Documentation on the NetBSD Packages System

Jul 2000

Documentation on the NetBSD Packages System

by Hubert Feyrer and Alistair Crooks

Table of Contents

I. Introduction ..... **??**
1. Intro ..... **??**
1.1. Overview ..... **??**
1.2. Terminology ..... **??**
II. User's Guide ..... **??**
2. Installing a precompiled binary package ..... **??**
2.1. Where to get ..... **??**
2.2. How to use ..... **??**
2.2.1. Local packages ..... **??**
2.2.2. Packages available via FTP ..... **??**
2.2.3. Tips & Tricks ..... **??**
2.3. A word of warning ..... **??**
3. Installing by Building ..... **??**
3.1. Where to get pkgsrc ..... **??**
3.2. Fetching distfiles ..... **??**
3.3. How to build and install ..... **??**
4. Making a precompiled package ..... **??**
III. Package Constructor's Guide ..... **??**
5. Package components - files, directories and contents ..... **??**
5.1. `Makefile` ..... **??**
5.2. `files/*` ..... **??**
5.3. `patches/*` ..... **??**
5.4. `pkg/*` ..... **??**
5.4.1. Mandatory files ..... **??**
5.4.2. Optional files ..... **??**
5.5. `scripts/*` ..... **??**
5.6. `work/*` ..... **??**
5.7. Importing the package into CVS ..... **??**
6. `PLIST*` issues ..... **??**
6.1. Miscellaneous ..... **??**
6.2. MD/MI vs. general `PLIST` ..... **??**
6.2.1. `PLIST_SRC` ..... **??**
6.2.2. `PLIST-mi`, `PLIST-md.shared`, `PLIST-md.static` ..... **??**
6.2.3. Order in the PLIST* file(s) ..... **??**
7. Notes on fixes for packages ..... **??**
7.1. CPP defines ..... **??**
7.2. Shared libraries - libtool ..... **??**
7.3. Using libtool on GNU packages that already support libtool ..... **??**

7.4. Gotchas of FreeBSD ports ..... **??**
7.5. Feedback to the author ..... **??**
8. The build process ..... **??**
8.1. Program locations ..... **??**
8.2. Main targets ..... **??**
8.3. Other helpful targets ..... **??**
9. Debugging ..... **??**
10. FAQs & features of the package system ..... **??**
10.1. Packages using GNU autoconfig ..... **??**
10.2. Other distrib methods than `.tar.gz` ..... **??**
10.3. Packages not creating their own subdirectory ..... **??**
10.4. Custom configuration process ..... **??**
10.5. Packages not building in their `DISTNAME` directory ..... **??**
10.6. How to fetch all distfiles at once ..... **??**
10.7. How to fetch files from behind a firewall ..... **??**
10.8. If your patch contains an RCS ID ..... **??**
10.9. How to pull in variables from `/etc/mk.conf` ..... **??**
10.10. Is there a mailing list for pkg-related discussion? ..... **??**
10.11. How do I tell "`make fetch`" to do passive FTP? ..... **??**
10.12. Dependencies on other packages ..... **??**
10.13. Conflicts with other packages ..... **??**
10.14. Software which has a WWW Home Page ..... **??**
10.15. How to handle modified distfiles with the 'old' name ..... **??**
10.16. What does `Don't know how to make /usr/share/tmac/tmac.andoc` mean? . **??**
10.17. How to handle incrementing versions when fixing an existing package ..... **??**
10.18. `Could not find bsd.own.mk` - what's wrong? ..... **??**
11. Submitting ..... **??**
11.1. Precompiled binary packages ..... **??**
11.2. packages ..... **??**
12. A simple example of a package: bison ..... **??**
12.1. Files ..... **??**
12.1.1. `Makefile` ..... **??**
12.1.2. `pkg/COMMENT` ..... **??**
12.1.3. `pkg/DESCR` ..... **??**
12.1.4. `pkg/PLIST` ..... **??**
12.2. Checking a package with "`pkglint`" ..... **??**
12.3. Steps for building, installing, packaging ..... **??**
A. Build logs ..... **??**
A.1. Building top ..... **??**
A.2. Packaging top ..... **??**
B. Layout of the FTP server's package archive ..... **??**

I. Introduction

Chapter 1. Intro

There is a lot of software freely available for Unix based systems, which usually runs on NetBSD, too, sometimes with some modifications. The NetBSD Packages Collection incorporates any such changes necessary to make that software run on NetBSD, and makes the installation (and deinstallation) of the software package easy by means of a single command.

