PIFS Guide

Christopher Lang

March 30, 2023

Contents

Chapter 1

Preface

1.1 Introduction

'PIFS' stands for '(Raspberry) Pi (Operating System) From Scrach' This guide describes how to build a Linux based OS for the 'Raspberry Pi 2 Zero W'. An x86-64 Linux host will be used to cross compile software for the ARM Raspberry Pi.

The OS will consist of the kernel, a shell, core command line utilities and a native toolchain.

1.2 Prerequisites

You must be familiar with basic Linux system administration. You must be confident using a unix shell and compiling C software. You must be able to use `vi` to edit text files. Knowledge of the C programming language is recommended but not required.

The following hardware is required:

- Raspberry Pi 2 Zero W
- x86-64 computer with a working Linux OS installed
- Power Supply for Raspberry Pi
- Monitor that can connect to the Raspberry Pi's HDMI mini port
- US keyboard that can connect to the Raspberry Pi's USB micro port
- Micro SD card 8GB or larger

1.3 Terminology

'system', 'OS' and 'operating system' are used interchangeably. The x86-64 system that this guide be compiling from is referred to as the 'host'. The Raspberry Pi and the OS to be built for the Raspberry Pi is referred to as the 'target'.

1.4 Typography

In this guide, bash commands are written in the following form:

```
# echo "run this command as root user"
$ echo "run this command as pifs user"
```

The first line is prepended with a '`#`' (hashtag), meaning that the following command, `echo "run this command as root user"`, should be run as root. The second line, prepended with a '`$`' (dollar sign), should be run as the `pifs` user - a user created later in the guide.

If your host has `sudo`[1] installed, you can use it to conveniently run commands as root.

[1] *Sudo.* '`https://www.sudo.ws/`'.

Chapter 2

Preparing for the Build

2.1 Creating the PIFS User

2.1.1 Why a New User?

This section will create a new user on the host. Users already in your host system may set environment variables that affect the build process. Creating a new user helps to ensure a clean environment[1]. By modifying the new user's `.bash_profile` and `.bashrc` files, the shell can be prepared for cross-compilation.

2.1.2 Add the New User

```
# groupadd pifs
# useradd -g pifs -m -k /dev/null pifs
```

Relevant options for `useradd`[2]:

`-g, --gid GROUP`
: The group name or number of the user's initial login group.

`-m, --create-home`
: Create the user's home directory if it does not exist.

`-k, --skel SKEL_DIR`
: The skeleton directory, which contains files and directories to be copied in the user's home directory, when the home directory is created by useradd.

Set a password for the new user using `passwd`[3].

```
# passwd pifs
```

[1] *Linux From Scratch 11.1.* '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.
[2] *useradd(8) Linux Manual Page.* '`https://linux.die.net/man/8/useradd`'.
[3] *passwd(1) Linux Manual Page.* '`https://linux.die.net/man/1/passwd`'.

2.1.3 Add the New User to Sudoers

This subsection may be skipped if you do not wish to use `sudo` to make running commands as root more convenient.

To give the `pifs` user privilege to use the `sudo` command, we must add the user to the sudoers file. Editing the sudoers file with `visudo` will ensure the modifications made do not lock the main user out of the system[4].

The following command will open the editor specified by the `EDITOR` environment variable.

```
# visudo
```

In the editor, add the line:

```
pifs ALL=(ALL) ALL
```

This means: allow the `pifs` user to, on all hosts, as all users, run all commands[5].

2.1.4 Switching to the New User and Creating `.bash_profile`

To switch to the newly created `pifs` user:

```
$ su - pifs
```

Relevant options for `su`[6]:

`-, -l, --login`
: Start the shell as a login shell with an environment similar to a real login.

A login shell must be used as this will cause `.bash_profile` to be executed.

Create `/home/pifs/.bash_profile` with the content:

```
exec env -i HOME=${HOME} TERM=${TERM} PS1='\u:\w\$ ' /bin/bash
```

This creates a new (clean) environment when we enter a login shell. `exec` replaces the shell with the specified command. `env` creates a new shell environment.

Relevant options for `env`[7]:

`-i, --ignore-environment`
: Start with an empty environment.

