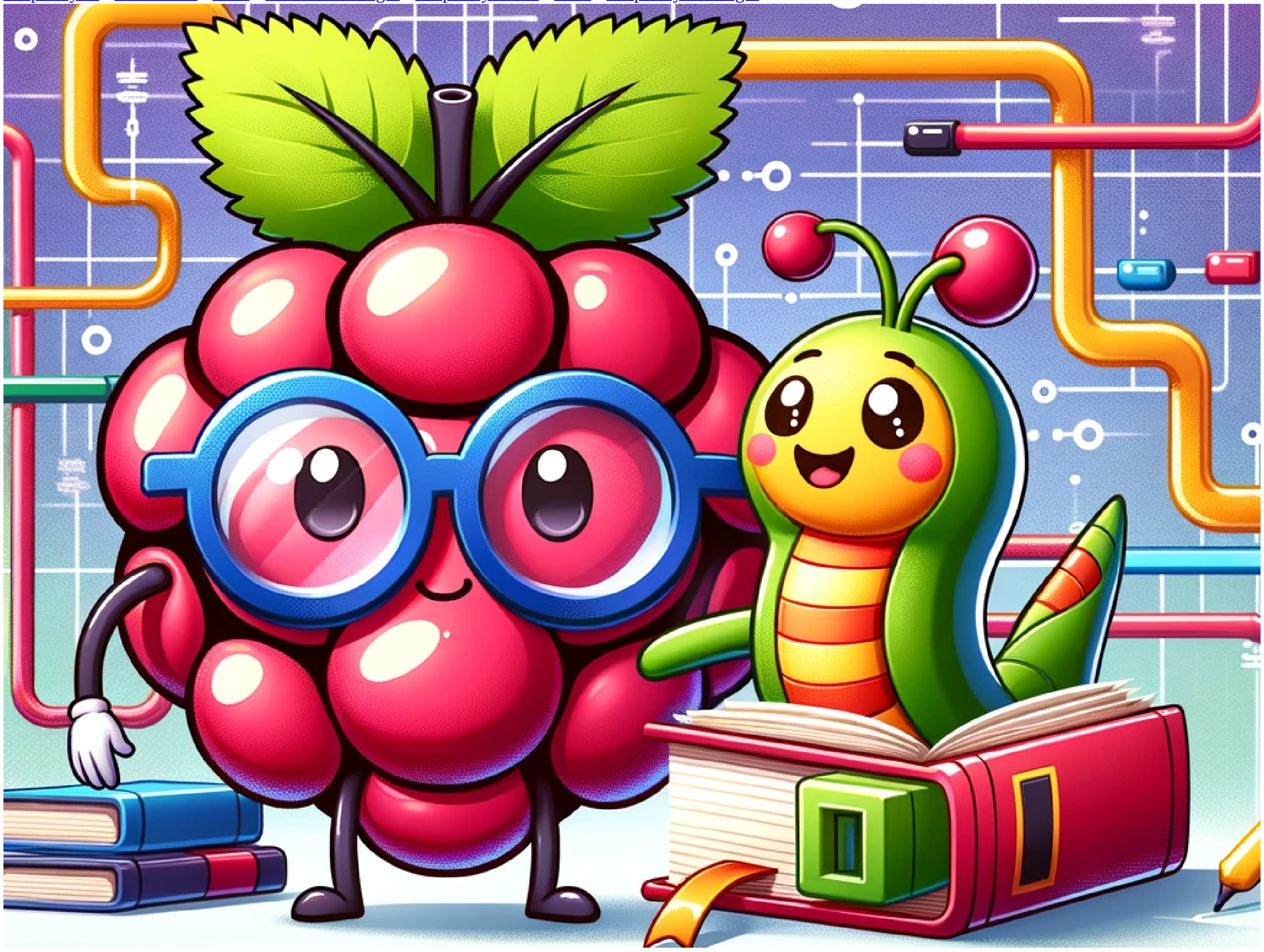
Headless WiFi setup on Raspberry Pi OS "Bookworm" without the Raspberry Pi Imager

January 6, 2024
Raspberry Pi • Bookworm • WiFi • Network Manager • Raspberry Pi OS • Bash • Raspberry Pi Imager



Setting up a Raspberry Pi headless without the Raspberry Pi Imager used to be a fairly simple process for the average Linux user, to the point where a how-to and a few searches on the Raspberry Pi forums would sort the process out. After flashing the image with `dd`, creating `ssh` in the boot partition and populating `wpa_supplicant.conf` was normally enough to get started.

