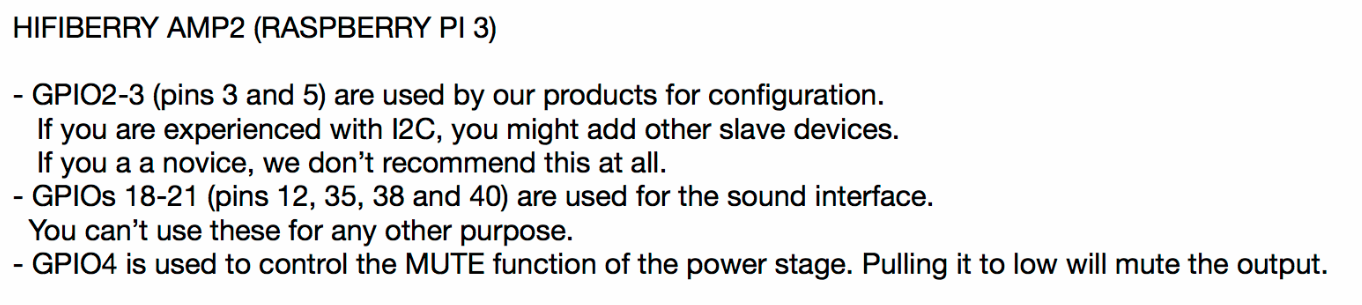
# **Using a rotary encoder / switch as a volume control – mute for HiFiBerry AMP2 as a Roon client v.1.9 or higher**

The HifiBerry Amp2 used as a Roon client should be controlled by the Roon app volume slider. I figured out how to add a **hardware-like volume control** using a rotary encoder and a daemon that listens for the signals it sends.

The Roon image v.1.9 is Raspbian based Linux so use a remote terminal[[1]](#footnote-1) - PuTTY SSH - and “sudo nano” to copy/paste from the clipboard[[2]](#footnote-2). Remember how to paste in a PuTTYwindow using nano text editor => don’t forget to select PuTTY window by clicking in it to make it active (!) followed by ***“Shift+ Right click****”*.

Starting from a genuine v.1.9 image, the scripts volume-monitor and volume-monitor.service are tested and working. Changes made in advance are for your own account and no longer guarantee the proper functioning of the system. The scripts can easily be copied from the "word document" with the extensive commentary # into the nano text editor.



Use free GPIO (40-pin header) of RPi

## Rotary encoder

An incremental rotary encoder (volume knob) looks like a standard potentiometer (variable resistor, analog volume knob) we all know, except (a) you can keep turning it in either direction for as long as you want, and thus (b) it talks to the RPi differently (pulses) than a potentiometer would.

You can pick up [this one](https://www.adafruit.com/products/377) from Adafruit, (a breakout board to solder also available) but there are plenty others available. This rotary encoder also has a push button function so it can be used for toggling mute on and off.

<https://cdn-shop.adafruit.com/datasheets/pec11.pdf>

So there are 5 wires to hook up: three for the encoder (A, B, and C common), and two for the push button part (momentary switch). Here's how I hooked them up ([reference](https://pinout.xyz/)):

| **Description** | **Raspberry Pi BCM #** | **RPi Board pin #** |
| --- | --- | --- |
| knob A (channel A) | GPIO 26 | 37 |
| knob B (channel B) | GPIO 16 *(if HiFiBerry Amp2 HAT is used)* | 36 |
| knob ground | ground pin below GPIO 26 | 39 |
| button common | GPIO 13 *(or GPIO4 can be used to control the MUTE function of the Amp2 power stage*) | 33 |
| button ground | ground pin opposite GPIO 13 | 34 |

You can use whichever pins you want, just update the script if you change them.