After entering our new environment, the `.bashrc` file will be run.

[4] *visudo(8) Linux Manual Page.* '`https://linux.die.net/man/8/visudo`'.
[5] *sudoers(5) Linux Manual Page.* '`https://linux.die.net/man/5/sudoers`'.
[6] *su(1) Linux Manual Page.* '`https://linux.die.net/man/1/su`'.
[7] *env(1) Linux Manual Page.* '`https://linux.die.net/man/1/env`'.

2.1.5 Creating `.bashrc`

Create `/home/pifs/.bashrc` with the content:

.bashrc

```
set +h
umask 022

TARGET_TRIPLET=arm-buildroot-linux-gnueabihf
CROSS_TOOLS=/home/pifs/cross-tools
PIFS_ROOT=/mnt/pi-root
PIFS_BOOT=/mnt/pi-boot
STAGING_ROOT=/home/pifs/staging-root

LC_ALL=POSIX # localization
PATH=${CROSS_TOOLS}/bin:/bin:/sbin:/usr/bin:/usr/sbin

CC="${TARGET_TRIPLET}-gcc"
CXX="${TARGET_TRIPLET}-g++"
AR="${TARGET_TRIPLET}-ar"
AS="${TARGET_TRIPLET}-as"
RANLIB="${TARGET_TRIPLET}-ranlib"
LD="${TARGET_TRIPLET}-ld"
STRIP="${TARGET_TRIPLET}-strip"

unset CFLAGS CXXFLAGS
export PATH LC_ALL CC CXX AR AS RANLIB LD STRIP \
    CROSS_TOOLS PIFS_ROOT PIFS_BOOT TARGET_TRIPLET \
    STAGING_ROOT
```

Explanation of `.bashrc` commands:

`set +h`
: Do not remember the location of binaries found in `PATH`.

`umask 022`
: Files this user creates can be read by every user.

`TARGET_TRIPLET=arm-buildroot-linux-gnueabihf`
: The target triplet that prepends the cross-toolchain binaries. A target triplet 'describes the platform on which code runs'[8]. They are in the format: `machine-vendor-os`. In the case of `arm-buildroot-linux-gnueabihf` the `os` part is two fields: `linux-gnueabihf`.

`CROSS_TOOLS=/home/lfs/cross-tools`
: Where the cross-toolchain will be located.

`PIFS_ROOT=/mnt/pi-root`
: Where the target's root partition will be mounted.

`PIFS_BOOT=/mnt/pi-boot`
: Where the target's boot partition will be mounted.

[8] *OS Dev - Target Triplet.* '`https://wiki.osdev.org/Target_Triplet`'.

`STAGING_ROOT=/home/lfs/staging-root`

An intermediate location where packages will be installed before they are copied to the root partition. This is useful for selecting what parts of a package's installation will be included in the target OS.

```
CC="${TARGET_TRIPLET}-gcc"
...
STRIP="${TARGET_TRIPLET}-strip"
```

Cross-toolchain binaries which may be read by build scripts.

`unset CFLAGS CXXFLAGS`

Unset environment variables that may affect build process.

Change into the new environment by executing the `.bash_profile` script in the current shell.

```
$ source ~/.bash_profile
```

Create folders in the new home directory which will be used in future chapters.

```
$ mkdir ~/{cross-tools,staging-root,src,tarballs}
```

2.2 Installing a Cross-Toolchain

The cross-toolchain will be generated by '`https://toolchains.bootlin.com`'[9]. Open the link in a browser, select architecture: `armv7-eabihf`, libc: `glibc` and copy the stable download link. At time of writing, the package versions used are:

- binutils 2.36.1
- gcc 10.3.0
- gdb 10.2
- glibc 2.34-9-g9acab0b...
- linux-headers 4.9.291

Use the link to download the archive and extract it. Move its contents into the `cross-tools` directory. In the following commands, replace `<LINK>` with the link to the archive download you just copied.

```
$ cd ~
$ wget <LINK>
$ tar -xf armv7-eabihf--glibc--stable-2021.11-1.tar.bz2
$ rm armv7-eabihf--glibc--stable-2021.11-1.tar.bz2
$ mv armv7-eabihf--glibc--stable-2021.11-1/* cross-tools/
$ rm -r armv7-eabihf--glibc--stable-2021.11-1/
```

[9] *toolchains.bootlin.com.* '`https://toolchains.bootlin.com/`'.