However with the recently released Raspberry Pi OS 12 "Bookworm" this second step doesn't work anymore and the only recommendation that users receive is to "just use the Raspberry Pi Imager" (like here).

But what does the Imager really do to configure the OS? Is it really that complex that it requires downloading a dedicated installer?

In this post I'm going to find out first how to get the OS connect to the WiFi without Imager, and then I'm going to dig a bit deeper to find out why such advice is given and how the Imager performs this configuration step.

Network Manager Link to heading

In the announcement of the new OS release, one of the highlights is the move to NetworkManager as the default mechanism to deal with networking. While this move undoubtely brings many advantages, it is the reason why the classic technique of dropping a `wpa_supplicant.conf` file under `/etc/wpa_supplicant/` no longer works.

The good news is that also NetworkManager can be manually configured with a text file. The file needs to be called `SSID.nmconnection` (replace `SSID` with your network's SSID) and placed under `/etc/NetworkManager/system-connections/` in the Pi's `rootfs` partition.

```
[connection]

id=SSID
uuid= # random UUID in the format 11111111-1111-1111-1111-111111111111
type=wifi
autoconnect=true

[wifi]
mode=infrastructure
ssid=SSID

[wifi-security]
auth-alg=open
key-mgmt=wpa-psk
psk=PASSWORD

[ipv4]
method=auto

[ipv6]
method=auto
```

(replace `SSID` and `PASSWORD` with your wifi network's SSID and password). Here you can find the full syntax for this file.

You'll need also to configure its access rights as:

```
sudo chmod -R 600 <path-to-rootfs>/etc/NetworkManager/system-connections/SSID.nmconnection
sudo chown -R root:root <path-to-rootfs>/etc/NetworkManager/system-connections/SSID.nmconnection
```

Once this is done, let's not forget to create an empty `ssh` file in the `bootfs` partition to enable the SSH server:

```
touch <path-to-bootfs>/ssh
```

and, as it was already the case in Bullseye to configure the default user with `userconfig.txt`:

```
echo 'mypassword' | openssl passwd -6 -stdin | awk '{print "myuser:"$1}' > <path-to-bootfs>/userconfig.txt
```

So far it doesn't seem too complicated. However, interestingly, this is **not** what the Raspberry Pi Imager does, because if you use it to flash the image and check the result, these files are nowhere to be found. Is there a better way to go about this?

Raspberry Pi Imager Link to heading

To find out what the Imager does, my first idea was to have a peek at its source code. Being a Qt application the source might be quite intimidating, but with a some searching it's possible to locate this interesting snippet:

```
void ImageWriter::setImageCustomization(const QByteArray &config, const QByteArray &cmdline, const QByteArray &firstrun, const QByteArray &cl
{
    _config = config;
    _cmdline = cmdline;
    _firstrun = firstrun;
    _cloudinit = cloudinit;
    _cloudinitNetwork = cloudinitNetwork;

    qDebug() << "Custom config.txt entries:" << config;
    qDebug() << "Custom cmdline.txt entries:" << cmdline;
    qDebug() << "Custom firstuse.sh:" << firstrun;
    qDebug() << "Cloudinit:" << cloudinit;
}
```

I'm no C++ expert, but this function tells me a few things:

1. The Imager writes the configuration in these files: `config.txt`, `cmdline.txt`, `firstuse.sh` (we'll soon figure out this is a typo: the file is actually called `firstrun.sh`).
2. It also prepares a "Cloudinit" configuration file, but it's unclear if it writes it and where
3. The content of these files is printed to the console as debug output.

So let's enable the debug logs and see what they produce:

```
rpi-imager --debug
```

The console stays quiet until I configure the user, password, WiFi and so on in the Imager, at which point it starts printing all the expected configuration files to the console.