The NetBSD Package System is used to enable such freely available third-party software to be built easily on NetBSD hosts. Once the software has been built, it is manipulated with the pkg_* tools so that installation and de-installation, printing of an inventory of all installed packages and retrieval of one-line comments or more verbose descriptions are all simple.

The NetBSD Package System and parts of the NetBSD Packages Collection are derived from FreeBSD and it's ports collection.

1.1. Overview

This document is divided into two parts. The first, "User's Guide", describes how one can use one of the packages in the Package Collection, either by installing a precompiled binary package, or by building your own copy using the NetBSD package system. The second part, "Package Constructor's Guide", explains how to prepare a package so it can be easily built by other NetBSD users without knowing about the package's building details.

1.2. Terminology

There has been a lot of talk about "ports", "packages", etc. so far. Here is a description of all the terminology used within this document:

Package:

A set of files and building instructions that describe what's necessary to build a certain piece of software using the NetBSD package system. Packages are traditionally stored under `/usr/pkgsrc`.

The NetBSD Package System:

This is the part of the NetBSD operating system handling building (compiling), installing, and removing of packages.

Distfile:

This term describes the file or files that are provided by the author of the piece of freely available software to distribute his work. All the changes necessary to build on NetBSD are reflected in the corresponding package. Usually the distfile is in the form of a compressed tar-archive, but other types are possible, too. Distfiles are stored below `/usr/pkgsrc/distfiles`.

Port:

This is the term used by FreeBSD people for what NetBSD calls a package. In NetBSD terminology, "port" refers to the code for getting NetBSD going on a certain hardware architecture.

Precompiled (binary) package:

A set of binaries built by the NetBSD package system from a distfile using the NetBSD package system and stuffed together in a single .tgz file so it can be installed on machines of the same machine architecture without the need to recompile. Packages are generated in `/usr/pkgsrc/packages` by the NetBSD package system; there is also an archive on ftp.netbsd.org.

Sometimes, this is referred to by the term "package" too, especially in the context of precompiled packages.

Program:

The piece of software to be installed which will be constructed from all the files in the Distfile by the actions defined in the corresponding package.

II. User’s Guide

Chapter 2. Installing a precompiled binary package

This section describes how to find, retrieve and install a precompiled binary package that someone else already prepared for your type of machine.

2.1. Where to get

Precompiled packages are stored on ftp.netbsd.org and its mirrors in the directory `/pub/NetBSD/packages` for anon FTP access. Please pick the right subdirectory there as indicated by "`sysctl hw.machine_arch`". In that directory, there is a subdirectory for each category, plus a subdirectory `All` which includes the actual binaries in .tgz-files. The category subdirectories use symbolic links to those files. (This is the same directory layout as in `/usr/pkgsrc/packages`).

This same directory layout applies for CDROM distributions, only that the directory may be rooted somewhere else, probably somewhere below `/cdrom`. Please consult your CDROM's documentation for the exact location!

2.2. How to use

This section describes how to install binary packages from CDROM, local disk or via FTP. Regardles of where you install from, be sure to have `/usr/pkg` and `/usr/X11R6` in your $PATH so you can actually start the just installed program.

2.2.1. Local packages

If you have the files on a CDROM or downloaded them to your hard disk, you can install them with the following command (be sure to "`su`" to root first):

`pkg_add` *`/path/to/package-vers.tgz`*

Examples:

```
# pkg_add /usr/pkgsrc/packages/All/wget-1.5.3.tgz
# pkg_add /cdrom/packages/All/wget-1.5.3.tgz
```

If you want to see what's going on, you can always add the -v option to "`pkg_add`".

2.2.2. Packages available via FTP

If you have FTP access and you don't want to download the packages via FTP prior to installation, you can do this automatically by giving pkg_add an ftp-URL:

```
pkg_add ftp://ftp.netbsd.org/pub/NetBSD/packages/OS-Version/arch/All/package-vers.tgz
```

If there is any doubt about which *OS-Version* and *arch* you need to use, the "sysctl" utility can be used to determine them by running "`sysctl kern.osrelease hw.machine_arch`".