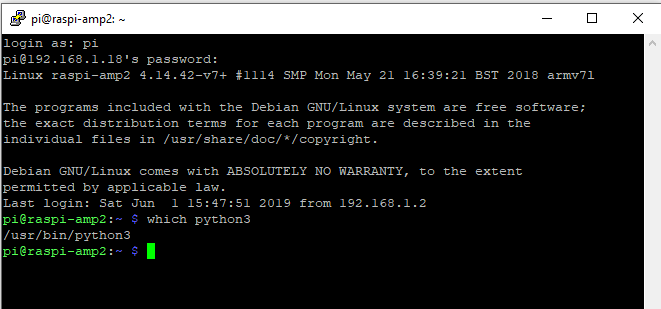
If the pinout (channel A and B) of your encoder does not match (turning left increases the volume > simply swap the pins or the corresponding number in the python script).

## Volume daemon

Since the GPIO pins accept arbitrary signals, you need a script on the Pi that knows what to do with them. A GPIO library for the Pi will do the job. Python 3 is preinstalled in the roon image v1.9 but de GPIO library is lacking (!) so install this first.

Further in this tutorial command line input will be marked yellow and/or text coloured **purple**

Start a SSH remote terminal using PuTTY on your PC (hostname, login with pi /hifiberry)



**which python3** <enter>

Python3 is installed but the library GPIO is lacking

sudo apt-get install python3 python3-rpi.gpio

Create a bin directory in your pi folder (if it doesn't exist already), then drop the script into it.

mkdir ~/bin

nano ~/bin/monitor-volume

Drop the script in it starting with #!/usr/bin/env python3 and ending with EVENT.clear(): it listens on the specified pins, and when the knob is turned one way or another, it uses the states of the A and B pins to figure out whether the knob was turned to the left or to the right. That way it knows whether to increase or decrease the system volume in response, which it does with the command-line program amixer.

[**monitor-volume**](https://gist.github.com/savetheclocktower/9b5f67c20f6c04e65ed88f2e594d43c1#file-monitor-volume)

