RaspberryPi Git Server

25/03/2020 By An7ar35

Contents

1 Introduction

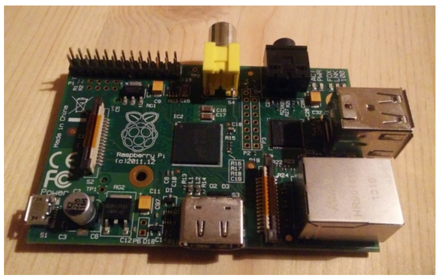
The idea with this little project is to get a localised networked Git server to mirror existing repositories but also enable some backup/synchronisation of other text based content (dot files, TODO lists, etc...).

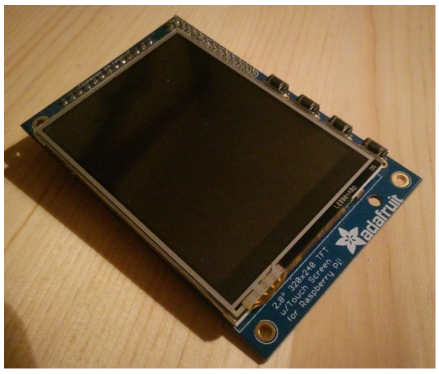
Text based TODO list are simple to maintain and can easily be managed in Git. It's also a bonus that with a bit of Conky magic, these can be displayed on the desktop in Linux. If you are interested, my basic configuration is in the appendix A.

Dot files are also a great candidate for version control. This only hiccup is that the important ones (for me anyway) are located in the root of the user's directory so initiating a git repo there would not be appropriate. It is better to create a dedicated directory that houses all of these files and then to just create symbolic links (`ln -s <source> <target>`) to them.

With all that said, let's get on with it...

2 Hardware





The rather dated RaspberryPi, TFT assembly and case were bought years ago for a completely different project that never saw any ground so the hardware laid in a box gathering dust to my great shame... Well, no longer! It shan't be a testament to un-lived silicon-based potential for long!

To note that the model B is running an ARM11 based 32bit CPU (with ARM6Z instruction sets) running at a casual 700Mhz. The board is so 'vintage' (released in Q2 of 2012) it is not featured on the official site's shop any more.

Here's the hardware list:

- 1 × RaspberryPi B
- 1 × Adafruit PiTFT 2.8" Resistive Touchscreen Display
- 1 × Piromoni PiTFT case
- 1 × 5V PSU salvaged from the box of forgotten adapters
- 1 × 32Gb SD card which is admittedly overkill but that's all I could find in my box of loose computer crap
- 1 × 128Gb USB stick
- 1 × ethernet patch cable (something that supports at least 10/100 Mbit/s - either cat 5 or 6 is fine)

3 System setup

3.1 Arch Linux

I chose the Arch Linux variant for the RaspberryPi out of familiarity mostly as well as wanting a lightweight deployment. I'm not running on bleeding edge hardware after all!

3.1.1 Installing Arch

First things first; the Arch Linux zipped image need to be acquired [link].

Then it's just a matter of going through the instructions found on the Arch Linux ARM6 installation guide page.

In summary; the SD card is to be partitioned as such:

1. 100Mb '`W95 FAT32 (LBA)`' boot partition (`/boot`)
2. rest of the space as a '`ext4`' root partition (`/`)

Then the image needs to be extracted using `bsdtar` onto the root partition and, finally, the files in `/boot/*` must be moved to the boot partition.

Once all that is done the Pi can be booted up with the SD card whilst connected to the LAN and we can login remotely with SSH: `ssh alarm@192.168.X.X` (password: "alarm"). You'll need to know the IP address of the Pi. Routers will have in their administration interface a way to see what devices are connected to them and what IP are assigned for those.

3.1.2 Updating Arch

To get all the system updates, access to the root account is needed: `su` (password: "root")

Now the pacman keyring can be initialized, the package list downloaded and the system updated:

```
$ pacman-key --init
$ pacman-key --populate archlinuxarm
$ pacman -Syyu
```

Next on the list, once the updates are done, is the raspberry firmwares:

```
$ pacman -S raspberrypi-firmware
```

Some extras while we're here:

```
$ pacman -S htop wget usbutils gpio-utils wiringpi
```

...and reboot (`reboot`).

3.2 Getting the TFT panel to work

```
Arch Linux 4.19.106-1-ARCH (tty1)
alarmpi login:
```

Now the base system done, displaying stuff to the little screen would be a nice addition since it's there. Unfortunately, there isn't a whole lot of updated Arch-specific instructions in that subject but once you finally get enough pieces of the puzzle from various forums and articles, it is not actually that bad.

It needs to be noted that ArchPi for ARM6 (32bit) includes all the kernel overlays normally found in the official RaspberryPi distro Raspbian. These enable all the board's specific functionalities.

