"Simplify, then add lightness." - Colin Chapman

# 1. PREFACE

This handbook (and Zeppe-Lin itself) is inspired and based on the *CRUX Handbook*. The original version was written by Per Liden *mailto:per@fukt.bth.se* and maintained by CRUX Team. This implementation was reworked and adapted for Zeppe-Lin by Alexandr Savca *mailto:alexandr.savca89@gmail.com*.

# 2. INTRODUCTION

Welcome to the Zeppe-Lin Handbook! It's not an extensive guide on how to use and configure common Linux software. The purpose of the handbook is to explain how to install, configure, and maintain Zeppe-Lin systems, and to highlight the differences between common Linux distributions and Zeppe-Lin.

- The online version of this handbook is placed at: *https://zeppe-lin.github.io/handbook.7.html*.
- The pdf version of this handbook can be downloaded at: *https://zeppe-lin.github.io/handbook.7.pdf*.
- The local copy of this handbook can be installed on Zeppe-Lin system by the following command:

```
# pkgman install handbook
```

## 2.1. What is Zeppe-Lin?

Zeppe-Lin is a lightweight GNU/Linux distribution for the x86-64 architecture targeted at experienced users. It is forked from CRUX and the primary focus of this distribution is "keep it simple" too.

The KISS principle reflects in a simple *tar.gz*-based package system, BSD-style init scripts, and a relatively small collection of trimmed packages.

The secondary focus is the utilization of new GNU/Linux features and recent tools and libraries.

## 2.2. Why use Zeppe-Lin?

In short, Zeppe-Lin might suit you very well if you are:

- An experienced user who wants a clean and simple GNU/Linux distribution as a foundation of your installation.
- A person who prefers editing configuration files with an editor to using GUI.
- Someone who does not hesitate to download and compile programs from the source.
- Someone who wants a KISS GNU/Linux distribution as a foundation of your own distro.

# 3. LICENSE

## 3.1. Packages

Since Zeppe-Lin is a GNU/Linux distribution, it contains software written by a lot of different people. Each software package comes with its own license, chosen by its author(s). To find out how a particular package is licensed, have a look at its source code.

## 3.2. Build Scripts

All package build scripts in Zeppe-Lin are Copyright (C) 2000-2021 by Per Liden *mailto:per@fukt.bth.se* and CRUX team *http://crux.nu* and are released under the GPLv3+: GNU General Public License version 3 or later *https://gnu.org/licenses/gpl.html*.

## 3.3. NO WARRANTY

Zeppe-Lin is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. Use it at YOUR OWN RISK.

# 4. INSTALLATION

## 4.1. Supported Hardware

Packages on the official **rootfs** image are compiled with optimization for x86-64 (AMD Athlon 64, Intel Core, Intel Atom) or newer processors. Do not try to install it on an i686 (Pentium Pro, Celeron, Pentium-III) or lower processor, it simply will not work.

## 4.2. Boot a Live System

Since Zeppe-Lin is distributed in the form of a compiled tarball containing a root filesystem, you will first need to boot your computer using a Linux-based image, so called "Live CD/DVD/USB" operating system.

Boot your preferred Live system, open a terminal and get root privileges (e.g. `sudo su`).

## 4.3. Disk Partitions and Filesystems

### 4.3.1. UEFI and LVM-on-LUKS

This section describes how to set up Zeppe-Lin on a fully encrypted disk (apart from the bootloader partition). We will have an LVM container installed inside an encrypted partition. To encrypt the partition containing the LVM volume group, **dm-crypt** (which is managed by the *cryptsetup(8)* command) and its LUKS subsystem is used.

**Important**:

Make sure you have the following packages installed on your live system: **parted**, **dosfstools**, **cryptsetup**, and **lvm2**.

#### 4.3.1.1. Partition scheme

This is a quite simple partition scheme used in this section. There is a SCSI disk `/dev/sda`, but if you have an NVME disk (like `/dev/nvme0n1`) or another SCSI disk (like `/dev/sdb`), it's simple as run `sed 's/sda/sdb/g'`.

```
+-----------+------------+------------------+----------------+
| Partition | Filesystem | Size             | Description    |
+-----------+------------+------------------+----------------+
| /dev/sda1 | fat32      | 512MB            | boot partition |
| /dev/sda2 | luks       | rest of the disk | luks partition |
+-----------+------------+------------------+----------------+
```

**Important**:

On UEFI systems with a GPT-partitioned disk, there must be an EFI system partition (ESP). The suggested size is around 512 MiB.

#### 4.3.1.2. Create the partitions

Using *parted(8)* utility we can create all required partitions.

```
# parted /dev/sda
(parted) mklabel gpt
(parted) mkpart ESP fat32 1MiB 513MiB
(parted) set 1 boot on
(parted) name 1 efiboot
(parted) mkpart primary 513MiB 100%
(parted) name 2 luks
(parted) quit
```

Encrypt the second (luks) partition with LUKS and open the LUKS device (for example, as `crypt` mapped name):

```
# cryptsetup luksFormat /dev/sda2
```

```
# cryptsetup luksOpen /dev/sda2 crypt
```

4.3.1.3. Create LVM inside LUKS device

Create a physical volume and a new volume group (named `zpln` for example):

```
# pvcreate /dev/mapper/crypt
# vgcreate zpln /dev/mapper/crypt
```

Before creating the logical volumes, this is a quite simple logical volumes scheme used in this section:

```
+-------------+------------+------------------+-------------+
| Volume name | Filesystem | Size             | Description |
+-------------+------------+------------------+-------------+
| swap        | swap       | 2 * RAM          | swap area   |
| root        | ext4       | rest of the disk | root fs     |
+-------------+------------+------------------+-------------+
```

To see the actual amount of RAM, type the following command:

```
# free -m
```

For example, we have 4GB RAM. The swap will have 8GB. Let's create the aforementioned two logical volumes:

```
# lvcreate -L 8G -n swap zpln
# lvcreate -l 100%FREE -n root zpln
```

4.3.1.4. Create the filesystems

Create a FAT32 filesystem for the first (boot) partition:

```
# mkfs.vfat -F32 /dev/sda1
```

Create root filesystem:

```
# mkfs.ext4 /dev/zpln/root
```

Format swap logical volume as swap area and activate it:

```
# mkswap /dev/zpln/swap
# swapon /dev/zpln/swap
```

4.3.1.5. Mount prepared partitions

In this handbook, the `/mnt` directory is used as a default mountpoint.

```
# mount /dev/zpln/root /mnt
# mkdir /mnt/boot
# mount /dev/sda1 /mnt/boot
```

## 4.4. Install Base System

### 4.4.1. Download rootfs tarball

As mentioned before, the `/mnt` directory is the default mountpoint for our system. Let's download the rootfs tarball into this directory so we won't pollute live CD/DVD/USB RAM:

```
# cd /mnt
# wget -c
```

```
https://github.com/zeppe-lin/pkgsrc-core/releases/download/v1.0/rootfs-v1.0
-x86_64.tar.xz{,.sig}
```

#### 4.4.2. Verify downloaded tarball

Zeppe-Lin rootfs tarball is signed using GPG. It's a good practice to verify authenticity and integrity of downloaded files when possible.

```
# gpg --keyserver keyserver.ubuntu.com --recv-keys 59ec1986fbd902cf
# gpg --verify rootfs-v1.0-x86_64.tar.xz{.sig,}
```

#### 4.4.3. Extract rootfs tarball

Once the rootfs tarball is downloaded and verified, extract the contents with the following command:

```
# tar --numeric-owner --xattrs --xattrs-include='*' -xpf \
    rootfs-v1.0-x86_64.tar.xz
```

**Important**:

It is very important to use all the options included above. See below for details.