The cross-toolchain binaries are now located in `~/cross-tools/bin`. This directory was added to your `PATH` in the `.bashrc` created earlier so the shell should be able to locate the new binaries. Verify this by running the compiler, `$CC`, specified in `.bashrc`.

```
$ ${CC} --version
```

2.3 Preparing the Storage Medium

2.3.1 Partitioning

This guide partitions the SD card using `cfdisk` but alternative partition tools are available. This table shows common partition tools for Linux.

Package	CLI	TUI	GUI
fdisk[10]	fdisk	cfdisk	partitionmanager
parted[11]	parted	-	gparted, gnome-disk-utility

Using your preferred partition tool, with a DOS partition table, create a 256MiB FAT32 boot partition followed by a Linux root partition that fills the rest of the SD card. The boot partition will contain Raspberry Pi firmware including the kernel but not kernel modules. It is a FAT32 filesystem because the bios may not be able to read more complex types.

If using `cfdisk`, the partition table should appear similar to the following (though the size of the second partition may be different to fill the capacity of the storage device):

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/sdX1	*	2048	526335	524288	256M	b	W95 FAT32
/dev/sdX2		526336	30930943	30404608	14.5G	83	Linux

Determine the device files name of your boot and root partition. This can be done using the `lsblk`[12] command which will list block devices. `/dev/sdb1` and `/dev/sdb2` are common device file names for a secondary disk. In the following examples, this guide will use `/dev/sdX1` to refer to the boot partition and `/dev/sdX2` to refer to the root partition. But you must replace these with your device file names.

2.3.2 Create Filesystems

To use the partitions, they must be formatted with filesystems. This guide uses FAT32 for the boot partition and ext4 for the root partition.

```
# mkfs -t vfat /dev/sdX1
# mkfs -t ext4 /dev/sdX2
```

[12] *lsblk(8) Linux Manual Page.* '`https://linux.die.net/man/8/lsblk`'.

Relevant options for `mkfs`[13]:

`-t, --type type`
: Specify the type of filesystem to be built.

2.3.3 Mounting

Root and boot filesystems are now ready to be mounted. They must be mounted into the paths specified in `.bashrc` created earlier: `/mnt/pi-boot` and `/mnt/pi-root`. First, create these directories if they do not already exist.

```
# mkdir -p /mnt/pi-{boot,root}
```

Mount the two partitions.

```
# mount /dev/sdX1 /mnt/pi-boot
# mount /dev/sdX2 /mnt/pi-root
```

2.3.4 Creating Target Directory Structure

Create essential directories in our target system's filesystem. This directory structure is loosely based of the Linux Filesystem Hierarchy Standard[14].

```
# mkdir -p ${PIFS_ROOT}{/dev,/proc,/sys,/lib,/bin,/sbin,/usr/lib,\
   /usr/bin,/usr/sbin,/usr/include,/usr/libexec}
```

Explanation of unobvious directories:

`/dev`
: Virtual filesystem populated by the kernel for device files.

`/proc`
: Virtual filesystem populated by the kernel where information about running processes can be found.[15]

`/sys`
: Virtual filesystem populated by the kernel where information about kernel components can be found.[16]

`/libexec`
: Where binaries that do not belong in `PATH` are installed.[17] GCC uses this directory.

`/sbin` and `/usr/sbin`
: 'special' binaries. For example, busybox's `poweroff`. These directories are in `PATH`.

[13] *mkfs(8) Linux Manual Page.* '`https://linux.die.net/man/8/mkfs`'.
[14] *Filesystem Hierarchy Standard Version 3.0.* '`https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html`'. 2015.
[15] *proc(5) Linux Manual Page.* '`https://linux.die.net/man/5/proc`'.
[16] *sysfs(5) Linux Manual Page.* '`https://man7.org/linux/man-pages/man5/sysfs.5.html`'.
[17] ncoghlan. *Stack Exchange — What is the purpose of /usr/libexec?* '`https://unix.stackexchange.com/a/386015`'.