Make sure you are on the root account for all that (`su`).

```
$ modprobe -a fbtft
```

Then edit the `/boot/config.txt` file (use `nano`) and add the following:

```
dtoverlay=pitft28-resistive ,rotate=90,speed=64000000,fps=30
dtparam=audio=on
```

Lines 2 just enables the audio. As we are already in there, might as well...

To get the console to show on the display the `/boot/cmdline.txt` file needs to be edited and the following added to the end of the line:

```
fbcon=map:10 fbcon=font:VGA8x8
```

Finally, reboot and the TFT panel should come to life. If you switch to root you can output text to the TFT console with:

```
$ echo "Hello world!" > /dev/tty1
```

3.3 Assigning the shutdown functionality to a TFT panel button

A little button to safely shutdown the system thus avoiding data corruption and even premature drive death can be quite useful... The other alternative is to trigger the shutdown from within the system remotely via SSH which is not always practical.

As with the TFT display, make sure you are on the root account.

In `/boot/config.txt` add the line:

```
dtoverlay=gpio-shutdown ,gpio_pin=23
```

Pin # `23` is located under the bottom-left button on that PiTFT display. That leaves us with another 3 buttons to play around with later.

After a reboot the button should now trigger a shutdown-halt after which it will be safe to pull the power out from the device.

3.4 sudo

`sudo` will be useful to have users outside of root be able to run commands requiring elevated privileges. Let's install that:

```
$ pacman -S sudo
```

Next some light editing of the configuration is required:

```
$ visudo
```

Find the line:

```
# %sudo ALL=(ALL) ALL
```

...and uncomment it to enable `sudo` access to any user in the "sudo" group:

```
%sudo   ALL=(ALL) ALL
```

Now to actually create that "sudo" group:

```
$ groupadd sudo
```

3.5 Users and Groups

Just for the sake of mild security, the default `alarm` user needs to be replaced by something else. Since I'd like that user to have `sudo` privileges, it also needs to be added to the "sudo" group.

Here are the steps to make a new user "`pi`":

```
$ useradd -m -g users -G sudo -s /bin/bash pi
$ passwd pi
```

Reboot (`reboot`) and login via SSH with the new user: `ssh pi@<ip address>`

Finally, to remove the old "alarm" account, change back into the root account (`su`) and:

```
$ userdel -r alarm
```

3.6 RSA credentials for SSH client login

So that the client(s) systems on the LAN can connect to the Raspberry Pi server without having to put a password every time SSH keys need to be generated for each.

The steps for each user@client are:

1. Generate keys: `ssh-keygen -t rsa`

2. Copy the public key to the "`pi`" user on the Raspberry Pi: `ssh-copy-id pi@<ip address>`

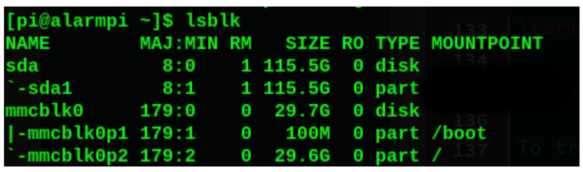
After that it is possible to remove the SSH passworded logins option altogether. To do that, use an editor to open `/etc/ssh/sshd_config` on the RaspberryPi and change/uncomment the following options as:

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

3.7 Permanent USB storage

To treat the USB stick as a permanent storage device, it needs to be automatically mounted at boot.

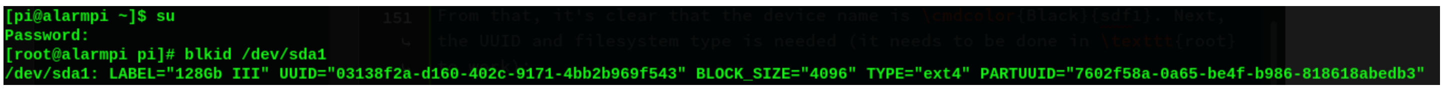
Once plugged-in, the device name given by the kernel needs to be found. `lsblk` can help with that (see right).



```
[pi@alarmpi ~]$ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda           8:0    1 115.5G  0 disk
`-sda1        8:1    1 115.5G  0 part
mmcblk0     179:0    0  29.7G  0 disk
|-mmcblk0p1 179:1    0   100M  0 part /boot
`-mmcblk0p2 179:2    0  29.6G  0 part /
```

From that, it's clear that the device name is `sdf1`. Next, the UUID and filesystem type is needed (it needs to be done in `root` to work):

```
$ blkid /dev/sdf1
```



```
[pi@alarmpi ~]$ su
Password:
[root@alarmpi pi]# blkid /dev/sda1
/dev/sda1: LABEL="128Gb III" UUID="03138f2a-d160-402c-9171-4bb2b969f543" BLOCK_SIZE="4096" TYPE="ext4" PARTUUID="7602f58a-0a65-be4f-b986-818618abedb3"
```