Here are what the options to *tar(1)* do:

**--numeric-owner**

Without this option, *tar(1)* will map ownership and group ownership based on the UID to user and GID to group mappings as defined on the Live CD/DVD/USB. By specifying **--numeric-owner** we tell that we want the *numeric values* of the UIDs and GIDs in the tarball to be preserved on disk, so when your Zeppe-Lin system boots, the UIDs and GIDs are set correctly for Zeppe-Lin.

**--xattrs --xattrs-include='*'**

Zeppe-Lin uses filesystem extended attributes to set Linux capabilities, which allow for certain programs such as ping to have enhanced privileges without having to be fully 'suid root'. Even with the **-p** option, *tar(1)* will not restore extended attributes we need unless these two options are specified.

**-xpf**

Extract (**x**), preserve regular permissions and ownership (**p**), and use the filename (**f**) specified.

### 4.5. Chroot Into Base System

Copy DNS configuration to have the availability to use the network from the chrooted root:

```
# cp /etc/resolv.conf etc/resolv.conf
```

Mount Linux's pseudo-filesystems:

```
# mount -B /dev /mnt/dev
# mount -B /tmp /mnt/tmp
# mount -B /run /mnt/run
# mount -t proc proc /mnt/proc
# mount -t sysfs none /mnt/sys
# mount -t devpts -o noexec,nosuid,gid=tty,mode=0620 devpts \
    /mnt/dev/pts

(UEFI only)
# mount -B /sys/firmware/efi/efivars /mnt/sys/firmware/efi/efivars
```

```
# chroot /mnt /bin/bash
```

Set the **SHELL** environment variable to */bin/bash* in the chrooted environment, to be available to call `:shell` in vim, for example, or other tools that relies on `SHELL`. This needs if the shell used outside chroot is zsh, fish, or something else that is not present in chroot environment:

```
(chrooted) # export SHELL=/bin/bash
```

### 4.6. Configure The Base System

Set the root password:

```
(chrooted) # passwd root
```

Make sure files have proper ownership and permissions:

```
(chrooted) # chown root:root /
(chrooted) # chmod 755 /
```

**Glibc** does not contain all possible locales, thus you'll have to generate the locales you need/use. To ensure the proper operation of *pkgmk(8)*, the locale `C.UTF-8` is generated as part of the Zeppe-Lin installation. Any other desired locales must be created by the administrator. Let's prepare `en_US.UTF-8` (you may choose locale you need):

```
(chrooted) # localedef -i en_US -f UTF-8 en_US.UTF-8
```

Then add `export LANG=en_US.UTF-8` to */etc/profile* to make it system-wide.

Edit */etc/fstab* (see *fstab(5)* for more information) to configure your filesystems and add the prepared partitions. For example, for *4.3.1. UEFI and LVM-on-LUKS* installation, you may specify the following:

```
/dev/zpln/root  /      ext4  defaults,noatime,nodiratime  1 2
/dev/sda1       /boot  vfat  defaults,noatime,nodiratime  1 2
/dev/zpln/swap  swap   swap  defaults                     0 0
```

**Note**:

> You can use `/dev/disk/by-uuid/*` or `UUID=xxx` instead of `/dev/*` used above to prevent boot failures on machines with multiple disks. Use *blkid(8)* to verify the disk's UUID:
>
> ```
> (chrooted) # blkid -o value -s UUID <DEVICE>
> ...
> ```

Uncomment */var/cache/pkgmk/work* if you want to build packages in RAM.

Mount **fstab** entries:

```
At least python3 won't build without that:
(chrooted) # mount /dev/shm

If you setup UEFI:
(chrooted) # mount /sys/firmware/efi/efivars

If you setup using tmpfs for /tmp:
(chrooted) # mount /tmp

If you setup to build packages in RAM:
(chrooted) # mount /var/cache/pkgmk/work
```

Edit */etc/rc.conf* (see *rc.conf(5)* for more information) to configure font, keyboard, timezone, hostname, and services.

Edit */etc/rc.d/net*, */etc/hosts*, and */etc/resolv.conf* to configure your network (IP address/Gateway/Hostname/Domain/DNS). If you want to configure the networking bridge, see */etc/rc.d/bridge*.

It's better to add the ordinary user now if you want a specific user id because the following installation of packages creates its own users and they may occupy your id:

```
(chrooted) # useradd --shell /bin/bash --create-home  \
    --groups audio,video,scanner,cdrom,input,users    \
    --uid 1000 --user-group $USERNAME

(chrooted) # passwd $USERNAME
```

To make this user privileged (e.g. as Ubuntu does), first, you need to add the `$USERNAME` to the `wheel` group:

```
(chrooted) # usermod -aG wheel $USERNAME
```

And second, grant the users in the `whell` group to be root:

```
#
# /etc/sudoers.d/00wheel: grant users in the wheel group to be root
#

%wheel ALL=(ALL:ALL) ALL

# End of file.
```

### 4.7. Prepare The PKGSRC Repos

Packages' sources are organized in so-called collections, see *6.1.3. The pkgsrc collections* for more information about that.

Clone the collections you need, but keep in mind that each subsequent collection depends on the previous ones.

```
(chrooted) # cd /usr/src/
(chrooted) # git clone https://github.com/zeppe-lin/pkgsrc-core    \
                --branch 1.x
(chrooted) # git clone https://github.com/zeppe-lin/pkgsrc-system  \
                --branch 1.x
(chrooted) # git clone https://github.com/zeppe-lin/pkgsrc-xorg    \
                --branch 1.x
(chrooted) # git clone https://github.com/zeppe-lin/pkgsrc-desktop \
                --branch 1.x
```

Where `--branch 1.x` corresponds to the current Zeppe-Lin release branch.

Edit */etc/pkgman.conf* (see *pkgman.conf(5)* for more information) and enable the collections you cloned. By default, only `pkgsrc-core` collection is enabled. Also, you may like to edit */etc/pkgmk.conf* (see *pkgmk.conf(5)* for more information).

### 4.8. Update The Base System

Before installing any package you need, it's highly recommended to update the system which you just installed.

**Warning**:

Starting from version `6.0`, `pkgutils` has broken backward compatibility when generating footprint files. So, at the beginning it is recommended to update `pkgmk` and `pkgutils`:

```
(chrooted) # pkgman update -f pkgmk pkgutils
```

The following command tells to *pkgman(1)* to run system update (with dependency handling/sorting) and to stop if installation of at least one package fails:

```
(chrooted) # pkgman sysup --deps --depsort --group
```

It's also recommended to merge the files that were rejected during updates:

```
(chrooted) # rejmerge
```

And it's a good idea to check which packages need to be rebuilt:

```
(chrooted) # revdep
```

If **revdep(1)** showed anything rebuild them as follows:

```
(chrooted) # pkgman update -fr --depsort `revdep`
```

## 4.9. Install essential packages

Let's install all the necessary packages for the so-called workstation:

```
(chrooted) # pkgman install --deps --group \
    cryptsetup e2fsprogs dosfstools grub2 grub2-efi iw gnupg \
    lvm2 pinentry wireless-tools wpa-supplicant dhcpcd iputils
```

Edit */etc/rc.d/dhcpcd* and */etc/rc.d/wpa_supplicant* and setup your network interfaces.

**Note**:

If you configured the networking bridge, don't forget to specify the bridge interface to **dhcpcd** and/or **wpa_supplicant** RC scripts.