▶ *Click here to see the full output*

Among these the most interesting file is `firstrun.sh`, which we can quickly locate in the `bootfs` partition. Here is its content:

```
#!/bin/bash

set +e

CURRENT_HOSTNAME=`cat /etc/hostname | tr -d " \    \
\\r"`
if [ -f /usr/lib/raspberrypi-sys-mods/imager_custom ]; then
   /usr/lib/raspberrypi-sys-mods/imager_custom set_hostname raspberrypi
else
   echo raspberrypi >/etc/hostname
   sed -i "s/127.0.1.1.*$CURRENT_HOSTNAME/127.0.1.1\    raspberrypi/g" /etc/hosts
fi
FIRSTUSER=`getent passwd 1000 | cut -d: -f1`
FIRSTUSERHOME=`getent passwd 1000 | cut -d: -f6`
if [ -f /usr/lib/raspberrypi-sys-mods/imager_custom ]; then
```

```
   /usr/lib/raspberrypi-sys-mods/imager_custom enable_ssh
else
   systemctl enable ssh
fi
if [ -f /usr/lib/userconf-pi/userconf ]; then
   /usr/lib/userconf-pi/userconf 'myuser' '<hash-of-the-user-password>'
else
   echo "$FIRSTUSER:"'<hash-of-the-user-password>' | chpasswd -e
   if [ "$FIRSTUSER" != "myuser" ]; then
      usermod -l "myuser" "$FIRSTUSER"
      usermod -m -d "/home/myuser" "myuser"
      groupmod -n "myuser" "$FIRSTUSER"
      if grep -q "^autologin-user=" /etc/lightdm/lightdm.conf ; then
         sed /etc/lightdm/lightdm.conf -i -e "s/^autologin-user=.*/autologin-user=myuser/"
      fi
      if [ -f /etc/systemd/system/getty@tty1.service.d/autologin.conf ]; then
         sed /etc/systemd/system/getty@tty1.service.d/autologin.conf -i -e "s/$FIRSTUSER/myuser/"
      fi
      if [ -f /etc/sudoers.d/010_pi-nopasswd ]; then
         sed -i "s/^$FIRSTUSER /myuser /" /etc/sudoers.d/010_pi-nopasswd
      fi
   fi
fi
if [ -f /usr/lib/raspberrypi-sys-mods/imager_custom ]; then
   /usr/lib/raspberrypi-sys-mods/imager_custom set_wlan 'MY-SSID' 'MY-PASSWORD' 'PT'
else
cat >/etc/wpa_supplicant/wpa_supplicant.conf <<'WPAEOF'
country=PT
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
ap_scan=1

update_config=1
network={
    ssid="MY-SSID"
    psk=MY-PASSWORD
}

WPAEOF
   chmod 600 /etc/wpa_supplicant/wpa_supplicant.conf
   rfkill unblock wifi
   for filename in /var/lib/systemd/rfkill/*:wlan ; do
       echo 0 > $filename
   done
fi
if [ -f /usr/lib/raspberrypi-sys-mods/imager_custom ]; then
   /usr/lib/raspberrypi-sys-mods/imager_custom set_keymap 'us'
   /usr/lib/raspberrypi-sys-mods/imager_custom set_timezone 'Europe/Lisbon'
else
   rm -f /etc/localtime
   echo "Europe/Lisbon" >/etc/timezone
   dpkg-reconfigure -f noninteractive tzdata
cat >/etc/default/keyboard <<'KBEOF'
XKBMODEL="pc105"
XKBLAYOUT="us"
XKBVARIANT=""
XKBOPTIONS=""

KBEOF
   dpkg-reconfigure -f noninteractive keyboard-configuration
fi
rm -f /boot/firstrun.sh
sed -i 's| systemd.run.*||g' /boot/cmdline.txt
exit 0
```

▶ *Side note: how does the OS know that it should run this file on its first boot?*

That's a lot of Bash in one go, but upon inspection one can spot a recurring pattern. For example, when setting the hostname, it does this:

```
if [ -f /usr/lib/raspberrypi-sys-mods/imager_custom ]; then
   /usr/lib/raspberrypi-sys-mods/imager_custom set_hostname raspberrypi
else
   echo raspberrypi >/etc/hostname
   sed -i "s/127.0.1.1.*$CURRENT_HOSTNAME/127.0.1.1\    raspberrypi/g" /etc/hosts
fi
```

The script clearly messages that there is a "preferred" way to set the hostname: to use `/usr/lib/raspberrypi-sys-mods/imager_custom set_hostname [NAME]`. Only if this executable is not available, then it falls back to the "traditional" way of setting the hostname by editing `/etc/hosts`.