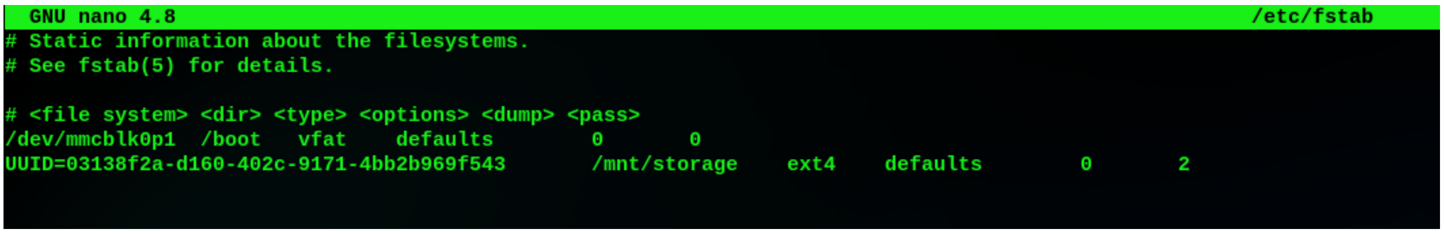
A mount point needs to be created and its permission set so that the new user has ownership:

```
$ mkdir /mnt/storage
$ chown pi /mnt/storage
```

Note

In my case I had already formatted the USB stick with an `ext4` filesystem. As it is a Linux filesystem I don't need to install any packages to be able to read/write to it.

Now that all the needed information is gathered, the USB storage can be added to the `/etc/fstab` file. All that is required is the UUID, the mount point, the filesystem, and various options flags:



```
  GNU nano 4.8                                                      /etc/fstab
# Static information about the filesystems.
# See fstab(5) for details.

# <file system> <dir> <type> <options> <dump> <pass>
/dev/mmcblk0p1  /boot   vfat    defaults        0       0
UUID=03138f2a-d160-402c-9171-4bb2b969f543       /mnt/storage    ext4    defaults        0       2
```

On the next reboot the USB storage should automatically be mounted on `/mnt/storage`.

4 Git

4.1 Basic Client-Server configuration

First, Git must me installed on the RaspberryPi:

```
$ sudo pacman -S git
```

Then, Git repositories can be created:

```
$ mkdir /mnt/storage/git-repos/<repo name>
$ cd /mnt/storage/git-repos/<repo name>
$ git init --bare
```

From thereon, the RaspberryPi Git server will be designated as ”`lan-git`” on the clients so as to be representative.

Now on the client machine from the existing ”`<repo name>`” directory:

```
$ git remote add lan-git ssh://pi@<ip address>:/mnt/storage/git-repos/<repo
  name>
```

To check the details have been added correctly: `git remote -v`

Ooops, ”fatal error”

If an error where the the remote directory could not be read/seen as a git repository make sure that:

- you have access rights,
- the repository exists on the Git server,
- the remote access details are correct (ip address?),
- the remote path is correct.

If there is a typo, the remote repository details can be removed from the client’s git repository by: `git remote rm lan-git`

All we need to do now is to upload the repository’s content to the RaspberryPi server’s remote git folder:

Client repository with an upstream already defined

To push the content (inc. all branches) to `lan-git`:

```
$ git push lan-git --all
```

Client repository without an upstream defined

If the repository on the client is bran-new then `lan-git` must be specified as the default upstream remote:

```
$ git push --set-upstream lan-git --all
```

4.2 Unifying remotes into 1 call

If there are multiple remote repositories to which the version controlled content is pushed to, a ”catch-all” designation might be appropriate.

For example, let’s say we have 2 remotes. Pushing to all of them would require a call to `git push ...` 2 times.

In the repository’s folder there is a sub-directory called ”.git/”. Within it the `config` file for the repository resides.

Just add a section named ” `[remote "all"]` ” and all the remote repositories’ URLs. As an example:

```
[remote "all"]
url = https://user@domain/path/to/repository.git
url = ssh://user@ip-address:/path/to/repository
```

Alternatively, in the case where only 1 of the 2 repository should be pulled from but both can be pushed to, the `[remote "origin"]` section can be edited as such instead:

```
[remote "origin"]
url = https://user@domain/path/to/repository.git
pushurl = ssh://user@ip-address:/path/to/repository
fetch = +refs/heads/*:refs/remotes/origin/*
```

5 Software

5.1 Some button testing with python

Developing with Python3

For an easier time programming with the RaspberryPi, Python is a great starting option for any budding developer out there. Just install it and the needed libraries with:

`pacman -S python3 python-pip`

`pip3 install gpiozero` (new library)

`pip3 install rpi.gpio` (old, depreciated library)