## 4.10. Prepare a Linux Kernel

Follow only one of the two following subsections.

### 4.10.1. Install kernel package

The `pkgsrc-system` collection provides a package with the kernel and non-free firmware:

```
(chrooted) # pkgman install --deps --group \
    --config-append="runscripts no" linux linux-firmware
```

This **linux** package has a **post-install** script that executes *mkinitramfs(8)* and updates the **grub** config. We turn off the execution of this script because we'll do it manually. It is useful when regularly updating the system but now it's unnecessary.

**Note**:

You may like to lock the **linux** package and not update it during regular system updates, since the update will remove the current working kernel and its modules. If you plan not to turn off the computer for a long time, it is better to do this and update the kernel separately, through *pkgman-update(8)*. See *pkgman-lock(8)* for more information about package locking.

If you installed the kernel package through a package manager, you can jump over the next paragraph, to the *4.11. Prepare Initramfs Image*.

### 4.10.2. Install the kernel manually

Since the *pkgman(1)* build the package that contains only the binaries (kernel and modules), you may want to build your own kernel. For example, you do not need such a fat kernel, which goes by default, or you need the sources of the running kernel because you want to build the Nvidia driver and/or VirtualBox.

In this case, it is recommended to use the kernel from package sources.

So, let's download the **linux** source:

```
(chrooted) # pkgman install -do linux
```

The `-do` option means `download-only`. The source is downloaded to */var/cache/pkgmk/sources* if you have not changed the **PKGMK_SOURCE_DIR** location in */etc/pkgmk.conf*. Unpack it:

```
(chrooted) # tar -xvf \
    /var/cache/pkgmk/sources/linux-5.4.X.tar.?z -C /usr/src/
```

You may want to enable Zeppe-Lin's patches:

```
(chrooted) # cd linux-5.4.X
(chrooted) # for p in $(pkgman path linux)/*.patch; \
    do patch -Np1 -i $p; done
```

If you have downloaded a non-packaged kernel, you can print the results of applying the patches without actually changing any files. Just add `--dry-run` option.

Next, you can create your own, minimal config and adjust it to your system:

```
(chrooted) # make menuconfig
```

Or you can use the `all-inclusive` config from the package:

```
(chrooted) # cp $(pkgman path linux)/x86_64-dotconfig .config
(chrooted) # make olddefconfig
```

Build the kernel and modules:

```
(chrooted) # make -j$(nproc) all
```

Install the kernel and modules:

```
(chrooted) # KV=$(make kernelversion)
(chrooted) # cp arch/x86/boot/bzImage /boot/vmlinuz-$KV
(chrooted) # cp .config /boot/config-$KV
(chrooted) # make modules_install
```

You may still need to install the kernel non-free firmware:

```
(chrooted) # pkgman install --deps linux-firmware
```

Next, we need to prepare the initramfs and update our bootloader.

### 4.11. Prepare Initramfs Image

Install **mkinitramfs** package:

```
(chrooted) # pkgman install --deps mkinitramfs
```

Add the **mkinitramfs** configuration file (*/etc/mkinitramfs/config*) with the following content:

```
#
# /etc/mkinitramfs/config: mkinitramfs(8) configuration
#
# See mkinitramfs.config(5) for more information.
#

hostonly=1 #(optional)
compress="gzip --fast"
hooks="eudev luks lvm resume"
root=/dev/zpln/root
root_type=ext4
resume=/dev/zpln/swap
luks_name=crypt
luks_root=/dev/sda2

# End of file.
```

See *mkinitramfs.config(5)* for more information.

**Note**:

As in case of */etc/fstab*, you can use `/dev/disk/by-uuid/*` or `UUID=...` instead of `/dev/*` to prevent boot failures on machines with multiple disks. Use *blkid(8)* to verify the disk's `UUID`:

```
(chrooted) # blkid -o value -s UUID <DEVICE>
...
```

Now, prepare an initramfs. If you installed the `linux` kernel manually, you have already set `KV` variable to kernel version. Otherwise, obtain the kernel version from package source like the following:

```
(chrooted) # KV=$(pkgman printf %v --filter=linux)
```

Now, generate an initramfs image:

```
(chrooted) # mkinitramfs -o /boot/initramfs-$KV.img -k $KV
```

## 4.12. Install a Bootloader

### 4.12.1. GRUB

Create */etc/default/grub* file with the following content:

```
GRUB_TIMEOUT=3
GRUB_DISTRIBUTOR=ZPLN
GRUB_CMDLINE_LINUX_DEFAULT="quiet resume=/dev/zpln/swap"
```

Next, install **GRUB** on `/dev/sda` (your case may differ) drive:

```
(chrooted) # grub-install --target=x86_64-efi \
    --efi-directory=/boot /dev/sda
```

Update **GRUB** configuration file:

```
(chrooted) # grub-mkconfig -o /boot/grub/grub.cfg
```

## 4.13. Post-installation Tasks

### 4.13.1. Install X11

See the available X11 video and input drivers, and choose the right one:

```
# pkgman search -vv xf86-
```

Next, install the **xorg** package and the drivers you wish:

```
# pkgman install --deps --group xorg ...
```

### 4.13.2. Reboot

Exit from chroot and umount */mnt* and reboot:

```
(chrooted) # exit
# cd /
# umount -R /mnt
# shutdown -r now
```

# 5. THE PACKAGE SYSTEM

## 5.1. Introduction

### 5.1.1. Basic package management tools

The package system (**pkgutils**) is a fork of CRUX's pkgutils. It is made with simplicity in mind, where all packages are plain *tar.gz* files (i.e. without any kind of metadata).

Packages follow the naming convention *name#version-release.pkg.tar.gz*, where *name* is the name of the program, *version* is the version number of the program, and *release* is the version number of the package.

The *pkg.tar.gz* extension is used (instead of just *tar.gz*) to indicate that this is not just any *tar.gz* file, but a *tar.gz* that is meant to be installed using *pkgadd(8)*. This helps distinguish packages from other *tar.gz* files. Note that *pkgmk(8)* supports additional compression schemes like **bzip2** with the *tar.bz2* extension, **lzip** with *tar.lz*, **xz** with *tar.xz*, or **zstd** ending with *tar.zst*.

*pkgadd(8)*, *pkgrm(8)*, and *pkginfo(1)* are the basic package management utilities and are part of the package **pkgutils**. In addition to them, package management includes utilities such as *pkgmk(8)*, *rejmerge(8)*, and *revdep(1)*.

They could also be considered basic, but they are placed in separate packages so that changes in one utility do not require rebuilding all of them.

With these utilities, you can install, uninstall, inspect, make packages, query the package database, merge files that were rejected during package upgrades, and check for missing libraries of installed packages.

When a new package is installed using *pkgadd(8)*, a new record is added to the package database (stored in */var/lib/pkg/db*). The basic package system does not have any kind of dependency checking, this will not warn you if you install a package that requires other packages to be installed. The included *pkgman(1)* tool (**pkgman**), however, does support dependencies.

The following sections will describe in short how to use the package utilities. Additional information about these utilities can be found on their respective manual page.

