represents	Options
N (bit 31)	Overvoltage	0: Overvoltage allowed
		1: Overvoltage disallowed
O (bit 30)	OTP Program[1]	0: OTP programming allowed
		1: OTP programming disallowed
Q (bit 29)	OTP Read[1]	0: OTP reading allowed
		1: OTP reading disallowed
uuu (bits 26-28)	Unused	Unused
W (bit 25)	Warranty bit[2]	0: Warranty is intact
		1: Warranty has been voided by overclocking
u (bit 24)	Unused	Unused

Part	Represents	Options
F (bit 23)	New flag	1: new-style revision
		0: old-style revision
MMM (bits 20-22)	Memory size	0: 256MB
		1: 512MB
		2: 1GB
		3: 2GB
		4: 4GB
		5: 8GB
CCCC (bits 16-19)	Manufacturer	0: Sony UK
		1: Egoman
		2: Embest
		3: Sony Japan
		4: Embest
		5: Stadium
PPPP (bits 12-15)	Processor	0: BCM2835
		1: BCM2836
		2: BCM2837
		3: BCM2711
TTTTTTTT (bits 4-11)	Type	0: A
		1: B
		2: A+
		3: B+
		4: 2B
		5: Alpha (early prototype)
		6: CM1
		8: 3B
		9: Zero
		a: CM3
		c: Zero W
		d: 3B+
		e: 3A+
		f: Internal use only
		10: CM3+
		11: 4B
		12: Zero 2 W
		13: 400

Part	Represents	Options
		14: CM4
		15: CM4S
RRRR (bits 0-3)	Revision	0, 1, 2, etc.

[1] Information on programming the OTP bits.

[2] The warranty bit is never set on Raspberry Pi 4.

New-style revision codes in use

NOTE

This list is not exhaustive - there may be codes in use that are not in this table. Please see the next section for best practices on using revision codes to identify boards.

Code	Model	Revision	RAM	Manufacturer
900021	A+	1.1	512MB	Sony UK
900032	B+	1.2	512MB	Sony UK
900092	Zero	1.2	512MB	Sony UK
900093	Zero	1.3	512MB	Sony UK
9000c1	Zero W	1.1	512MB	Sony UK
9020e0	3A+	1.0	512MB	Sony UK
920092	Zero	1.2	512MB	Embest
920093	Zero	1.3	512MB	Embest
900061	CM1	1.1	512MB	Sony UK
a01040	2B	1.0	1GB	Sony UK
a01041	2B	1.1	1GB	Sony UK
a02082	3B	1.2	1GB	Sony UK
a020a0	CM3	1.0	1GB	Sony UK
a020d3	3B+	1.3	1GB	Sony UK
a02042	2B (with BCM2837)	1.2	1GB	Sony UK
a21041	2B	1.1	1GB	Embest
a22042	2B (with BCM2837)	1.2	1GB	Embest
a22082	3B	1.2	1GB	Embest
a220a0	CM3	1.0	1GB	Embest
a32082	3B	1.2	1GB	Sony Japan
a52082	3B	1.2	1GB	Stadium
a22083	3B	1.3	1GB	Embest
a02100	CM3+	1.0	1GB	Sony UK

Code	Model	Revision	RAM	Manufacturer
a03111	4B	1.1	1GB	Sony UK
b03111	4B	1.1	2GB	Sony UK
b03112	4B	1.2	2GB	Sony UK
b03114	4B	1.4	2GB	Sony UK
b03115	4B	1.5	2GB	Sony UK
c03111	4B	1.1	4GB	Sony UK
c03112	4B	1.2	4GB	Sony UK
c03114	4B	1.4	4GB	Sony UK
c03115	4B	1.5	4GB	Sony UK
d03114	4B	1.4	8GB	Sony UK
d03115	4B	1.5	8GB	Sony UK
c03130	Pi 400	1.0	4GB	Sony UK
a03140	CM4	1.0	1GB	Sony UK
b03140	CM4	1.0	2GB	Sony UK
c03140	CM4	1.0	4GB	Sony UK
d03140	CM4	1.0	8GB	Sony UK
902120	Zero 2 W	1.0	512MB	Sony UK

Using revision codes for board identification

From the command line we can use the following to get the revision code of the board:

```
$cat /proc/cpuinfo | grep Revision
Revision      : c03111
```

In this example above, we have a hexadecimal revision code of `c03111`. Converting this to binary, we get `0 0 0 000 0 0 1 100 0000 0011 00010001 0001`. Spaces have been inserted to show the borders between each section of the revision code, according to the above table.