Example:

```
# sysctl kern.osrelease hw.machine_arch
kern.osrelease = 1.4.2
hw.machine_arch = i386
# pkg_add ftp://ftp.netbsd.org/pub/NetBSD/packages/1.4.2/i386/All/wget-1.5.3.tgz
#
```

Note that any prerequisite packages needed to run the package in question will be installed too, assuming they are present where you install from. You do not need to handle these manually, your desired package will arrange for everything to get installed that it needs.

2.2.3. Tips & Tricks

If you don't know a package's version to install it, you can simply omit it, i.e.:

```
pkg_add /path/to/package
pkg_add ftp://ftp.netbsd.org/pub/NetBSD/packages/OS-Version/arch/All/package
```

will find out the most recent version of *package*, and then install it. Examples:

```
# pkg_add /usr/pkgsrc/packages/i386ELF/All/wget
# pkg_add ftp://ftp.netbsd.org/pub/NetBSD/packages/1.4.2/i386/All/wget
```

When installing packages via FTP and you don't give it a version number, don't be surprised by any errors you see - "`pkg_add`" will first try to find the exact file you give it, before trying to find any version, and as the exact file isn't there, there will be an error. If you use -v on "`pkg_add`", you'll see what's going on in detail.

If you want to install many packages and don't want to type the full path to the package each time, you can set the PKG_PATH environment variable to contain a semi(!)colon seperate list of locations that may contain packages. Example:

```
# PKG_PATH=/usr/pkgsrc/packages/All';'ftp://ftp.netbsd.org/pub/NetBSD/packages/1.4.2/spa
# export PKG_PATH
```

After you've set this (maybe in your root's `.cshrc` or `.profile`), you can then just install a package without giving it's path/URL or version. Example:

```
# pkg_add wget
#
```

2.3. A word of warning

Please pay very careful attention to the warnings expressed in the "`pkg_add`" manual page about the inherent dangers of installing binary packages which you did not create yourself, and the security holes that can be introduced onto your system by indiscriminate adding of such files.

Chapter 3. Installing by Building

This assumes that the package is already part of the NetBSD Package System. If it is not and you want to make it ready for pkgsrc„ then you are advised to read part II of this document, “Package Constructor’s Guide”.

3.1. Where to get pkgsrc

To get the package source going, you need to get the `pkgsrc.tar.gz` file from ftp://ftp.netbsd.org/pub/NetBSD-current/tar_files/pkgsrc.tar.gz and unpack it into `/usr/pkgsrc`.

As an alternative, you can get pkgsrc via the Software Update Protocol, SUP. To do so, make sure your supfile has a line saying “release=pkgsrc” in it, see the examples in `/usr/share/examples/supfiles`, and that the directory `/usr/pkgsrc` does exist. Then, simply start “`sup -v /path/to/your/supfile`”.

3.2. Fetching distfiles

The distribution file (i.e. the unmodified source) must exist on your system for the packages system to be able to build it. If it does not, then ftp(1) is used to fetch the distribution files automatically.

You can overwrite some of the major distribution sites to fit to sites that are close to your own. Have a look at `/usr/pkgsrc/mk/mk.conf.example` to find some examples. This may save some of your bandwidth and time. When you have selected your settings, install your configuration into `/etc/mk.conf`.

If you don’t have a permanent Internet connection and you want to know which files to download, “`make fetch-list`” will give you a shell script that will help you downloading the files. When done, put them into `/usr/pkgsrc/distfiles`.

3.3. How to build and install

Assuming you have fetched the distfiles, become root and change into the relevant subdirectory of `/usr/pkgsrc`. Then you can type

```
make
```

at the shell prompt to build the various components of the package, and

```
make install
```

at the shell prompt to install the various components into the correct places on your system.

Taking the top system utility as an example, we can install it on our system by building as shown in appendix *Building top*.

The program is installed under the default root of the packages tree, `/usr/pkg`. Should this not conform to your tastes, simply set the `LOCALBASE` variable in `/etc/mk.conf`, and it will use that value as the root of your packages tree. So, to use `/usr/local`, put

```
LOCALBASE=/usr/local
```

in your `/etc/mk.conf` file. There is, of course, one exception to this. X11 packages are traditionally installed in the X11 tree, which is identified by `X11BASE` and defaults to `/usr/X11R6`.