## 5.2. Using the Package System

### 5.2.1. Installing a package

Installing a package is done by using *pkgadd(8)*. This utility requires at least one argument, the package you want to install. Example:

```
# pkgadd bash#5.0.18-1.pkg.tar.gz
```

When installing a package the package manager will ensure that no previously installed files are overwritten. If conflicts are found, an error message will be printed and *pkgadd(8)* will abort without installing the package. The error message will contain the names of the conflicting files. Example:

```
# pkgadd bash#5.0.18-1.pkg.tar.gz
bin/sh
usr/share/man/man1/sh.1.gz
pkgadd error: listed files already installed
    (use -f to ignore and overwrite)
```

To force the installation and overwrite the conflicting files, you can use the option **-f/--force**. Example:

```
# pkgadd -f bash#5.0.18-1.pkg.tar.gz
```

The package system allows a file to be owned by exactly one package. When forcing an installation the ownership of the conflicting files will be transferred to the package that is currently being installed. Directories can however be owned by more than one package.

**Warning**:

> It is often not a good idea to force the installation unless you really know what you are doing. If a package conflicts with already installed files it could be a sign that the package is broken and installs unexpected files. Use this option with extreme care, preferably not at all.

As earlier, the package file itself does not contain any metadata. Instead, the *pkgadd(8)* uses the package filename to determine the package name and version. Thus, when installing a package file named *bash#5.0.18-1.pkg.tar.gz*, *pkgadd(8)* will interpret this as a package named *bash* at version *5.0.18-1*. If *pkgadd(8)* is unable to interpret the filename (e.g. # is missing or the filename does not end with .pkg.tar.gz) an error message will be printed and *pkgadd(8)* will abort without installing the package.

### 5.2.2. Upgrading a package

Upgrading a package is done using *pkgadd(8)* with the **-u** option. Example:

```
# pkgadd -u bash#5.0.18-1.pkg.tar.gz
```

This will replace the previously installed bash package with the new one. If you have not previously installed bash, *pkgadd(8)* will print an error message. *pkgadd(8)* does not care about the version number of the package in that you can "upgrade" version 2.05-1 with version 2.04-1 (or even with version 2.05-1 itself). The installed package will be replaced with the specified package.

Upgrading a package is equivalent to executing *pkgrm(8)* followed by *pkgadd(8)* with one (big) exception. When upgrading a package (with `pkgadd -u`) you have the option to prevent some of the already installed files from getting replaced. This is typically useful when you want to preserve configuration and log files.

When executing *pkgadd(8)* the file */etc/pkgadd.conf* will be read. This file can contain rules describing how *pkgadd(8)* should behave when doing upgrades. A rule is built out of three fragments: *event*, *pattern*, and *action*. The *event* describes in what kind of situation this rule applies. Currently, only one type of event is supported, that is UPGRADE. The *pattern* is a filename pattern expressed as a regular expression and the *action* applicable to the UPGRADE event is YES or NO. More than one rule of the same event type is allowed, in which case the first rule will have the lowest priority and the last rule will have the highest priority. Example:

```
#
# /etc/pkgadd.conf: pkgadd(8) configuration
#

UPGRADE       ^etc/.*$               NO
UPGRADE       ^var/log/.*$           NO
UPGRADE       ^etc/X11/.*$          YES
UPGRADE       ^etc/X11/xorg.conf$    NO

# End of file.
```

The above example will cause *pkgadd(8)* to never upgrade anything in */etc/* or */var/log/* (subdirectories included), except files in */etc/X11/* (subdirectories included), unless it's the file */etc/X11/xorg.conf*. The default rule is to upgrade everything, rules in this file are exceptions to that rule.

**Caution**:

> A pattern should never contain an initial `/` since you are referring to the files in the package, not the files on the disk.

See *pkgadd.conf(5)* for more information.

5.2.2.1. Rejected files

If *pkgadd(8)* finds that a specific file should not be upgraded, it will install it under */var/lib/pkg/rejected/* . Files in this directory are never added to the package database. The user is then free to examine, use and/or remove that file manually. Another option is to use *rejmerge(8)*. For each rejected file found in */var/lib/pkg/rejected/*, *rejmerge(8)* will display the difference between the installed version and the rejected version. The user can then choose to keep the installed version, upgrade to the rejected version or perform a merge of the two. Example (using the above */etc/pkgadd.conf*):

```
# pkgadd -u bash#5.0.18-1.pkg.tar.gz
pkgadd: rejecting etc/profile, keeping existing version


# tree --charset=ascii /var/lib/pkg/rejected
/var/lib/pkg/rejected
`-- etc
    `-- profile
```

### 5.2.3. Removing a package

Removing a package is done by using *pkgrm(8)*. This utility requires one argument, the name of the package you want to remove. Example:

```
# pkgrm bash
```

**Warning**:

> This will remove all files owned by the package, no questions asked. Think twice before doing it and make sure that you did not misspell the package name since that could remove something completely different (e.g. think about what could happen if you misspelled glib as glibc).

### 5.2.4. Querying the package database

Querying the package database is done using *pkginfo(1)*. This utility has a few options to answer different queries.

```
+-------------------------+-----------------------------------------+
| Option                  | Description                             |
+-------------------------+-----------------------------------------+
| -f, --footprint=file    | print footprint for file                |
| -i, --installed         | list installed packages and their version |
| -l, --list=package|file | list files in package or file           |
| -o, --owner=pattern     | list owner(s) of file(s) matching pattern |
| -r, --root=rootdir      | specify an alternate root directory     |
| -v, --version           | print version and exit                  |
| -h, --help              | print help and exit                     |
+-------------------------+-----------------------------------------+
```

List installed packages and their version:

```
$ pkginfo -i
audiofile 0.2.3-1
autoconf 2.52-1
automake 1.5-1
...
xmms 1.2.7-1
zip 2.3-1
zlib 1.1.4-1
```

List files in package or file:

```
$ pkginfo -l bash
bin/
bin/bash
etc/
etc/profile
usr/
usr/share/man/
usr/share/man/man1/
usr/share/man/man1/bash.1.gz

$ pkginfo -l grep#2.5-1.pkg.tar.gz
usr/
usr/bin/
usr/bin/egrep
usr/bin/fgrep
usr/bin/grep
usr/share/man/
usr/share/man/man1/
usr/share/man/man1/egrep.1.gz
usr/share/man/man1/fgrep.1.gz
usr/share/man/man1/grep.1.gz
```

List owners of files matching *bin/ls*:

```
$ pkginfo -o bin/ls
e2fsprogs  usr/bin/lsattr
fileutils  bin/ls
modutils   sbin/lsmod
```

Print footprint for file:

```
$ pkginfo -f xorg-xkill#1.0.5-1.pkg.tar.gz
drwxr-xr-x      root/root       usr/
drwxr-xr-x      root/root       usr/bin/
-rwxr-xr-x      root/root       usr/bin/xkill
drwxr-xr-x      root/root       usr/share/
drwxr-xr-x      root/root       usr/share/man/
drwxr-xr-x      root/root       usr/share/man/man1/
-rw-r--r--      root/root       usr/share/man/man1/xkill.1.gz
```

This feature is mainly used by *pkgmk(8)* for creating and comparing footprints.

The **-r**/**--root** option should be used if you want to display information about a package that is installed on a temporarily mounted partition, which is "owned" by another system. By using this option you specify which database to use.

## 5.3. Package Management Front-end: pkgman

In its current form **pkgutils** does not have a concept of dependency handling. To address this, a front-end utility called **pkgman** exists. *pkgman(1)* supports dependency handling (with the caveat mentioned below) as well as some overlap with **pkgutils** features.