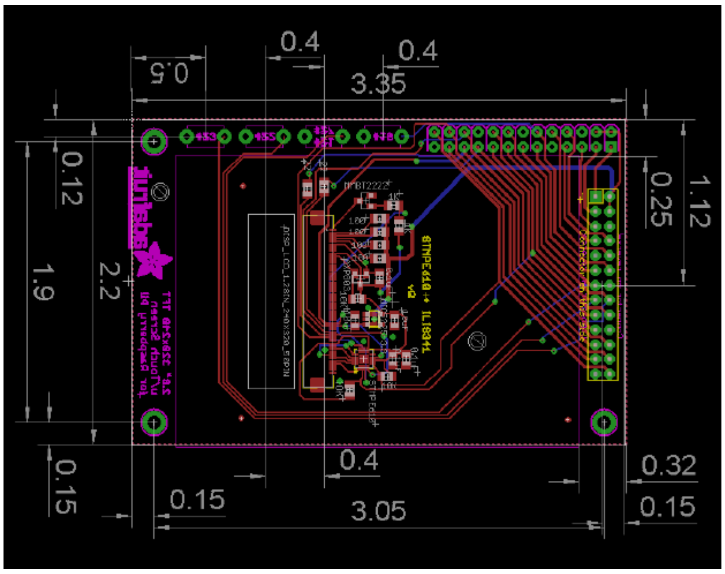
In order to check the buttons were working a small python program was written. With this it became clear that something was amiss... Whilst the second (#22) and fourth (#18) button on the TFT board worked as expected, the third (#21) one didn't register.

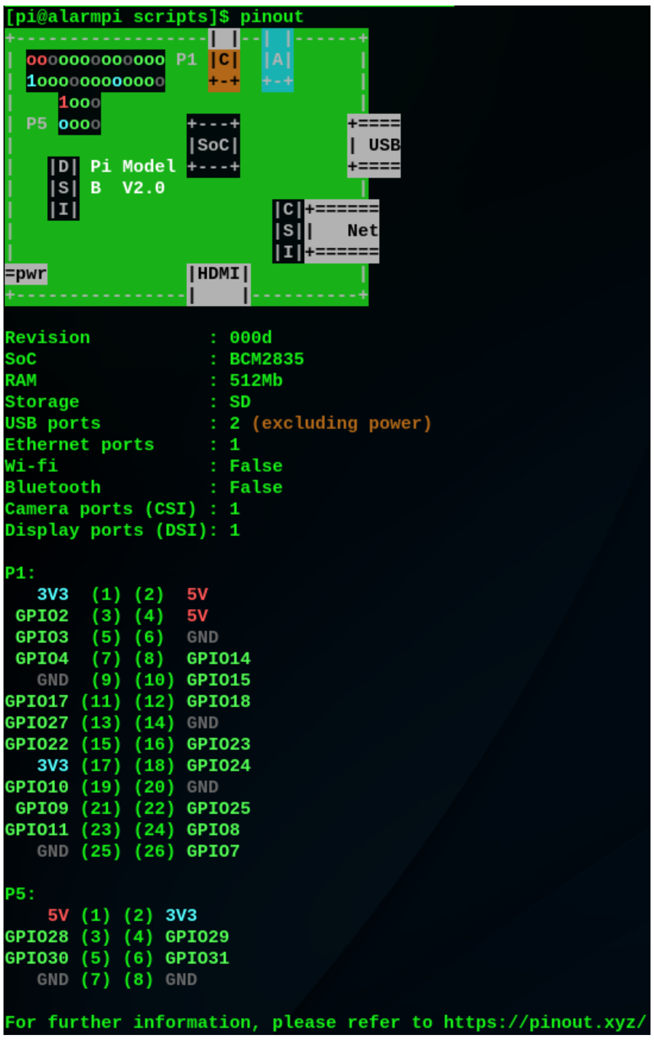
Alright then, I checked the soldering more closely and everything seemed OK on the surface. Next, the multimeter to check the contacts were actually fine. Again, everything tested as working.

Mystery mystery mystery...

Next was checking the trace routing and on what pin it landed back on the Raspberry Pi with the help of the TFT tracing (see below) and the `pinout` output (see right).

It turns out that the TFT is either mislabelled or was designed for the first version of the Raspberry Pi. The actual pin it lands on is #13 (so **GPIO #27**)!





```
[pi@alarmpi scripts]$ pinout
+------------------| |--| |------+
| ooooooooooooo P1 |C|  |A|      |
| 1oooooooooooo    +-+  +-+      |
|    1ooo                        |
| P5 oooo        +---+          +====
|                |SoC|          | USB
|   |D| Pi Model +---+          +====
|   |S| B  V2.0                  |
|   |I|                  |C|+======
|                        |S||   Net
|                        |I|+======
=pwr             |HDMI|          |
+----------------|    |----------+

Revision           : 000d
SoC                : BCM2835
RAM                : 512Mb
Storage            : SD
USB ports          : 2 (excluding power)
Ethernet ports     : 1
Wi-fi              : False
Bluetooth          : False
Camera ports (CSI) : 1
Display ports (DSI): 1

P1:
    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

P5:
     5V  (1) (2)  3V3
 GPIO28  (3) (4)  GPIO29
 GPIO30  (5) (6)  GPIO31
    GND  (7) (8)  GND

For further information, please refer to https://pinout.xyz/
```

Adjustment to the testing program (see appendix C.1) confirmed that the pin was indeed GPIO #27.