Starting from the lowest order bits, the bottom four (0-3) are the board revision number, so this board has a revision of 1. The next eight bits (4-11) are the board type, in this case binary `00010001`, hex `11`, so this is a Raspberry Pi 4B. Using the same process, we can determine that the processor is a BCM2711, the board was manufactured by Sony UK, and it has 4GB of RAM.

Getting the revision code in your program

Obviously there are so many programming languages out there it's not possible to give examples for all of them, but here are two quick examples for `C` and `Python`. Both these examples use a system call to run a bash command that gets the `cpuinfo` and pipes the result to `awk` to recover the required revision code. They then use bit operations to extract the `New`, `Model`, and `Memory` fields from the code.

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
  FILE *fp;
  char revcode[32];

  fp = popen("cat /proc/cpuinfo | awk '/Revision/ {print $3}'", "r");
  if (fp == NULL)
    exit(1);
  fgets(revcode, sizeof(revcode), fp);
  pclose(fp);

  int code = strtol(revcode, NULL, 16);
  int new = (code >> 23) & 0x1;
  int model = (code >> 4) & 0xff;
  int mem = (code >> 20) & 0x7;

  if (new && model == 0x11 && mem >= 3)  // Note, 3 in the mem field is 2
GB
     printf("We are a 4B with at least 2GB of RAM!\n" );

  return 0;
}
```

And the same in Python:

```
import subprocess

cmd = "cat /proc/cpuinfo | awk '/Revision/ {print $3}'"
revcode = subprocess.check_output(cmd, shell=True)

code = int(revcode, 16)
new = (code >> 23) & 0x1
model = (code >> 4) & 0xff
mem = (code >> 20) & 0x7

if new and model == 0x11 and mem >= 3 : # Note, 3 in the mem field is 2GB
    print("We are a 4B with at least 2GB RAM!")
```

Best practice for revision code usage

Raspberry Pi advises against using the revision code as a whole (`c03111`) to avoid problems when new board revisions are created. For example, you might consider having a list of supported revision codes in your program, and comparing the detected code with your list to determine if your program is allowed to run. However, this mechanism will break when a new board revision comes out, or if the production location changes, each of which would create a new revision code that's not in your program's list. Your program would now reject the unrecognised code, and perhaps abort, even though revisions of the same board type are always backwards-compatible. You would need to release a new version of your program with the specific revision added to the list, which can be a maintenance burden.

Similarly, using revision codes to indicate which model your program supports can create issues. If your program is only intended to work on devices with 2GB of RAM or more, a naive approach would be to look at the list of revision codes for models that have 2GB of RAM or more, and build that list in to your program. But of course, this breaks as soon as a new board revision is released, or if boards are manufactured at a different location.

A better mechanism is to just use the board-type field (3A, 4B, etc.) to determine which model your program supports; or perhaps just the amount-of-memory field. So you might say you will support any Raspberry Pi 4Bs, whatever their board revision code, because that should always work. Or you might want to restrict your program to 4B devices with

2GB of RAM or more. Simply look at those two fields to determine whether you are going to allow your program to run.

The examples in the previous section use the recommended approach. They pull out the board type and memory size from the revision code, and use them to determine whether or not they are a Raspberry Pi 4B with 2GB or more of RAM.

NOTE

You should always check bit 23, the 'New' flag, to ensure that the revision code is the new version before checking any other fields. The examples here also do this.