### 5.3.1. Functionality

Some examples of pkgman's functionality and use are as follows:

Listing installed packages:

```
$ pkgman list
acl
attr
```

```
...

$ pkgman list -v
acl 2.3.1-1
attr 2.5.1-1
...

$ pkgman list -vv
acl 2.3.1-1: Access Control List filesystem support
attr 2.5.1-1: Extended attribute support library for ACL support
...
```

Querying information about a package source:

```
$ pkgman info acl
Name:         acl
Path:         /usr/src/pkgsrc-core
Version:      2.3.1
Release:      1
Description:  Access Control List filesystem support
URL:          http://savannah.nongnu.org/projects/acl
Dependencies: attr
```

Searching for packages sources by name:

```
$ pkgman search -vv glibc
-- search ([i] = installed)
[i] glibc 2.32-5: GNU C Library
[i] glibc-32 2.32-5: GNU C Library (32bit)

$ pkgman search -vv --regex '^(glib)c?$'
-- search ([i] = installed)
[i] glib 2.70.1-1: Common C routines used by Gtk+ and other libs
[i] glibc 2.32-5: GNU C Library
```

Searching for packages sources by words in their description:

```
$ pkgman dsearch -vv archive
-- search ([i] = installed)
[i] cpio 2.13-2: Copy files into or out of a cpio or tar archive
[i] libarchive 3.5.2-1: Multi-format archive and compression
library
[ ] unrar 6.0.7-1: Extracts RAR archives
[ ] zip 3.0-1: Compression and file packaging/archive utility
```

Viewing dependency lists:

```
$ pkgman dep vim
acl
ncurses

$ pkgman dep vim --resursive
acl
 attr
ncurses

$ pkgman dep vim --recursive --tree -vv
-- dependencies ([i] = installed, --> seen before)
```

```
    [i] vim 9.1.0889-1: Highly configurable text editor
    [i]   acl 2.3.2-1: Access Control List filesystem support
    [i]     attr 2.5.2-1: Extended attribute support library for ACL
support
    [i]   ncurses 6.5-1: System V Release 4.0 curses emulation
library
```

Installing packages:

```
$ pkgman install --deps xterm
```

**Important**:

The `install` command **DOES NOT** process dependencies without **--deps** option, and it's usually recommended to use **--deps** with `install`. It is also worth noting that *pkgman(1)* by default tries to install all the packages specified in the dependencies, and if one of the dependencies does not build, it will skip it, and go further. In the end, it will show the `build-failed packages` in the report. This behavior can have unexpected consequences if you install many programs with many dependencies at a time. Because, one dependency may not build, and affect the building of another dependency, and thereby affect the third, etc.

We can tell *pkgman(1)* not to skip the fails, but to stop. Then we can fix the package build and start installation further.

There is an option **--group** for this. In this handbook, it is recommended to use it always, unless you know what you are doing.

```
$ pkgman install --deps --group xterm
```

In case one of the dependency builds has failed, just fix it and retry:

```
$ pkgman install --deps --group --force xterm
```

The option **--force** tell to *pkgman(1)* to skip installation of already installed package(s). It just ignores the package and installs next in the listed order. At the end of the installation procedure, all skipped packages will be in the report.

Viewing and updating outdated packages:

Since the packages sources for Zeppe-Lin are distributed via *git(1)*, the first thing to do is to update the `pkgsrc` collections:

```
$ git -C /usr/src/pkgsrc-core    pull
$ git -C /usr/src/pkgsrc-system  pull
$ git -C /usr/src/pkgsrc-xorg    pull
$ git -C /usr/src/pkgsrc-desktop pull
```

It's maybe annoying to do these few steps every time you want to synchronize `pkgsrc` repositories. If so, just add these steps to your *crond(8)* daemon and synchronize your local repos once a week, for example.

Listing installed packages that are out of date (including their new dependencies):

```
$ pkgman diff --deps --full
-- Differences between installed packages and packages' sources
Package                         Installed          Available

bind                            9.16.7-1           9.16.8-1
bindutils                                          9.16.8-1

--
1 update, 1 install
```

Updating an individual package:

```
$ pkgman update --deps --group bind
```

Updating all installed packages:

```
$ pkgman sysup --deps --depsort --group
```

#### 5.3.2. Configuration

pkgman's main configuration file, */etc/pkgman.conf*, contains options that can be used to change pkgman's behavior. Notably in this file, the following options can be configured:

* `pkgsrcdir`

  This option can occur multiple times and specifies a directory with a packages' sources "collection" which *pkgman(1)* should check in its operation. By default, the *core* collection is enabled, but *system*, *xorg*, *desktop*, and *stuff* collections are commented.

* `runscripts`

  This option configures *pkgman(1)* to run `pre-install`, `post-install`, `pre-remove`, and `post-remove` scripts if they exist in the package source directory. These scripts are run during *install*, *update*, *sysup*, and *remove* operations.

  It is recommended that this be enabled as in many cases if these scripts exist in a package source directory, it is required to be run for proper operation.

* `logfile`

  This option configures a file for pkgman to log its operation if desired.

This is **NOT** an exhaustive list of all of pkgman's commands, features, and configuration options, merely a starting point. More information can be found in *pkgman(1)* and *pkgman.conf(5)* manual pages.

### 5.4. Creating Packages

Creating a package is done using *pkgmk(8)*. This utility uses a file called *Pkgfile* (see *Pkgfile(5)* for more info), which contains information about the package (such as name, version, etc) and the commands that should be executed in order to compile the package in question. To be more specific, the *Pkgfile* file is actually a POSIX *sh(1p)* script, which defines a number of variables (name, version, release, and source) and a function (build). Below is an example of what a *Pkgfile* file might look like. The example shows how to package the *grep(1)* utility. Some comments are inserted for explanation.

```
# Specify the name of the package.
name=grep


# Specify the version of the package.
version=2.4.2


# Specify the package release.
release=1


# The source(s) used to build this package.
source=ftp://ftp.ibiblio.org/pub/gnu/$name/$name-$version.tar.gz


# The build() function below will be called by pkgmk when
# the listed source files have been unpacked.
build() {
        # The first thing we do is to cd into the source directory.
        cd $name-$version
```

```
        # Run the configure script with desired arguments.
        # In this case we want to put grep under /usr/bin and
        # disable national language support.
        ./configure --prefix=/usr --disable-nls

        # Compile.  Use the verbose flag (V=1) to see/log the
        # compilation flags at build time.
        make V=1

        # Install the files, BUT do not install it under /usr,
        # instead we redirect all the files to $PKG/usr by setting
        # the DESTDIR variable.  The $PKG variable points to a
        # temporary directory which will later be made into a
        # tar.gz-file.  Note that the DESTDIR variable is not used
        # by all Makefiles, some use prefix and others use ROOT,
        # etc.  You have to inspect the Makefile in question to
        # find out.  Some Makefiles do not support redirection at
        # all.  In those cases, you will have to create a patch for
        # it.
        make DESTDIR=$PKG install

        # Remove unwanted files, in this case, the info-pages.
        rm -rf $PKG/usr/info
}
```

In reality, you do not include all those comments, so the real Pkgfile for *grep(1)* looks like this:

```
# Description: Pattern matching utilities
# URL:         https://www.gnu.org/software/grep/grep.html

name=grep
version=2.4.2
release=1
source=https://ftpmirror.gnu.org/gnu/$name/$name-$version.tar.xz

build() {
        cd $name-$version

        ./configure --prefix=/usr --disable-nls

        make V=1
        make DESTDIR=$PKG install

        rm -rf $PKG/usr/info
}
```

Note that the `build()` function in the example above is just an example of how grep is built. The contents of the function can differ significantly if the program is built in some other way, e.g. does not use **autoconf**.