5.2 Button listener in Python using the `sysfs` ABI

The test script can be modified to now trigger more useful things than just messages of what buttons are being pressed. One particular use case is switching the backlight on/off for some power saving.

One way to do just that is to access to the Linux GPIO `sysfs` Application Binary Interface (ABI). Although officially depreciated in favour of character devices, it is still available for the sake of backwards compatibility.

The backlight state can be checked and controlled with:

`/sys/class/backlight/soc:backlight/bl_power`.

`0` is "on" and `1` is "off" so the binary value can be both read and written to by opening a file descriptor since in Linux everything is a file of some type. With this in mind the code for switching the backlight can be as follows:

```python
class Backlight:
    ON: Final = b'0'
    OFF: Final = b'1'

def switchBacklight():
    fd = os.open("/sys/class/backlight/soc:backlight/bl_power", os.O_RDWR)
    state = os.read(fd, 1)

    if state == Backlight.ON:
        os.write(fd, Backlight.OFF)
    else:
        os.write(fd, Backlight.ON)

    os.close(fd)
```

The second button (GPIO #22) can be assigned this functionality. As for the other remaining 2, some placeholders can be left for the time where they might be useful (see appendix C.2 for the code).

5.3 Other attempts and failures/problems encountered

The simplest approach is just to avoid using GPIO libraries (see appendix F for some of these) and use the old school `sysfs` interface.

In my case the interface, although existent, stopped updating values for.. well I don't exactly know. Earlier on on the first install it checked out and then, after later updates, it stopped tracking values.

This led me to a long and frustrating time experimenting with various approaches to accessing the GPIO information to no avail...

To summarise the things tried:

1. Direct from the shell access to the `sysfs` ABI,

2. A number of variation of the above using `poll` in C, setting up interrupt requests on the pins, and so on...
3. Using the `libgpiod` tools that come with the library to test events on pins (which also failed to detect value changes either),
4. Direct register access using `mmap` on both `/dev/mem` and `/dev/gpiomem` to access the GPIO register as a superuser (see appendix D). Looping on the all register pin values whilst pressing the buttons still failed to show changes.

It is important to mention that the python script still works despite everything else failing which is odd given that I reproduced the register access code from the `gpiozero` library[1] in C and got no value change on the button presses. Given more time and a second 64bit board to test with a newer kernel I would look into this problem in more details.

There might be an issue with the way the TFT board interferes with the button presses but that wouldn't explain the python script still working.

In any case this is where I get off the ride for the time being as the primary goal of this project is already fulfilled after all (Git server) even if there's a sour note to all this... I must have missed something...

[1] https://github.com/gpiozero/gpiozero/blob/master/gpiozero/pins/native.py

6 Sources and Further Reading

6.1 Documentation

- Arch Linux ARM installation guide
- Adafruit PiTFT 28 inch resistive touchscreen display installation guide PDF
- Linux kernel documentation: GPIO
- Linux kernel documentation: `sysfs`
- Direct register access in C

6.2 Posts & Articles

- An Introduction to chardev GPIO and Libgpiod on the Raspberry PI (Craig Peacock, 16 Oct 2018)
- GPIO Programming: Using the sysfs Interface (Jeff Tranter, 10 Jul 2019)
- How to Control GPIO Hardware from C or C++ (Jeff Tranter, 14 Aug 2019)
- Writting a Linux Kernel Module (Derek Molloy, 14 April 2015) - Parts 1, 2 and 3

6.3 Code Repositories

- Basic skeleton of a linux daemon written in C (Pascal Werkl)
- `libgpiod` library (README)
- `pigpio` library

Appendices

A Conky configuration

This configuration just gets some text files to display on the desktop without any bells or whistle.

Side note: KDE Plasma5

To have Conky show up on all virtual desktops start it with:

```
$ kstart5 --windowclass normal --alldesktops conky
```

```
# Create own window instead of using desktop
own_window
# own_window_class conky-semi
own_window_class normal
own_window yes
own_window_type normal
own_window_transparent yes
own_window_hints undecorated,below,skip_taskbar,skip_pager
own_window_argb_visual yes
own_window_argb_value 0

# Use double buffering (reduces flicker, may not work for everyone)
double_buffer yes

# fiddle with window
use_spacer none

# Update interval in seconds
update_interval 3.0

# Minimum size of text area
# minimum_size 800 600

# Draw shades?
draw_shades no

# Text
use_xft
xftalpha 1
draw_outline no
draw_borders no
font Hack:size=11

# Border/margins
stippled_borders 2
```

```
border_inner_margin 10

# Default colors and also border colors, grey90 == #e5e5e5
default_color grey

own_window_transparent yes

# Text alignment, other possible values are commented
#alignment top_left
alignment top_right
#alignment bottom_left
#alignment bottom_right

# Gap between borders of screen and text
gap_x 40
gap_y 20

# stuff after 'TEXT' will be formatted on screen
TEXT
${color cyan}TODO:

${color white}${execi 30 cat ~/git-repos/TODOs/conky/TODO.txt}

${color cyan}Projects:

${color white}${execi 30 cat ~/git-repos/TODOs/conky/PROJECTS.txt}
```