It is possible to install X11 packages in the LOCALBASE tree, for which you must install the xpkgwedge package (`pkgsrc/pkgtools/xpkgwedge`), see *Program locations* for further details.

Some packages look in `/etc/mk.conf` to alter some configuration options at build time. Have a look at `/usr/pkgsrc/mk/mk.conf.example` to get an overview of what you can set there.

If you want to (re)install a binary package that you've created (see below) or that you put into `.../pkgsrc/packages` manually, you can use the the "bin-install" target, which will install a binary package - if available - via "`pkg_add`", and do a "`make package`" else.

Chapter 4. Making a precompiled package

Once you have built and installed the package as mentioned above, you can build it into a “binary package” - you might want to do this so that you can use the binaries you have just built on another NetBSD system, or to provide a simple means for others to use your binary package instead of wasting CPU time - this is done by changing to the appropriate directory in the `pkgsrc` tree, and typing the command

```
make package
```

at the shell prompt. This will build and install your package (if not already done), and then construct a binary package out of the results so that you can use the pkg_* tools to manipulate this. The binary package is stored under `/usr/pkgsrc/packages`, it’s in the form of a gzipped tar-file at the present time. See *Packaging top* for a continuation of the above top example.

Please see *Submitting* later in this document on submitting binary packages.

III. Package Constructor’s Guide

Chapter 5. Package components - files, directories and contents

Whenever you're preparing a package from the FreeBSD/OpenBSD ports collection or doing it from scratch, there are a number of files involved which are described in the following sections. Special directions are given for what differs from FreeBSD/OpenBSD ports for each file.

5.1. `Makefile`

Building, installation and creation of a binary package are all controlled by the package's Makefile.

There is a Makefile for each package. This file includes the standard `bsd.pkg.mk` file (referenced as `"../../mk/bsd.pkg.mk"`), which sets all the definitions and actions necessary for the package to compile and install itself. The mandatory fields are the `DISTNAME` which specifies the base name of the distribution file to be downloaded from the site on the Internet, `MASTER_SITES` which specifies that site, `CATEGORIES` which denotes the categories into which the package falls, `PKGNAME` which is the name of the package and the `MAINTAINER` name.

The `MASTER_SITES` may be set to one of the predefined sites:

- `${MASTER_SITE_XCONTRIB}`
- `${MASTER_SITE_GNU}`
- `${MASTER_SITE_PERL_CPAN}`
- `${MASTER_SITE_TEX_CTAN}`
- `${MASTER_SITE_SUNSITE}`

If one of these predefined sites is chosen, you may require the ability to specify a subdirectory of that site. Since these macros may expand to more than one actual site, you *must* use the following construct to specify a subdirectory:

```
${MASTER_SITE_GNU:=subdirectory/name/}
```

Note the trailing slash after the subdirectory name. Use of the deprecated `MASTER_SITE_SUBDIR` will not work.

Currently the following values are available for `CATEGORIES`. If more than one is used, they need to be separated by spaces:

```
archivers    corba        fonts        math         packages     sysutils
audio        cross        games        mbone        parallel     templates
benchmarks   databases    graphics     meta-pkgs    pkgtools     test
```

```
biology      devel        ham          misc         plan9        textproc
cad          distfiles    japanese     mk           print        www
comms        editors      lang         net          security     x11
```

See the NetBSD packages(7) manual page for a description of all available options and variables.

Please pay attention to the following gotchas, especially when preparing a package from the FreeBSD ports collection:

- Remove all `MAN`*x* and `CAT`*x* definitions from the package `Makefile` - NetBSD has implemented automatic manual page handling, and these definitions are obsolete.
- Add `MANCOMPRESSED` (if not already there) if manpages are installed in compressed form by the package; Packages that evaluate the `MANZ` variable on their own should set this[1]. See comment in `bsd.pkg.mk`.
- Replace `/usr/local` by `${PREFIX}` in all files (see below)
- Delete any "`ldconfig`" commands - this will be done automatically for you if the NetBSD platform supports it, and other measures will be taken on platforms which don't. (e.g. NetBSD/Alpha).
- If modifying a package from the FreeBSD ports collection, preserve their RCS ID: remove the '`$`'s around the FreeBSD RCS Id, and insert the word `FreeBSD`, then add a `$NetBSD$`, i.e.:

  before:

  ```
  # $Id: Makefile,v 1.17 1997/06/16 06:39:51 max Exp$
  ```

  after:

  ```
  # $NetBSD$
  # FreeBSD Id: Makefile,v 1.17 1997/06/16 06:39:51 max Exp
  ```

- If the package installs any info files, the main info directory file needs to be updated to reflect this fact. NetBSD now has an `INFO_FILES` definition, which is used to do this. For example, to install the indent.info entry into the info directory file, simply use the

  ```
  INFO_FILES= indent.info
  ```

  definition in the package `Makefile`. If the package does this insertion for you, you should specify `USE_GTEXINFO=1` in the package Makefile, to ensure that the pre-requisite GNU texinfo package is installed on your system.