The same patterns repeat a few times to perform the following operations:

- set the hostname (`/usr/lib/raspberrypi-sys-mods/imager_custom set_hostname [NAME]`)
- enable ssh (`/usr/lib/raspberrypi-sys-mods/imager_custom enable_ssh`)
- configure the user (`/usr/lib/userconf-pi/userconf [USERNAME] [HASHED-PASSWORD]`)
- configure the WiFi (`/usr/lib/raspberrypi-sys-mods/imager_custom set_wlan [MY-SSID [MY-PASSWORD] [2-LETTER-COUNTRY-CODE]`)
- set the keyboard layout (`/usr/lib/raspberrypi-sys-mods/imager_custom set_keymap [CODE]`)
- set the timezone (`/usr/lib/raspberrypi-sys-mods/imager_custom set_timezone [TIMEZONE-NAME]`)

It seems like using `raspberrypi-sys-mods` to configure the OS at the first boot is the way to go in this RPi OS version, and it might be true in future versions as well. There are hints that the Raspberry PI OS team is going to move to `cloud-init` in the near future, but for now this seems to be the way that the initial setup is done.

raspberrypi-sys-mods Link to heading

So let's check out what `raspberrypi-sys-mods` do! The source code can be found here: raspberrypi-sys-mods.

Given that we're interested in the WiFi configuration, let's head straight to the `imager_custom` script (here), where we discover that it's a Bash script which does this:

```
CONNFILE=/etc/NetworkManager/system-connections/preconfigured.nmconnection
  UUID=$(uuid -v4)
  cat <<- EOF >${CONNFILE}
    [connection]
    id=preconfigured
    uuid=${UUID}
    type=wifi
    [wifi]
    mode=infrastructure
    ssid=${SSID}
    hidden=${HIDDEN}
    [ipv4]
    method=auto
    [ipv6]
    addr-gen-mode=default
    method=auto
    [proxy]
    EOF

  if [ ! -z "${PASS}" ]; then
    cat <<- EOF >>${CONNFILE}
    [wifi-security]
    key-mgmt=wpa-psk
    psk=${PASS}
    EOF
  fi

  # NetworkManager will ignore nmconnection files with incorrect permissions,
  # to prevent Wi-Fi credentials accidentally being world-readable.
  chmod 600 ${CONNFILE}
```

So after all this searching, we're back to square one. This utility is doing exactly what we've done at the start: it writes a NetworkManager configuration file called `preconfigured.nmconnection` and it fills it in with the information that we've provided to the Imager, then changes the permissions to make sure NetworkManager can use it.

Conclusion Link to heading

It would be great if the Raspberry Pi OS team would expand their documentation to include this information, so that users aren't left wondering what makes the RPi Imager so special and whether their manual setup is the right way to go or rather a hack that is likely to break. For now it seems like there is one solid good approach to this problem, and we are going to see what is going to change in the next version of the Raspberry Pi OS.

On this note you should remember that doing a manual configuration of NetworkManager, using the Imager, or using `raspberrypi-sys-mods` may be nearly identical right now, but when choosing which approach to use for your project you should also keep in mind the maintenance burden that this decision brings.

Doing a manual configuration is easier on many levels, but only if you don't intend to support other versions of RPi OS. If you do, or if you expect to migrate when a new version comes out, you should consider doing something similar to what the Imager does: use a `firstrun.sh` file that tries to use `raspberrypi-sys-mods` and falls back to a manual configuration only if that executable is missing. That is likely to make migrations easier if the Raspberry Pi OS team should choose once again to modify the way that headless setups work.