B Adding SSH keys to Web-based repositories

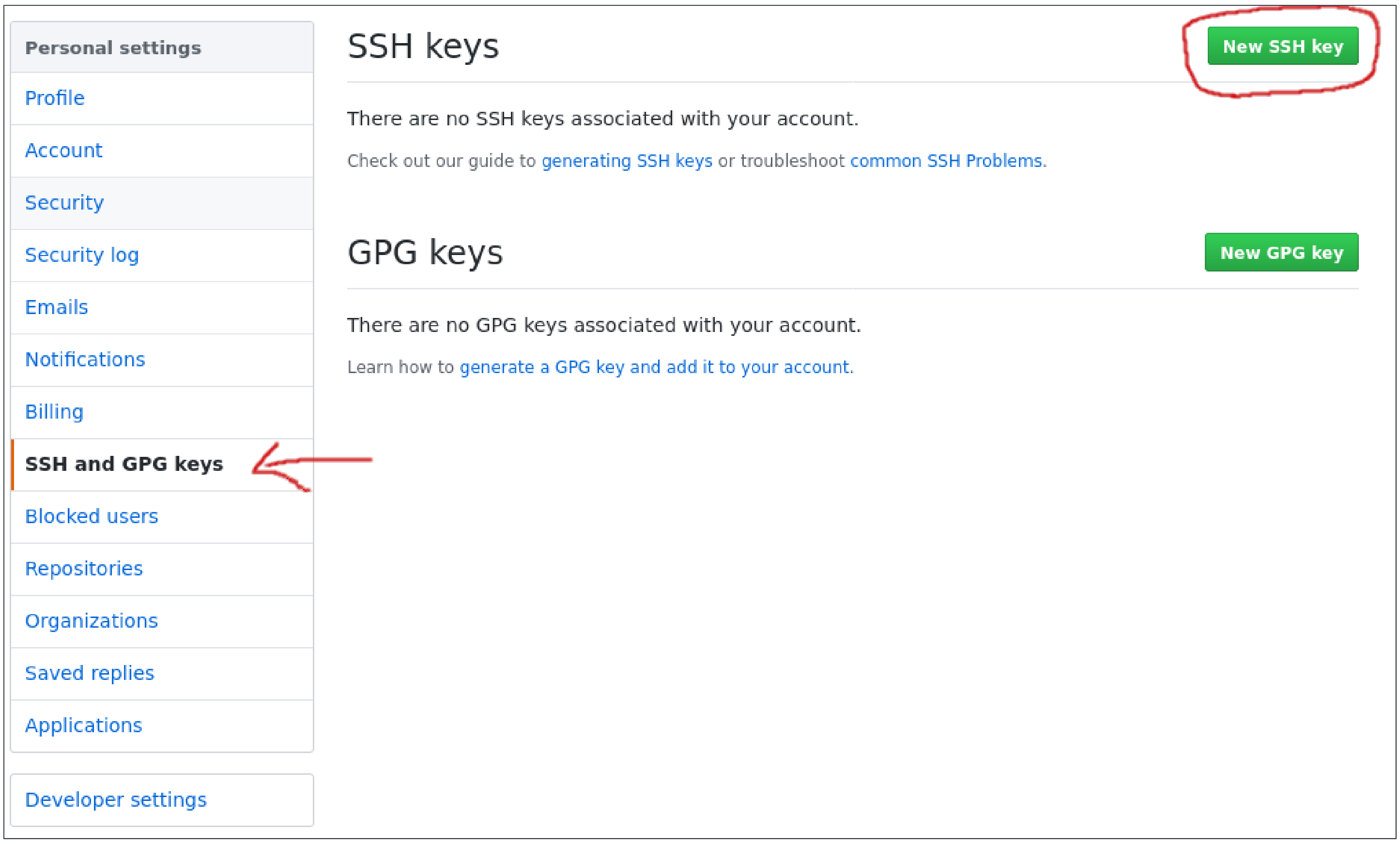
To use any cloud hosted repositories (GitHub/Bitbucket/Gitlab/etc...) an SSH key will need to be generated on the RaspberryPi server:

```
$ ssh-keygen -t rsa
```

The content of the `~/.ssh/id_rsa.pub` file usually needs to be copied onto a field in these cloud hosted Git sites.

B.1 Github

In the **`Settings`** page, there is an **`SSH and GPG keys`** entry where keys can be added.



There are no SSH keys associated with your account.