2.4 Installing Firmware

In the home directory of the `pifs` user. Run the following command to download the latest Raspberry Pi firmware[18]. This may take some time.

```
$ git clone https://github.com/raspberrypi/firmware
```

Copy the contents of the firmware's boot folder into the boot partition.

```
# cp -dr ~/firmware/boot/* ${PIFS_BOOT}
```

Relevant options for `cp`[19]:

`-d`
: Preserve links during copy. Same as `--no-dereference --preserve=links`.

Determine the Part-UUID of the root partition of the new system (`/dev/sdX2`). The following command lists devices Part-UUIDs.

```
$ ls -l /dev/disk/by-partuuid/
```

An example Part-UUID is `a82b43f76-02`.

Create a file owned by root: `$PIFS_BOOT/cmdline.txt`. This file is read by Raspberry Pi firmware and configures the command-line parameters passed to the kernel on boot. Set the content of the file to the following, replacing `<PUUID>` with the Part-UUID of the root partition:

cmdline.txt

```
root=PARTUUID=<PUUID> rootfstype=ext4 rw rootwait init=/init
```

Explanation of unobvious `cmdline.txt` options:[20]

`root=PARTUUID=...`
: The Part-UUID of the partition to use as root.

`rw`
: Mount root device read-write on boot

`rootwait`
: Wait (indefinitely) for root device to show up.

`init=/init`
: Run specified binary instead of `/sbin/init` as init process.

Copy kernel modules into the root partition.

```
sudo cp -dr ~/firmware/modules ${PIFS_ROOT}/lib
```

[18] *Raspberry Pi Firmware*. '`https://github.com/raspberrypi/firmware`'.
[19] *cp(1) Linux Manual Page*. '`https://linux.die.net/man/1/cp`'.
[20] *The Linux Kernel Documentation — Command-line Parameters*. '`https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html`'.

2.5 Downloading Core Software

Download the required source code tarballs.

```
$ cd ~/tarballs
$ wget https://www.busybox.net/downloads/busybox-1.35.0.tar.bz2
$ wget http://zlib.net/zlib-1.2.12.tar.gz
$ wget https://ftp.gnu.org/gnu/binutils/binutils-2.39.tar.gz
$ wget https://ftp.gnu.org/gnu/gcc/gcc-11.3.0/gcc-11.3.0.tar.gz
$ wget https://ftp.gnu.org/gnu/make/make-4.3.tar.gz
```

If a link no longer works, find an alternative download link using a search engine.

Chapter 3

Building Core Software

3.1 Glibc

Glibc is GNU's standard C library. A compilers output binaries must link with the same standard C library that was specified during the compilation of the compiler[1]. So the cross-compiled software must link against the glibc from the `$CROSS_TOOLS` directory. Copy the important glibc files into the new system's root.

```
# cd ${CROSS_TOOLS}/arm-buildroot-linux-gnueabihf/sysroot/
# cp -dr lib/* ${PIFS_ROOT}/lib
# cp -dr sbin/* ${PIFS_ROOT}/sbin
# cp -dr usr/lib/* ${PIFS_ROOT}/usr/lib
# cp -dr usr/include/* ${PIFS_ROOT}/usr/include
# cp -dr usr/bin/* ${PIFS_ROOT}/usr/bin
# cp -dr usr/sbin/* ${PIFS_ROOT}/usr/sbin
```

3.2 Busybox

Busybox 'combines tiny versions of many common UNIX utilities into a single small executable'[2]. Extract and configure the busybox source.

```
$ cd ~/src
$ tar -xf ~/tarballs/busybox-1.35.0.tar.bz2
$ cd busybox-1.35.0
$ make defconfig
```

`make defconfig` creates a default configuration file. The configuration can be modified using `make menuconfig` but this is not required for the guide.

[1] *Linux From Scratch 11.1.* '`https://www.linuxfromscratch.org/lfs/view/11.1/`'. 2022.
[2] *busybox 1.35.0.* '`https://www.busybox.net/`'. 2021.