|  |  |
| --- | --- |
|  | #!/usr/bin/env python3 |
|  |  |
|  | """ |
|  | The daemon responsible for changing the volume in response to a turn or press |
|  | of the volume knob. |
|  |  |
|  | The volume knob is a rotary encoder. It turns infinitely in either direction. |
|  | Turning it to the right will increase the volume; turning it to the left will |
|  | decrease the volume. The knob can also be pressed like a button in order to |
|  | turn muting on or off. |
|  |  |
|  | The knob uses two GPIO pins and we need some extra logic to decode it. The |
|  | button we can just treat like an ordinary button. Rather than poll |
|  | constantly, we use threads and interrupts to listen on all three pins in one |
|  | script. |
|  | """ |
|  |  |
|  | import os |
|  | import signal |
|  | import subprocess |
|  | import sys |
|  | import threading |
|  |  |
|  | from RPi import GPIO |
|  | from queue import Queue |
|  |  |
|  | DEBUG = False |
|  |  |
|  | # SETTINGS |
|  | # ======== |
|  |  |
|  | # The two pins that the encoder uses (BCM numbering). |
|  | GPIO\_A = 26 |
|  | GPIO\_B = 16 |
|  |  |
|  | # The pin that the knob's button is hooked up to. If you have no button, set |
|  | # this to None. |
|  | GPIO\_BUTTON = 13 |
|  |  |
|  | # The minimum and maximum volumes, as percentages. |
|  | # |
|  | # The default max is less than 100 to prevent distortion. The default min is |
|  | # greater than zero because if your system is like mine, sound gets |
|  | # completely inaudible \_long\_ before 0%. If you've got a hardware amp or |
|  | # serious speakers or something, your results will vary. |
|  | VOLUME\_MIN = 30 |
|  | VOLUME\_MAX = 96 |
|  |  |
|  | # The amount you want one click of the knob to increase or decrease the |
|  | # volume. I don't think that non-integer values work here, but you're welcome |
|  | # to try. |
|  | VOLUME\_INCREMENT = 1 |
|  |  |
|  | # (END SETTINGS) |
|  | # |
|  |  |
|  |  |
|  | # When the knob is turned, the callback happens in a separate thread. If |
|  | # those turn callbacks fire erratically or out of order, we'll get confused |
|  | # about which direction the knob is being turned, so we'll use a queue to |
|  | # enforce FIFO. The callback will push onto a queue, and all the actual |
|  | # volume-changing will happen in the main thread. |
|  | QUEUE = Queue() |
|  |  |
|  | # When we put something in the queue, we'll use an event to signal to the |
|  | # main thread that there's something in there. Then the main thread will |
|  | # process the queue and reset the event. If the knob is turned very quickly, |
|  | # this event loop will fall behind, but that's OK because it consumes the |
|  | # queue completely each time through the loop, so it's guaranteed to catch up. |
|  | EVENT = threading.Event() |
|  |  |
|  | def debug(str): |
|  | if not DEBUG: |
|  | return |
|  | print(str) |
|  |  |
|  | class RotaryEncoder: |
|  | """ |
|  | A class to decode mechanical rotary encoder pulses. |
|  |  |
|  | Ported to RPi.GPIO from the pigpio sample here: |
|  | http://abyz.co.uk/rpi/pigpio/examples.html |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self, gpioA, gpioB, callback=None, buttonPin=None, buttonCallback=None): |
|  | """ |
|  | Instantiate the class. Takes three arguments: the two pin numbers to |
|  | which the rotary encoder is connected, plus a callback to run when the |
|  | switch is turned. |
|  |  |
|  | The callback receives one argument: a `delta` that will be either 1 or -1. |
|  | One of them means that the dial is being turned to the right; the other |
|  | means that the dial is being turned to the left. I'll be damned if I know |
|  | yet which one is which. |
|  | """ |
|  |  |
|  | self.lastGpio = None |
|  | self.gpioA = gpioA |