When the build() function has been executed, the `$PKG` directory will be made into a package named *name#version-release.pkg.tar.gz*. Before the package creation is completed, *pkgmk(8)* will check the content of the package against the *.footprint* file. If this file does not exist, it will be created and the test will be skipped. The *.footprint* file will contain a list of all files that should be in the package if the

build was successful or a list of all the files that were installed in $PKG (if the *.footprint* did not already exist). If there is a mismatch the test will fail and an error message will be printed. You should NOT write the *.footprint* file by hand. Instead, when a package has been upgraded and you need to update the contents of the *.footprint* file you simply do `pkgmk -uf`. This test ensures that a rebuild of the package turned out as expected.

If the package is built without errors it's time to install it by using *pkgadd(8)* and try it out. It's highly recommended to look at the *Pkgfile* in another package(s) since looking at examples is a great way to learn.

A detailed guideline is described in *Pkgfile(5)* manual page.

### 5.5. Configuring the Package Build Process

Many settings of the package build process can be configured by editing the */etc/pkgmk.conf* configuration file. Some of these configurable settings include:

* `CFLAGS`, `CXXFLAGS`

  Define optimization and architecture options for package compilation.

  It's best NOT change these settings unless you absolutely know what you're doing!

* `PKGMK_SOURCE_MIRRORS`

  Define location(s) from which pkgmk will attempt to fetch source archives.

* `PKGMK_SOURCE_DIR`

  Define location where pkgmk will store (if downloading) and use source archives when building.

* `PKGMK_PACKAGE_DIR`

  Define location where pkgmk will create package files once the build process is complete.

* `PKGMK_WORK_DIR`

  Define a work area that pkgmk will use to build the package.

Here are some examples:

```
PKGMK_SOURCE_MIRRORS="http://fileserver.intranet/dist/sources/"
```

This setting instructs pkgmk to attempt to fetch all source archives from *http://fileserver.intranet/dist/sources/* before falling back to the source URL specified in the Pkgfile. Multiple URLs can be separated by spaces (the spaces in the URL itself replace by `%20`!).

```
PKGMK_SOURCE_DIR="/var/cache/pkgmk/sources"
```

This setting instructs pkgmk to store and find source archives in `/var/cache/pkgmk/sources`. An example benefit of this setup would be the ability to store `/var/cache/pkgmk/sources` on an NFS server on your local network for use by multiple Zeppe-Lin installations. `PKGMK_PACKAGE_DIR` can be set and used the same way.

```
PKGMK_WORK_DIR="/var/cache/pkgmk/work/$name"
```

This setting instructs pkgmk to use `/var/cache/pkgmk/work/$name` as a work area for building the specified package. Building the **grep** package would result in the work area being `/var/cache/pkgmk/work/grep`. An alternative would be to use a *tmpfs(5)* as your work directory.

There are a few more settings that can be found on the *pkgmk.conf(5)* manual page.

# 6.0. THE PKGSRC SYSTEM

## 6.1. Introduction

### 6.1.1. What is a package source?

A package source is a directory containing the files needed for building a package using *pkgmk(8)*. This means that this directory at least has the files *Pkgfile* (which is the package build description) and *.footprint* (which is used for regression testing and contains a list of files this package is expected to contain once it is built). Further, a package source directory can contain patches and/or other files needed for building the package, even software sources. It is important to understand that the actual source code for the package is not necessarily present in the package source directory. Instead, the *Pkgfile* contains an URL that points to a location where the source can be downloaded.

The use of the words "package source" in this context is borrowed from the NetBSD world, where a package refers to a set of files and building instructions that describe what's necessary to build a certain piece of software.

### 6.1.2. What is a pkgsrc repository?

The term *pkgsrc repository* refers to a remote repository containing packages' sources: build scripts and files for building binary packages.

### 6.1.3. The pkgsrc collections

Packages' sources are organized in so-called collections. There are four official collections:

1. *pkgsrc-core*

   This collection contains build scripts and files for the packages required to create a basic system, on the basis of which you can build your installation. The official rootfs images are built based on this collection.

   Of course, this collection shouldn't have dependencies outside itself. To enforce this rule during maintenance see *Makefile* in the collection root directory. This *Makefile* checks collection files for typical errors, including misconfigured dependencies, and is included in the other collections too. Also, see *pkgmaint* and corresponding **system/pkgmaint** package for various package maintaining utilities. It will not be superfluous to mention *revdep(1)*.

2. *pkgsrc-system*

   This collection contains build scripts and files for the packages required for minimal installation on real hardware: boot loader, filesystem utilities, hw-monitors, servers, and so on. Packages in this collection depend **only** on the `pkgsrc-core` collection.

3. *pkgsrc-xorg*

   This collection contains build scripts and files for the packages required for X Window System applications. Packages in this collection depend on the `pkgsrc-system` collection.

4. *pkgsrc-desktop*

   This collection contains build scripts and files for the packages typically used on a desktop: web browser, music/video player, file manager, chat messenger, etc. Packages in this collection depend on the `pkgsrc-xorg` collection.

In addition to the official ones, there are unofficial collections of packages:

* *pkgsrc-stuff*

  This collection contains build scripts and files for the packages that do not fit into the above categories. Packages in this collection may depend on any of the collections listed above. Since this collection contains packages of different (often dubious) quality and purpose, we will not use it in the examples and warn you that if you can live without it - it's better to do so.

* *pkgsrc-games*

  This collection contains build scripts and files for the games and emulators.

** pkgsrc-wmaker*

This collection contains build scripts and files for Window Maker & dockapps packages.

## 6.2. Using The Pkgsrc

### 6.2.1. Synchronizing your local pkgsrc repositories

When Zeppe-Lin is installed for the first time there are no local pkgsrc collections (*/usr/src/pkgsrc**). To obtain the collection you need the first time, use `git clone`. Example:

```
# cd /usr/src/
# git clone https://github.com/zeppe-lin/pkgsrc-core    --branch 1.x
# git clone https://github.com/zeppe-lin/pkgsrc-system  --branch 1.x
# git clone https://github.com/zeppe-lin/pkgsrc-xorg    --branch 1.x
# git clone https://github.com/zeppe-lin/pkgsrc-desktop --branch 1.x
```

Where `--branch 1.x` corresponds to the current Zeppe-Lin release branch.

Edit */etc/pkgman.conf* and enable the collections you cloned. By default, only `pkgsrc-core` collection is enabled. See *pkgman.conf(5)* for more information.

To bring your local collections up to date use `git pull`. Example:

```
# git -C /usr/src/pkgsrc-core     pull
# git -C /usr/src/pkgsrc-system   pull
# git -C /usr/src/pkgsrc-xorg     pull
# git -C /usr/src/pkgsrc-desktop  pull
```

Depending on what collections you already have.

It's maybe annoying to do these few steps every time you want to synchronize `pkgsrc` repositories. If so, just add these steps to your *crond(8)* daemon and sync your local repos once a week, for example.

### 6.2.2. Listing local pkgsrc repositories

When the local pkgsrc repositories have been cloned/updated the directory `/usr/src/` will contain some `pkgsrc-???` directories.