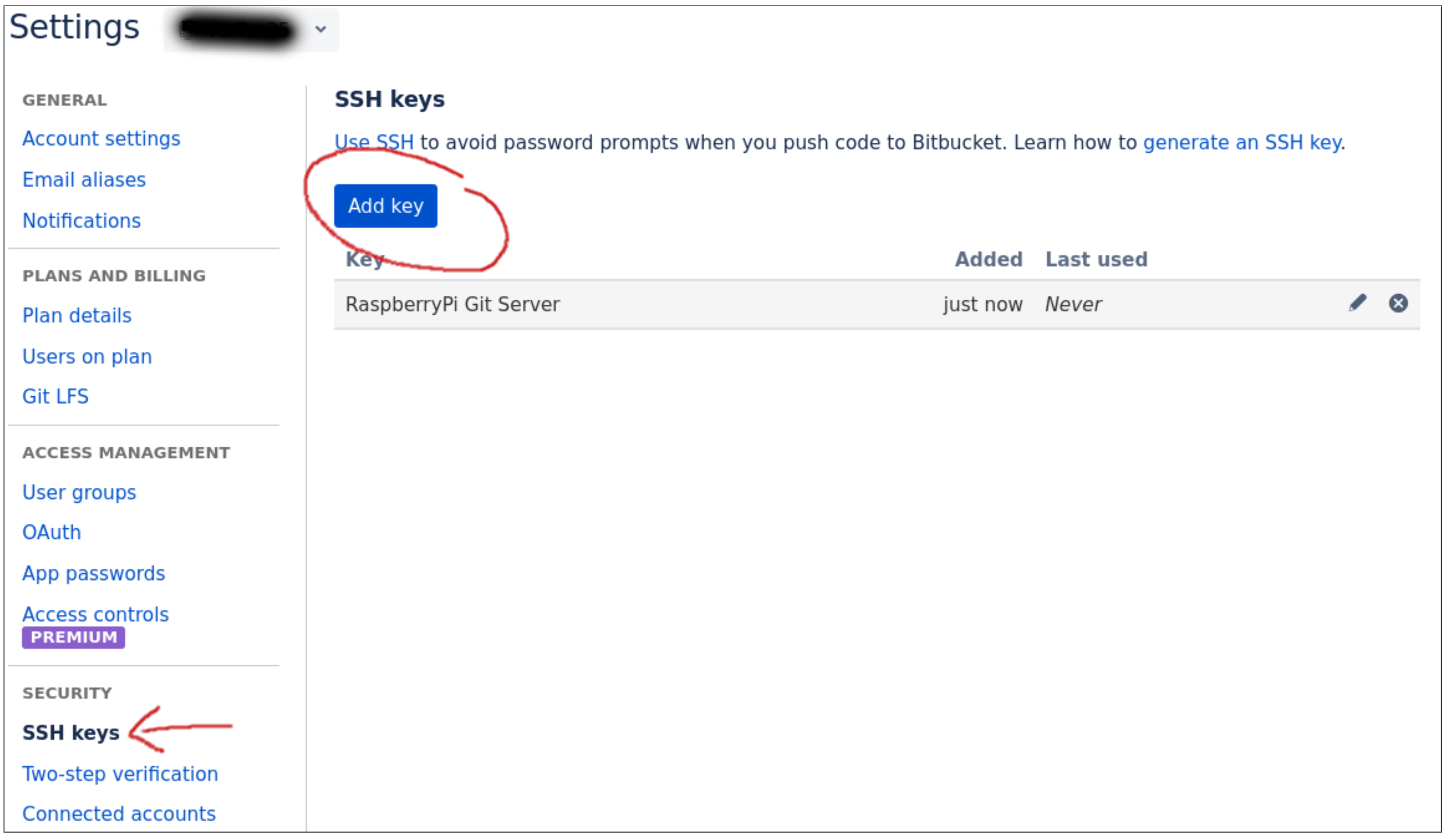
Check out our guide to generating SSH keys or troubleshoot common SSH Problems.

There are no GPG keys associated with your account.

Learn how to generate a GPG key and add it to your account.

B.2 Bitbucket

In the **`Bitbucket settings`** page, there is an **`SSH Keys`** entry where keys can be added.



Settings [redacted]

GENERAL
Account settings
Email aliases
Notifications

PLANS AND BILLING
Plan details
Users on plan
Git LFS

ACCESS MANAGEMENT
User groups
OAuth
App passwords
Access controls PREMIUM

SECURITY
SSH keys
Two-step verification
Connected accounts

SSH keys

Use SSH to avoid password prompts when you push code to Bitbucket. Learn how to generate an SSH key.