|  | self.gpioB = gpioB |
|  | self.callback = callback |
|  |  |
|  | self.gpioButton = buttonPin |
|  | self.buttonCallback = buttonCallback |
|  |  |
|  | self.levA = 0 |
|  | self.levB = 0 |
|  |  |
|  | GPIO.setmode(GPIO.BCM) |
|  | GPIO.setup(self.gpioA, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP) |
|  | GPIO.setup(self.gpioB, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP) |
|  |  |
|  | GPIO.add\_event\_detect(self.gpioA, GPIO.BOTH, self.\_callback) |
|  | GPIO.add\_event\_detect(self.gpioB, GPIO.BOTH, self.\_callback) |
|  |  |
|  | if self.gpioButton: |
|  | GPIO.setup(self.gpioButton, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP) |
|  | GPIO.add\_event\_detect(self.gpioButton, GPIO.FALLING, self.\_buttonCallback, bouncetime=500) |
|  |  |
|  |  |
|  | def destroy(self): |
|  | GPIO.remove\_event\_detect(self.gpioA) |
|  | GPIO.remove\_event\_detect(self.gpioB) |
|  | GPIO.cleanup() |
|  |  |
|  | def \_buttonCallback(self, channel): |
|  | self.buttonCallback(GPIO.input(channel)) |
|  |  |
|  | def \_callback(self, channel): |
|  | level = GPIO.input(channel) |
|  | if channel == self.gpioA: |
|  | self.levA = level |
|  | else: |
|  | self.levB = level |
|  |  |
|  | # Debounce. |
|  | if channel == self.lastGpio: |
|  | return |
|  |  |
|  | # When both inputs are at 1, we'll fire a callback. If A was the most |
|  | # recent pin set high, it'll be forward, and if B was the most recent pin |
|  | # set high, it'll be reverse. |
|  | self.lastGpio = channel |
|  | if channel == self.gpioA and level == 1: |
|  | if self.levB == 1: |
|  | self.callback(1) |
|  | elif channel == self.gpioB and level == 1: |
|  | if self.levA == 1: |
|  | self.callback(-1) |
|  |  |
|  | class VolumeError(Exception): |
|  | pass |
|  |  |
|  | class Volume: |
|  | """ |
|  | A wrapper API for interacting with the volume settings on the RPi. |
|  | """ |
|  | MIN = VOLUME\_MIN |
|  | MAX = VOLUME\_MAX |
|  | INCREMENT = VOLUME\_INCREMENT |
|  |  |
|  | def \_\_init\_\_(self): |
|  | # Set an initial value for last\_volume in case we're muted when we start. |
|  | self.last\_volume = self.MIN |
|  | self.\_sync() |
|  |  |
|  | def up(self): |
|  | """ |
|  | Increases the volume by one increment. |
|  | """ |
|  | return self.change(self.INCREMENT) |
|  |  |
|  | def down(self): |
|  | """ |
|  | Decreases the volume by one increment. |
|  | """ |
|  | return self.change(-self.INCREMENT) |
|  |  |
|  | def change(self, delta): |
|  | v = self.volume + delta |
|  | v = self.\_constrain(v) |
|  | return self.set\_volume(v) |
|  |  |
|  | def set\_volume(self, v): |
|  | """ |
|  | Sets volume to a specific value. |
|  | """ |
|  | self.volume = self.\_constrain(v) |
|  | output = self.amixer("set 'Digital' unmute {}%".format(v)) |
|  | self.\_sync(output) |
|  | return self.volume |
|  |  |
|  | def toggle(self): |
|  | """ |
|  | Toggles muting between on and off. |
|  | """ |
|  | if self.is\_muted: |
|  | output = self.amixer("set 'Digital' unmute") |
|  | else: |
|  | # We're about to mute ourselves, so we should remember the last volume |
|  | # value we had because we'll want to restore it later. |
|  | self.last\_volume = self.volume |
|  | output = self.amixer("set 'Digital' mute") |
|  |  |
|  | self.\_sync(output) |
|  | if not self.is\_muted: |
|  | # If we just unmuted ourselves, we should restore whatever volume we |
|  | # had previously. |
|  | self.set\_volume(self.last\_volume) |
|  | return self.is\_muted |
|  |  |
|  | def status(self): |
|  | if self.is\_muted: |
|  | return "{}% (muted)".format(self.volume) |
|  | return "{}%".format(self.volume) |
|  |  |
|  | # Read the output of `amixer` to get the system volume and mute state. |
|  | # |