Compile and install busybox into `$STAGING_ROOT`. `CROSS_COMPILE` sets the prefix for toolchain binaries to use.

```
$ make CROSS_COMPILE=${TARGET_TRIPLET}-
$ make CROSS_COMPILE=${TARGET_TRIPLET}- CONFIG_PREFIX=${STAGING_ROOT} \
    install
```

Inspect the files in `$STAGING_ROOT` to see what busybox installs. Copy the required files from the staging root into the new system's root.

```
# cp -dr ${STAGING_ROOT}/bin/* ${PIFS_ROOT}/bin
# cp -dr ${STAGING_ROOT}/sbin/* ${PIFS_ROOT}/sbin
# cp -dr ${STAGING_ROOT}/usr/bin/* ${PIFS_ROOT}/usr/bin
# cp -dr ${STAGING_ROOT}/usr/sbin/* ${PIFS_ROOT}/usr/sbin
```

Busybox uses symbolic links to contain multiple command line programs within a single binary[3]. Therefore the `-d` option is required.

Clean the staging root so that subsequent software can be installed into a clean directory.

```
$ rm -r ${STAGING_ROOT}/*
```

Create a file `${PIFS_ROOT}/.profile` with content:

.profile

```
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

This file is run when entering busybox's shell in login mode. The commands will set the `PATH` environment variable for the new system.

3.3 Init System

This guide will provide the source code for a simple init binary. Create a directory for the source.

```
$ mkdir ~/init
$ cd ~/init
```

Create a file `/home/pifs/init/init.c` with the following content.

init.c

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/mount.h>
#include <unistd.h>
```

[3] *busybox 1.35.0*. '`https://www.busybox.net/`'. 2021.

```
void tryProc() {
  if (mount("none", "/proc", "proc", 0, NULL)) {
    printf(
      "INIT: failed to mount proc (%d) %s\n",
      errno,
      strerror(errno));
    return;
  }
  printf("INIT: mounted proc\n");
}

void trySys() {
  if (mount("none", "/sys", "sysfs", 0, NULL)) {
    printf(
      "INIT: failed to mount sys (%d) %s\n",
      errno,
      strerror(errno));
    return;
  }
  printf("INIT: mounted sys\n");
}

void tryDev() {
  if (mount("none", "/dev", "devtmpfs", MS_DIRSYNC, NULL)) {
    printf(
      "INIT: failed to mount dev (%d) %s\n",
      errno,
      strerror(errno));
    return;
  }
  printf("INIT: mounted dev\n");
}

int main() {
  printf("INIT: start\n");

  tryProc();
  trySys();
  tryDev();

  printf("INIT: shell\n");

  char *args[] = {"/bin/sh", NULL};
  execv(args[0], args);

  for (;;);
  return 0;
}
```

The init program is the first process the kernel starts in user space. It is the

root of all other user space processes. This init program mounts the `dev`, `sys` and `proc` virtual filesystems then executes `/bin/sh`.

Cross compile the init program.

```
$ ${CC} init.c -o init
```

Copy the `init` binary into the root directory of the new system. The location of the `init` binary was specified earlier in the `cmdline.txt` file.

```
# cp init ${PIFS_ROOT}
```

3.4 Zlib

Zlib is a compression library required by bintuils and GCC.

Extract the archive.

```
$ cd ~/src
$ tar -xf ~/tarballs/zlib-1.2.12.tar.gz
$ cd zlib-1.2.12/
```

`zlib` uses environment variables to find compilation tools. These were set in `.bashrc`. Compile `zlib` and install into the staging root.

```
$ ./configure --prefix=${STAGING_ROOT}
$ make
$ make install
```

Copy required files into the new system's root filesystem.

```
# cp -dr ${STAGING_ROOT}/include/* ${PIFS_ROOT}/usr/include/
# cp -dr ${STAGING_ROOT}/lib/* ${PIFS_ROOT}/usr/lib/
```

Clean the staging root.

```
$ rm -r ${STAGING_ROOT}/*
```

3.5 Binutils

Binutils is part of the toolchain. It is a collection of tools for interacting with binary files including a linker (`ld`) and an assembler (`as`).