Add key

Key	Added	Last used
RaspberryPi Git Server	just now	*Never*

C Python programs

C.1 Button test program

code/pi–button–test.py

```python
# Pi—Buttons test program
# By An7ar35 (https://an7ar35.bitbucket.io)

from gpiozero import Button, PinNonPhysical

PinNonPhysical.printWarnings = False #This is not really working

def pressedB2():
    print("button 2 was pressed")

def pressedB3():
    print("button 3 was pressed")

def pressedB4():
    print("button 4 was pressed")


button2 = Button(22)
button3 = Button(27) #Marked as #21 on TFT board
button4 = Button(18)

while(True):
    button2.when_pressed = pressedB2
    button3.when_pressed = pressedB3
    button4.when_pressed = pressedB4
```

C.2 Button listener

code/pi–buttons.py

```python
#!/usr/bin/env python3

# Pi—Buttons listener program
# By An7ar35 (https://an7ar35.bitbucket.io)

from gpiozero import Button, PinNonPhysical
from typing import Final
from time import sleep
import os

PinNonPhysical.printWarnings = False

DELAY_TIME: Final = 60

class Backlight:
    ON: Final = b'0'
    OFF: Final = b'1'

try:
    button2: Final = Button(22)
    button3: Final = Button(27) #Marked as #21 on TFT board
    button4: Final = Button(18)
except PinNonPhysical:
    print("PinNonPhysical issue")

def switchBacklight():
    fd = os.open("/sys/class/backlight/soc:backlight/bl_power", os.O_RDWR)
    state = os.read(fd, 1)

    if state == Backlight.ON:
        os.write(fd, Backlight.OFF)
    else:
        os.write(fd, Backlight.ON)

    os.close(fd)

def pressedB3():
    print("button 3 is unassigned")

def pressedB4():
    print("button 4 is unassigned")

def main():
    try:
        while(True):
            button2.when_pressed = switchBacklight
            button3.when_pressed = pressedB3
            button4.when_pressed = pressedB4
    except Exception:
        print( "Problem starting... trying again in ", DELAY_TIME / 60, "mns." )
        sleep(DELAY_TIME)

main()
```

D Memory mapping (RaspberryPi Model B v2.0)

Using `sudo cat /proc/iomem`:

```
00000000-1bffffff : System RAM
00008000-00ffffff : Kernel code
01100000-012abfcf : Kernel data
20006000-20006fff : dwc_otg
20007000-20007eff : dma@7e007000
2000a000-2000a023 : watchdog@7e100000
2000b840-2000b87b : mailbox@7e00b840
2000b880-2000b8bf : mailbox@7e00b880
20100000-20100113 : watchdog@7e100000
20101000-20102fff : cprman@7e101000
20104000-2010400f : rng@7e104000
20200000-202000b3 : gpio@7e200000
20201000-202011ff : serial@7e201000
20201000-202011ff : serial@7e201000
20202000-202020ff : mmc@7e202000
20204000-202041ff : spi@7e204000
20212000-20212007 : thermal@7e212000
20215000-20215007 : aux@7e215000
20980000-2098ffff : dwc_otg
```

Just for the GPIO with `sudo cat /proc/iomem | grep gpio@`:

```
20200000-202000b3 : gpio@7e200000
```

E Developing with C/C++

To do more than just compile simple programs in C (or C++) on the Pi there are some stuff such that are going to be useful to do just that. Here are my recommended list of packages to get with `pacman`:

- `gcc` / `clang`
- `gdb`
- `cmake`
- `make`
- `perf`
- `patch`
- `automake`
- `autoconf`
- `autoconf-archive`
- `libtool`
- `pkg-config`
- `doxygen`
- `help2man`

Linux header packages

If developing with the legacy GPIO linux header strikes your fancy then you will need to install the following with `pacman`:

- `linux-api-headers`
- `linux-raspberrypi-headers`

The headers can then be found in: `/lib/modules/${VERSION}/build/include/linux/`

where `${VERSION}` can be found with `uname -r`.

F C based GPIO libraries installation guides

F.1 `libgpiod`

Go find the latest snapshot of the library that still supports[2] the 4.19 (32bit) kernel at the Linux kernel's libgpiod page and download it on the Pi.

Extract the archive: `tar -xzf libgpiod-1.4.2.tar.gz`

Note

The instructions coming up will install the library in the usual system paths for such things. If you want it elsewhere, a `--prefix=<target directory>` must be added to the `autogen.sh` options.

From the directory (`cd libgpiod-1.4.2/`) execute the following:

```
$ ./autogen.sh --enable-tools=yes --host=arm-linux-gnueabi
  ac_cv_func_malloc_0_nonnull=yes
$ make
$ sudo make install
```

F.2 `pigpio`

Clone the repository with git:

```
$ git clone https://github.com/joan2937/pigpio.git
```

Navigate to the clone repository's directory and compile/install the library:

```
$ cd pigpio
$ make
$ sudo make install
```

F.3 `wiringpi`

As long as it was installed during the "3.1.2 Updating Arch" section along with the other extra programs included there, it should already be available.

If not then just install it via the package manager:

```
$ sudo pacman -S wiringpi
```

[2] libgpiod v1.4.2 is the last version still supporting older 32bit kernels to date.