|  | # This is designed not to do much work because it'll get called with every |
|  | # click of the knob in either direction, which is why we're doing simple |
|  | # string scanning and not regular expressions. |
|  | def \_sync(self, output=None): |
|  | if output is None: |
|  | output = self.amixer("get 'Digital'") |
|  |  |
|  | lines = output.readlines() |
|  | if DEBUG: |
|  | strings = [line.decode('utf8') for line in lines] |
|  | debug("OUTPUT:") |
|  | debug("".join(strings)) |
|  | last = lines[-1].decode('utf-8') |
|  |  |
|  | # The last line of output will have two values in square brackets. The |
|  | # first will be the volume (e.g., "[95%]") and the second will be the |
|  | # mute state ("[off]" or "[on]"). |
|  | i1 = last.rindex('[') + 1 |
|  | i2 = last.rindex(']') |
|  |  |
|  | self.is\_muted = last[i1:i2] == 'off' |
|  |  |
|  | i1 = last.index('[') + 1 |
|  | i2 = last.index('%') |
|  | # In between these two will be the percentage value. |
|  | pct = last[i1:i2] |
|  |  |
|  | self.volume = int(pct) |
|  |  |
|  | # Ensures the volume value is between our minimum and maximum. |
|  | def \_constrain(self, v): |
|  | if v < self.MIN: |
|  | return self.MIN |
|  | if v > self.MAX: |
|  | return self.MAX |
|  | return v |
|  |  |
|  | def amixer(self, cmd): |
|  | p = subprocess.Popen("amixer {}".format(cmd), shell=True, stdout=subprocess.PIPE) |
|  | code = p.wait() |
|  | if code != 0: |
|  | raise VolumeError("Unknown error") |
|  | sys.exit(0) |
|  |  |
|  | return p.stdout |
|  |  |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  |  |
|  | gpioA = GPIO\_A |
|  | gpioB = GPIO\_B |
|  | gpioButton = GPIO\_BUTTON |
|  |  |
|  | v = Volume() |
|  |  |
|  | def on\_press(value): |
|  | v.toggle() |
|  | print("Toggled mute to: {}".format(v.is\_muted)) |
|  | EVENT.set() |
|  |  |
|  | # This callback runs in the background thread. All it does is put turn |
|  | # events into a queue and flag the main thread to process them. The |
|  | # queueing ensures that we won't miss anything if the knob is turned |
|  | # extremely quickly. |
|  | def on\_turn(delta): |
|  | QUEUE.put(delta) |
|  | EVENT.set() |
|  |  |
|  | def consume\_queue(): |
|  | while not QUEUE.empty(): |
|  | delta = QUEUE.get() |
|  | handle\_delta(delta) |
|  |  |
|  | def handle\_delta(delta): |
|  | if v.is\_muted: |
|  | debug("Unmuting") |
|  | v.toggle() |
|  | if delta == 1: |
|  | vol = v.up() |
|  | else: |
|  | vol = v.down() |
|  | print("Set volume to: {}".format(vol)) |
|  |  |
|  | def on\_exit(a, b): |
|  | print("Exiting...") |
|  | encoder.destroy() |
|  | sys.exit(0) |
|  |  |
|  | debug("Volume knob using pins {} and {}".format(gpioA, gpioB)) |
|  |  |
|  | if gpioButton != None: |
|  | debug("Volume button using pin {}".format(gpioButton)) |
|  |  |
|  | debug("Initial volume: {}".format(v.volume)) |
|  |  |
|  | encoder = RotaryEncoder(GPIO\_A, GPIO\_B, callback=on\_turn, buttonPin=GPIO\_BUTTON, buttonCallback=on\_press) |
|  | signal.signal(signal.SIGINT, on\_exit) |
|  |  |
|  | while True: |
|  | # This is the best way I could come up with to ensure that this script |
|  | # runs indefinitely without wasting CPU by polling. The main thread will |
|  | # block quietly while waiting for the event to get flagged. When the knob |
|  | # is turned we're able to respond immediately, but when it's not being |
|  | # turned we're not looping at all. |
|  | # |
|  | # The 1200-second (20 minute) timeout is a hack; for some reason, if I |
|  | # don't specify a timeout, I'm unable to get the SIGINT handler above to |
|  | # work properly. But if there is a timeout set, even if it's a very long |
|  | # timeout, then Ctrl-C works as intended. No idea why. |
|  | EVENT.wait(1200) |
|  | consume\_queue() |
|  | EVENT.clear() |