Under each of these directories, you will find packages' sources: build scripts and files for building binary packages. You can simply browse around in the directory structure to find out which packages' sources are available. Let's take `pkgsrc-core` as an example:

```
$ cd /usr/src/pkgsrc-core
$ ls -p
acl/              gcc/           libpcre/          psmisc/
asciidoctor/      gdbm/          libpcre2/         py3-setuptools/
attr/             gettext-tiny/  libpipeline/      python3/
autoconf/         git/           libtirpc/         rc/
automake/         glibc/         libtool/          rdate/
...
```

You can also use *pkgman(1)* with command **list** and option **--all** to list all local packages' sources. Example:

```
$ pkgman list --all
-- list ([i] = installed)
[i] acl
[i] alsa-lib
[ ] alsa-ucm-conf
[i] alsa-utils
...
```

**Note**:

**pkgman** will list the packages' sources from all `pkgsrc` collections indicated in */etc/pkgman.conf* as `pkgsrcdir`.

Add the **--path** option to see the package source location:

```
$ pkgman list --all --path
-- list ([i] = installed)
[i] /usr/src/pkgsrc-core/acl
[i] /usr/src/pkgsrc-system/alsa-lib
[ ] /usr/src/pkgsrc-system/alsa-ucm-conf
[i] /usr/src/pkgsrc-system/alsa-utils
...
```

If you are looking for a specific package source, it might be easier to use **search** command instead of `list | grep`:

```
$ pkgman search --path alsa
-- search ([i] = installed)
[i] /usr/src/pkgsrc-system/alsa-lib
[ ] /usr/src/pkgsrc-system/alsa-ucm-conf
[i] /usr/src/pkgsrc-system/alsa-utils
```

### 6.2.3. Listing version differences

To find out if the **pkgsrc** repository carries packages that are different (likely newer) compared to the versions currently installed you can use *pkgman-diff(1)*. If versions differences are found, the output could look like this:

```
$ pkgman diff --deps --full

-- Differences between installed packages and packages sources tree
Package                         Installed           Available

aircrack-ng-scm                 20211113-1          20211121-1
cowpatty                        4.8-1               4.8-2
feh                             3.7.2-1             3.7.2-2
joomscan-scm                    20211112-1          20211121-1
trinity-extra-theme             0.4-3cc4340-2       0.1-1
handbook                        0.3-1               0.3.1-1

-- Packages which was not found in the packages sources tree
Package                         Installed           Required by

libncurses5-compat                                  android-ndk-bin

--
6 updates
```

The option **--deps** calculates the new dependencies for available packages. The option **--full** shows the table above, not just the list of packages.

Pay attention to `trinity-extra-theme` package. The available package has a lower version than the installed one. You can use **--config-set**="preferhigher yes" to prefer higher installed versions over lower packages, overwriting default **preferhigher** configuration settings.

If no version differences were found, i.e. the system is in sync with the packages sources structure, the output will simply be:

```
$ pkgman diff
No differences found
```

### 6.2.4. Building and installing packages

**Note**:

> The recommended way is to use *pkgman-install(8)*. Here we will only describe the steps to better understand the build process.

Once you have found a package that you want to build and install you simply go into the desired package source directory and use *pkgmk(8)* to build it. Example:

```
# cd /usr/src/pkgsrc-core/gawk
# pkgmk -d
```

The **-d** option means download missing source files and tells *pkgmk(8)* to download the source(s) specified in *Pkgfile* (in case the source files are already downloaded this option is ignored). When the download is completed the package will be built.

**Warning**:

> It is often not a good idea to build the packages with root privileges. Some recipes for building programs (Makefiles, etc) may not be written correctly, and overwrite system files.

If you will build packages with unprivileged user, the installed files will have user permissions. This is wrong, both in terms of security and distribution of these packages. It is better to use *fakeroot(1)* to run *pkgmk(8)* in an environment faking root privileges. Example:

```
$ fakeroot pkgmk -d
```

The *pkgman(1)* already uses it in the default Zeppe-Lin distribution.

If the package was built successfully you can use *pkgadd(8)* to install or upgrade it. Example:

```
# pkgadd gawk#3.1.5-3.pkg.tar.gz
```

### 6.2.5. Build packages as an unprivileged user

It is already done in the default Zeppe-Lin distribution. The packages `core/filesystem`, `core/pkgmk`, and `core/pkgman` have corresponding configurations. However, it does not hurt to describe how to achieve this, so that you will know how it is done in Zeppe-Lin or if you would like to adjust default settings.

First, add a new user (`pkgmk` for example):

```
# useradd -b /var/cache/ -m -s /bin/false -U pkgmk
```

This command will create the user `pkgmk` and the group with the same name, with */var/cache/pkgmk* as the home directory and */bin/false* as the default shell.

Second, add the directories required for *pkgmk(8)* to build the packages and set user `pkgmk` (which we created previously) as their owner:

```
# mkdir -p /var/cache/pkgmk/{sources,packages,work}
# chown -R pkgmk:pkgmk /var/cache/pkgmk/{sources,packages,work}
```

Third, specify these directories in */etc/pkgmk.conf*, so that when you call *pkgmk(8)*, these directories will be used.

```
PKGMK_SOURCE_DIR="/var/cache/pkgmk/sources"
PKGMK_PACKAGE_DIR="/var/cache/pkgmk/packages"
```

```
PKGMK_WORD_DIR="/var/cache/pkgmk/work/$name-$$"
```

Fourth, tell *pkgman(1)* to call *pkgmk(8)* on behalf of `pkgmk` user. To do that, edit variable `makecommand` in the */etc/pkgman.conf* file as the following:

```
makecommand sudo -H -u pkgmk fakeroot pkgmk
```

### 6.2.6. Renice pkgmk's child process

If you keep using your computer while compiling packages you will notice that your box is much less responsive than usual. This is caused by having two `groups` of processes with the same nice priority: your usual running tasks on one side, and *pkgman(1)* (and its child processes) on the other. Now, if you could renice *pkgman(1)* and its children to a higher nice (i.e. lower priority!) value (default is 0), compiling would inevitably take somewhat longer, but you could use your workstation without noticing much difference to its usual performance.

Just set up the `makecommand` in the */etc/pkgman.conf* to something like this:

```
makecommand sudo -H -u pkgmk nice -n10 ionice -c2 -n6 fakeroot pkgmk
```

### 6.2.7. Build in ram

By default, Zeppe-Lin already has `pkgmk` user and */etc/fstab*'s entry for this. Just edit */etc/fstab*'s `pkgmk` entry, remove `-pipe` from */etc/pkgmk.conf*, and type `mount pkgmk`.

Below is described how this was done in Zeppe-Lin.

Find your user id for pkgmk:

```
$ id pkgmk
```

Edit */etc/fstab*:

Add the following line to */etc/fstab*

```
pkgmk /var/cache/pkgmk/work tmpfs size=<SIZE>,uid=<UID>,defaults 0 0
```

The `SIZE` is the memory size for */var/cache/pkgmk/work*. Add the postfix `M` for megabytes, or `G` for gigabytes. Example: `16G`.

The `UID` is the `pkgmk`'s user id.

Mount the tmpfs:

```
# mount pkgmk
```

Edit */etc/pkgmk.conf*:

Set where you put your RAM filesystem from */etc/fstab* file:

```
PKGMK_WORK_DIR="/var/cache/pkgmk/work/$name"
```

Remove `-pipe` from all your `CFLAGS` and `CXXFLAGS`.

# 7. CONFIGURATION

## 7.1. Generating Locales

**glibc** does not contain all possible locales, thus you'll have to generate the locales you need/use. To ensure the proper operation of *pkgmk(8)*, the locale `C.UTF-8` is generated as part of the **glibc** package. Any other desired locales must be created by the administrator.

The following example is a typical setup for US users, replace `en_US*` with the locale you want:

```
# localedef -i en_US -f UTF-8 en_US.UTF-8
```

To use this locale system-wide, add `export LANG=en_US.UTF-8` to */etc/profile*.