Extract the archive.

```
$ cd ~/src
$ tar -xf ~/tarballs/binutils-2.39.tar.gz
$ cd binutils-2.39/
```

Configure and compile `bintuils`, disabling unnecessary features.

```
$ ./configure --prefix=${STAGING_ROOT} --host=${TARGET_TRIPLET} \
    --target=${TARGET_TRIPLET} --disable-nls --disable-multilib
$ make
$ make install
```

Copy the required files into the new system's root filesystem.

```
# cp -dr ${STAGING_ROOT}/bin/* ${PIFS_ROOT}/usr/bin/
# cp -dr ${STAGING_ROOT}/include/* ${PIFS_ROOT}/usr/include/
# cp -dr ${STAGING_ROOT}/lib/* ${PIFS_ROOT}/usr/lib/
```

Clean the staging root.

```
$ rm -r ${STAGING_ROOT}/*
```

3.6 GCC

GCC is the GNU C compiler. It will be used to compile software on the new system.

Extract the archive.

```
$ cd ~/src/
$ tar -xf ~/tarballs/gcc-11.3.0.tar.gz
$ cd gcc-11.3.0/
```

To compile GCC, the cross-compiler needs access to some libraries and header files that currently exist in the new system's root filesystem. Copy these files into the cross-compiler's system root so that they can be found.

```
$ cp -dr ${PIFS_ROOT}/usr/include/* \
    ${CROSS_TOOLS}/arm-buildroot-linux-gnueabihf/sysroot/usr/include
$ cp ${PIFS_ROOT}/usr/lib/libz.a \
    ${CROSS_TOOLS}/arm-buildroot-linux-gnueabihf/sysroot/usr/lib
```

Download GMP, MPFT and MPC automatically. These are floating point arithmetic libraries that GCC depends on.

```
./contrib/download_prerequisites
```

[4]

Configure and compile GCC.

```
$ ./configure --prefix=${STAGING_ROOT} --host=${TARGET_TRIPLET} \
    --target=${TARGET_TRIPLET} --disable-multilib --disable-nls \
    --with-system-zlib --enable-languages=c --with-arch=armv7 \
    --with-float=hard --with-fpu=vfp
$ make
$ make install
```

[4] Free Software Foundation. *GCC Installation Manual.* 'https://gcc.gnu.org/install/'.

[5]

Copy the required files into the new system's root filesystem.

```
# cp -dr ${STAGING_ROOT}/bin/* ${PIFS_ROOT}/usr/bin/
# cp -dr ${STAGING_ROOT}/lib/* ${PIFS_ROOT}/usr/lib/
# cp -dr ${STAGING_ROOT}/libexec/* ${PIFS_ROOT}/usr/libexec/
```

Clean the staging root.

```
$ rm -r ${STAGING_ROOT}/*
```

3.7 Make

GNU Make is a build tool that will assist in compiling software in the new system. Extract the archive.

```
$ cd ~/src/
$ tar -xf ~/tarballs/make-4.3.tar.gz
$ cd make-4.3/
```

Configure and install the package into the staging root.

```
$ ./configure --prefix=${STAGING_ROOT} --host=${TARGET_TRIPLET}
$ make
$ make install
```

Copy the required files into the new system's filesystem.

```
# cp -dr ${STAGING_ROOT}/include/* ${PIFS_ROOT}/usr/include/
# cp -dr ${STAGING_ROOT}/bin/* ${PIFS_ROOT}/usr/bin/
```

[5] Free Software Foundation. *GCC Installation Manual.* '`https://gcc.gnu.org/install/`'.

Chapter 4

Final Notes

4.1 How to Boot Into the New System

Unmount the new system's partitions.

```
# umount /dev/sdX1 /dev/sdX2
```

Remove the micro SD card from the host and insert it into the Raspberry Pi. Connect the Raspberry Pi to a keyboard, monitor and power supply. The system should boot into a busybox shell.