chmod +x ~/bin/monitor-volume

TEST

1. Start SSH PuTTY remote access to RPi and use alsamixer command to open a GUI to the mixer.Turning the encoder left and right should decrease or increase the green/white/red coloured vertical bar “digital”.
2. Alternative :

*You can also run this script in the foreground just to test it. Edit the script and temporarily change DEBUG to True so that you can see what's going on, then simply run it with monitor-volume. When you turn or press the knob, the script should report what it's doing. (Make sure that the volume*increases*rather than*decreases*when you turn it to the right, and if it doesn't, swap your A and B pins, or just swap their numbers in the script.)*

*You should also play around with the constants defined at the top of the script. Naturally, if you picked other pins, you'll want to tell the script which GPIO pins to use. If your rotary encoder doesn't act like a button, or if you didn't hook up the button and don't care about it, you can set GPIO\_BUTTON to None. But you may also want to change the minimum and maximum volumes (they're percentages, so they should be between 1 and 100) and the increment (how many percentage points the volume increases/decreases with each "click" of the knob).*

If it's working the way you want, you can proceed to the next step: running monitor-volume automatically in the background whenever your Pi starts.

## Creating a systemd service

nano ~/monitor-volume.service

# paste in the contents of monitor-volume.service, save/exit nano

[**monitor-volume.service**](https://gist.github.com/savetheclocktower/9b5f67c20f6c04e65ed88f2e594d43c1#file-monitor-volume-service)

|  |  |
| --- | --- |
|  | [Unit] |
|  | Description=Volume knob monitor |
|  |  |
|  | [Service] |
|  | User=pi |
|  | Group=pi |
|  | ExecStart=/home/pi/bin/monitor-volume |
|  |  |
|  | [Install] |
|  | WantedBy=multi-user.target |

chmod +x ~/monitor-volume.service

sudo mv ~/monitor-volume.service /etc/systemd/system

sudo systemctl enable monitor-volume

Created symlink /etc/systemd/system/multi-user.target.wants/monitor-volume.service → /etc/systemd/system/monit or-volume.service.

sudo systemctl start monitor-volume

If that worked right, then you just told Raspbian to start up that script in the background on every boot (enable), and also to start it right now (start). At this point, and on every boot after this, your volume knob should Just Work.

For other projects available GPIO : **5,6,12**,21 (pin**29,31,32**,40) e.g. START, SHUTDOWN button



# Amp2 uses GPIO2,3 (pin 3,5) for I2C configuration

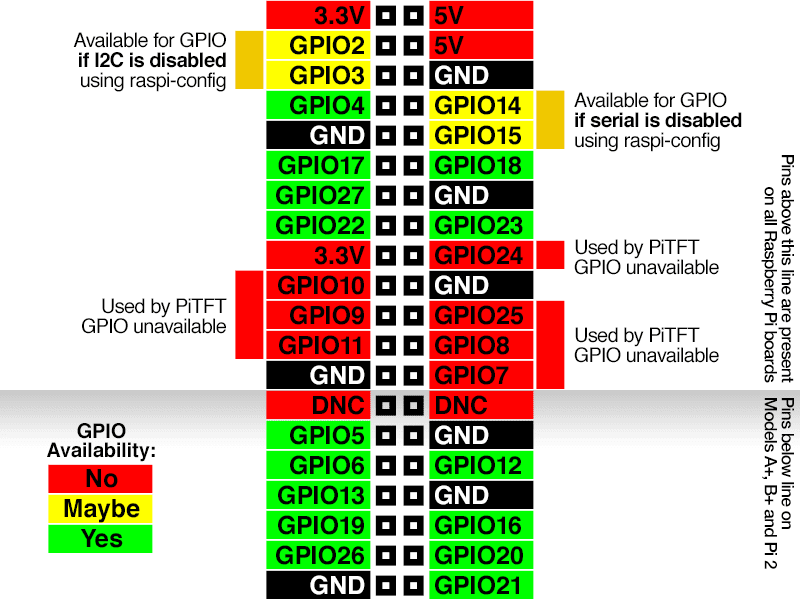
# Amp2 uses GPIO4 (pin7) as MUTE

# Amp2 uses GPIO 18 (pin12) for sound interface

# << NEVER USE THIS PINS (27,28 >Pi-RESERVED!!)

# Amp2 uses GPIO 19,20,21 (pin 35,38,40) for sound interface





# FAQ

# **[Setting Static IP on Raspberry Pi 3 Model B+ used as a ROON client](https://raspberrypi.stackexchange.com/questions/83107/setting-static-ip-on-raspberry-pi-3-model-b)** ?