## 7.2. Initialization Scripts

### 7.2.1. Runlevels

The following runlevels are used in Zeppe-Lin (defined in */etc/inittab*):

| Runlevel | Description |
|---|---|
| 0 | Halt |
| 1 (S) | Single-User Mode |
| 2 | Multi-User Mode |
| 3-5 | (Not Used) |
| 6 | Reboot |

See *inittab(5)* for more information about runlevels.

### 7.2.2. Layout

The initialization scripts used in Zeppe-Lin follow the BSD-style (as opposed to the SysV-style) and have the following layout:

| File | Description |
|---|---|
| /etc/rc | System boot script |
| /etc/rc.single | Single-user startup script |
| /etc/rc.multi | Multi-user startup script |
| /etc/rc.modules | Module initialization script |
| /etc/rc.local | Local multi-user startup script |
| /etc/rc.shutdown | System shutdown script |
| /etc/rc.conf | System configuration file |
| /etc/rc.d/ | Service start/stop directory |

*/etc/rc.local* is empty by default.

Modify */etc/rc.modules*, */etc/rc.local*, and */etc/rc.conf* according to your needs. See *rc.conf(5)* and *rc(8)* for more information about system configuration file and initialization scripts.

## 7.3. Network Configuration

### 7.3.1. Static address

The network configuration is found in the RC script */etc/rc.d/net*. To enable this service you need to add `net` to the `SERVICES` string in */etc/rc.conf*. By default, this RC script configures a static IP address. Example:

```
#!/bin/sh -e
#
# /etc/rc.d/net: start/stop network interface
#

DEV=enp11s0

ADDR=192.168.1.100
MASK=24
GW=192.168.1.1

case $1 in
start)
        /sbin/ip addr add $ADDR/$MASK dev $DEV broadcast +
        /sbin/ip link set $DEV up
        /sbin/ip route add default via $GW
        ;;
stop)
        /sbin/ip route del default
        /sbin/ip link set $DEV down
        /sbin/ip addr del $ADDR/$MASK dev $DEV
        ;;
restart)
        $0 stop
        $0 start
        ;;
*)
        echo "usage: $0 [start|stop|restart]"
        ;;
esac

# End of file.
```

You will also need to configure DNS settings in */etc/resolv.conf*. Example:

```
#
# /etc/resolv.conf: resolver configuration file
#

search <your internal domain>
nameserver <your DNS server>

# End of file.
```

### 7.3.2. Dynamic address

If you want to configure your system to use a dynamic IP address, install the `dhcpcd` package, edit and run */etc/rc.d/dhcpcd* service:

```
# $EDITOR /etc/rc.d/dhcpcd
# /etc/rc.d/dhcpcd start
```

### 7.3.3. Wireless network

Before using wireless networking, use *rfkill(8)* to check whether the relevant interfaces are soft- or hard-blocked:

```
$ rfkill list
# rfkill unblock <ID|TYPE>
```

Next, install the `wpa-supplicant` package. It includes utilities to configure wireless interfaces and handle wireless security protocols. To use `wpa_supplicant`, you will need to edit (specify your wireless interface) and enable */etc/rc.d/wpa_supplicant* service.

To use **WPA-PSK**, generate a pre-shared key with *wpa_passphrase(8)* and append the output to the */etc/wpa_supplicant.conf* file:

```
# wpa_passphrase <MYSSID> <PASSPHRASE> >> /etc/wpa_supplicant.conf
```

Then edit and run the following service files: */etc/rc.d/wpa_supplicant*, */etc/rc.d/dhcpcd*.

```
# $EDITOR /etc/rc.d/wpa_supplicant
# $EDITOR /etc/rc.d/dhcpcd
# /etc/rc.d/wpa_supplicant start
# /etc/rc.d/dhcpcd start
```

To use **WPA-EAP** generate the password hash like this:

```
# echo -n <PASSPHRASE> | iconv -t utf16le | openssl md4
```

For **WEP** configuration, add the following lines to your */etc/wpa_supplicant.conf*:

```
network={
    ssid="MYSSID"
    key_mgmt=NONE
    wep_key0="YOUR AP WEP KEY"
    wep_tx_keyidx=0
    auth_alg=SHARED
}
```

## 7.4. Passwords, User environment, and PAM modules

### 7.4.1. Passwords

Zeppe-Lin uses SHA512 passwords by default. To change the password encryption method set the ENCRYPT_METHOD variable in */etc/login.defs* to DES, MD5, or SHA256.

Furthermore, when compiling programs that use the *crypt(3)* function to authenticate users you should make sure that these programs are linked against the **libcrypt** library (i.e. use `-lcrypt` when linking) which contains the SHA512 version of the crypt function (this version is backward compatible and understands DES passwords as well).

### 7.4.2. User environment

Configuration settings in */etc/login.defs* like CREATE_HOME and USERGROUPS_ENAB control the behaviour of *useradd(8)* when creating new users.

Creating a new user via `useradd -m` will not populate the home directory with a basic shell startup file, like other distributions whose */etc/skel* contains the idea of an initial home directory.

The PATH value for shells that use */etc/profile* is consistent regardless of UID and is set to `/sbin:/usr/sbin:/bin:/usr/bin`. This configuration allows unprivileged users easy access to administrative commands without needing to type the full path, provided they use *bash(1)* or *dash(1)*/*sh(1p)* and have appropriate *sudo(8)* permissions.

**Note**:

> Zeppe-Lin users can create */etc/skel* skeletons, choose different shells and manage PATH themselves. The default setup aims to reduce confusion.

### 7.4.3. PAM modules

The core packages `linux-pam` and `dumb-runtime-dir` provide a variety of modules that can be loaded upon logging in. The files in */etc/pam.d* govern the association between the type of login (e.g., tty, SSH, su, X Display Manager) and the modules that get loaded (e.g., pam_env, pam_exec, pam_limits). Typical situations that can be handled cleanly with PAM modules are listed in the table below.

| File in /etc/pam.d | Typical usage |
|---|---|
| pam_dumb_runtime_dir.so | Create an XDG_RUNTIME_DIR for apps that confirm to the freedesktop.org specification. |
| pam_env.so | Export some common environment variables, no matter what login shell the user has chosen. |
| pam_limits.so | Increase the allowed number of opened files, to ensure proper operation of some games. |
| pam_xauth.so | Grant another user access to the X display of the logged-in user, so that programs invoked with su can work properly. |
| pam_mount.so | Automatically mount a LUKS-encrypted home partition after successful authentication. |

**Note**:

The existence of a writable `XDG_RUNTIME_DIR` is required for proper operation of many desktop applications. A **clean** Zeppe-Lin installation (starting from `v1.0`) will place a line in */etc/pam.d/common-session* that loads the module *pam_dumb_runtime_dir.so* to satisfy this requirement. An **upgrade** to Zeppe-Lin `v1.0` might not do so, depending on your `UPGRADE` directives in */etc/pkgadd.conf* or may require *rejmerge(8)* to update your */etc/pam.d* configuration.

*pam_dumb_runtime_dir.so* has a simple design and is limited to creating the runtime directory and exporting `XDG_RUNTIME_DIR`. It's not managing all environment variables defined in the freedesktop.org specification.

If you find yourself in one of the other situations in the table above, read the man page for the corresponding PAM module to learn how to accomplish the desired configuration.

## 8. REPORTING BUGS

For bug reports, use the issue tracker at: *https://github.com/zeppe-lin/handbook/issues*.