4.2 Installing Additional Software

Copy source code to the SD card and use the native GCC installation to compile additional C programs. `make` can be used to automate compilation if the package supports the build system. If you wish to install a large program, consider downloading a pre-built binary or cross-compiling using the tool-chain created in section 2.2. This is because the Raspberry Pi's limited hardware can take a long time to compile large packages.

4.3 Other Raspberry PI Hardware

The 'Raspberry Pi 2 Zero W' is the only device this guide has been tested on. However, it is likely to function on other Raspberry Pi devices. Other ARM processor with hard floating point support may be compatible.

The 'Raspberry Pi 2 Zero W' uses the BCM2837 chip. The same chip is used in the 'Raspberry Pi 3 Model B' and 'Raspberry Pi Compute Module 3' meaning that these devices are very likely to work with the guide. The 'Raspberry Pi 2 Model B' also uses a similar chip.[1]

[1] *Raspberry Pi Processors.* '`https://www.raspberrypi.com/documentation/computers/processors.html`'.

4.4 What Now?

By following this guide, you have created a very minimal system. This section will give some general advice on how you can use and add features to the OS.

Many applications of the Raspberry Pi require automatically running programs on boot. This can be configured either by modifying the `.profile` file created in the root directory or by updating the init program to fork-exec into the desired process.

Users that need many packages installed with complex dependencies may want a package management system. Because most Linux distributions come with it built in, there is little information online about installing a package manager. In addition, package formats often require the system to be configured in a particular way, which this OS is not. For these reasons, I warn against trying to install pre-packaged software.

4.5 Thanks

Thanks to the Linux From Scratch Project for inspiring and for being an invaluable resource in the creation of this guide. Thanks to my EPQ supervisors for useful advice and guidance.

Bibliography

[1] *busybox 1.35.0*. 'https://www.busybox.net/'. 2021.

[2] *cp(1) Linux Manual Page*. 'https://linux.die.net/man/1/cp'.

[3] *env(1) Linux Manual Page*. 'https://linux.die.net/man/1/env'.

[4] *fdisk(8) Linux Manual Page*. 'https://linux.die.net/man/8/fdisk'.

[5] *Filesystem Hierarchy Standard Version 3.0*. 'https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html'. 2015.

[6] Free Software Foundation. *GCC Installation Manual*. 'https://gcc.gnu.org/install/'.

[7] *Linux From Scratch 11.1*. 'https://www.linuxfromscratch.org/lfs/view/11.1/'. 2022.

[8] *lsblk(8) Linux Manual Page*. 'https://linux.die.net/man/8/lsblk'.

[9] *mkfs(8) Linux Manual Page*. 'https://linux.die.net/man/8/mkfs'.

[10] ncoghlan. *Stack Exchange — What is the purpose of /usr/libexec?* 'https://unix.stackexchange.com/a/386015'.

[11] *OS Dev - Target Triplet*. 'https://wiki.osdev.org/Target_Triplet'.

[12] *parted(8) Linux Manual Page*. 'https://linux.die.net/man/8/parted'.

[13] *passwd(1) Linux Manual Page*. 'https://linux.die.net/man/1/passwd'.

[14] *proc(5) Linux Manual Page*. 'https://linux.die.net/man/5/proc'.

[15] *Raspberry Pi Firmware*. 'https://github.com/raspberrypi/firmware'.

[16] *Raspberry Pi Processors*. 'https://www.raspberrypi.com/documentation/computers/processors.html'.

[17] *su(1) Linux Manual Page*. 'https://linux.die.net/man/1/su'.

[18] *Sudo*. 'https://www.sudo.ws/'.

[19] *sudoers(5) Linux Manual Page*. 'https://linux.die.net/man/5/sudoers'.

[20] *sysfs(5) Linux Manual Page*. 'https://man7.org/linux/man-pages/man5/sysfs.5.html'.

[21] *The Linux Kernel Documentation — Command-line Parameters.* 'https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html'.

[22] *toolchains.bootlin.com.* 'https://toolchains.bootlin.com/'.

[23] *useradd(8) Linux Manual Page.* 'https://linux.die.net/man/8/useradd'.

[24] *visudo(8) Linux Manual Page.* 'https://linux.die.net/man/8/visudo'.