**There is no NEED to set static IP addresses** - I haven't done this since 1980, unless configuring a DHCP server. I don't want to know what the IP of my 7 Pi or 3 Macs or other devices are, and don't care - I access them **by name**, and let the routers do their job! But using SSH puttyy you need to know the IP address once and save the profile (look in the LAN lease list of the router or sudo ifconfig eth0)

The **if** family of tools including ifconfig are being deprecated and replaced by the newer ip commands so you can use any one of the following from the command line to determine your IP address:

sudo ip addr show

or

sudo hostname --ip-address

or if you still want to use ifconfig, and it is not already installed

sudo apt-get install wireless-tools

sudo ifconfig -a

**but preferentially you can use the MAC address of your RPi 3 to reserve an IP address on your router (section Address Reservation)**

**Addendum**

Quick start (5 minute install for experienced RPi users)

pi@raspi-amp2:~ $ = pi@raspberrypi:~ $ (default) = hostname

sudo raspi config

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent

permitted by applicable law.

Last login: Sat Jun 1 11:08:42 2019

pi@raspi-amp2:~ $ mkdir ~/bin

pi@raspi-amp2:~ $ nano ~/bin/monitor-volume #don’t forget to drop script in it

pi@raspi-amp2:~ $ sudo apt-get install python3 python3-rpi.gpio

*[[Reading package lists... Done*

*Building dependency tree*

*Reading state information... Done*

*python3 is already the newest version (3.5.3-1).*

*python3 set to manually installed.*

*The following package was automatically installed and is no longer required:*

*libopus0*

*Use 'sudo apt autoremove' to remove it.*

*The following NEW packages will be installed:*

*python3-rpi.gpio*

*0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.*

*Need to get 23.0 kB of archives.*

*After this operation, 74.8 kB of additional disk space will be used.*

*Do you want to continue? [Y/n] y*

*Get:1 http://archive.raspberrypi.org/debian stretch/main armhf python3-rpi.gpio armhf 0.6.3~stretch-1 [23.0 kB]*

*Fetched 23.0 kB in 0s (180 kB/s)*

*Selecting previously unselected package python3-rpi.gpio.*

*(Reading database ... 42927 files and directories currently installed.)*

*Preparing to unpack .../python3-rpi.gpio\_0.6.3~stretch-1\_armhf.deb ...*

*Unpacking python3-rpi.gpio (0.6.3~stretch-1) ...*

*Setting up python3-rpi.gpio (0.6.3~stretch-1) ...]]*

pi@raspi-amp2:~ $ chmod +x ~/bin/monitor-volume

pi@raspi-amp2:~ $ nano ~/monitor-volume.service #don’t forget to drop script in it

pi@raspi-amp2:~ $ chmod +x ~/monitor-volume.service

pi@raspi-amp2:~ $ sudo mv ~/monitor-volume.service /etc/systemd/system

pi@raspi-amp2:~ $ sudo systemctl enable monitor-volume

Created symlink /etc/systemd/system/multi-user.target.wants/monitor-volume.service → /etc/systemd/system/monitor-volume.service.

pi@raspi-amp2:~ $ sudo systemctl start monitor-volume

pi@raspi-amp2:~ $

Done

# How to add a power button to your Raspberry Pi

Because you should always safely shut down your Pi.

<https://howchoo.com/g/mwnlytk3zmm/how-to-add-a-power-button-to-your-raspberry-pi>

# Build a simple Raspberry Pi LED power/status indicator

<https://howchoo.com/g/ytzjyzy4m2e/build-a-simple-raspberry-pi-led-power-status-indicator>

use pin6=GND ,pin8 GPIO= UART0TXD and a resistor 330R (to protect the Pi from overvoltage!)

1. Raspbian for Raspberry Pi has SSH server DISABLED by default. Use sudo raspi-config command to enable is first or add a SSH (empty) file in the root of the SD card. [↑](#footnote-ref-1)
2. Eventually change page layout in Word document to make copy/paste easier. [↑](#footnote-ref-2)