- Adjust `MAINTAINER` to be either yourself. Do not leave the FreeBSD value, as it is unlikely that the FreeBSD people will care about NetBSD packages.
- If there exists a home page for the software in question, please add the variable `HOMEPAGE` right after `MAINTAINER`. The value of this variable should be the URL for the home page.

5.2. `files/*`

`files/md5`:

Most important, the mandatory md5 checksum of all the distfiles needed for the package to compile, confirming they match the original file any patches were generated against. This ensures that the distfile retrieved from the Internet has not been corrupted during transfer or altered by a malign force to introduce a security hole. It can be generated by hand using the md5(1) command or by invoking "`make makesum`".

`files/patch-sum`:

The checksum file for all the official patches for the package, found in the `patches/` directory (see *patches/**). This checksum file includes an MD5 checksum of all lines in the patch file except the NetBSD RCS Id. This file is generated by invoking "`make makepatchsum`".

Besides that, if you have any files that you wish to be placed in the package prior to configuration or building, you could place these files here and use a "`${CP}`" command in the pre-configure target to achieve this. Alternatively, you could simply diff the file against `/dev/null` and use the patch mechanism to manage the creation of this file.

5.3. `patches/*`

This directory contains files that are used by the patch(1) command to modify the sources as distributed in the distribution file into a form that will compile and run perfectly on NetBSD. The files are applied successively in alphabetic order (as returned by a shell "patches/patch-*" glob expansion), so `patch-aa` is applied before `patch-ab` etc.

The `patch-??` files should be in "`diff -bu`" format, and apply without a fuzz to avoid problems. (The latter condition is ensured by setting `PKG_DEVELOPER` in `/etc/mk.conf` - the build will fail if

a patch applies with fuzz only). Furthermore, do not put changes for more than one file into a single patch-file, as this will make future modifications and maintainance more difficult.

One important thing to mention is to pay attention that no RCS IDs get stored in the patch files, as these will cause problems when later checked into the NetBSD CVS tree. To avoid this, use the "`-U 2`" or "`-U 1`" option to diff. handle this.

If you don't want to worry about the problems in the last two paragraphs yourself, use "`pkgdiff`" from the `pkgsrc/pkgtools/pkgdiff` package, which takes care of any RCS IDs by itself.

For even more automation, we recommend using "`mkpatches`" from the same package to make a whole set of patches. You just have to backup files before you edit them to `filename.orig`, e.g. with "`cp -p filename filename.orig`". If you upgrade a package this way, you can easily compare the new set of patches with the previously existing one with the "`patchdiff`" command.

When preparing a FreeBSD port for the NetBSD packages system, it's likely that the FreeBSD port will work on NetBSD. However, check that the person who ported the software to FreeBSD has not played fast and loose with the __`FreeBSD`__ cpp definition without good cause - a simple way to do this is to do

```
grep -i freebsd patches/patch-??
```

in the package directory.

Besides taking care of any FreeBSDisms, be sure to provide patches to replace any occurrence of `/usr/local` in any `Makefile`s in the original package with `${PREFIX}`.

When you have finished a package, remember to generate the checksums for the patch files by using the "`make makepatchsum`" command, see *files/**.

5.4. pkg/*

This directory contains several files used to manage the creation of binary packages. Files from this directory are used in the binary package itself, and will thus be installed on other machines, so you should be aware that there is a wider audience than you might think for your comments and witticisms.

5.4.1. Mandatory files

`pkg/COMMENT`:

> A one-line description of the piece of software. There is no need to mention the package's name - this will automatically be added by the pkg_* tools when they are invoked.

`pkg/DESCR`:

A multi-line description of the piece of software. This should include any credits where they are due. Please bear in mind that others do not share your sense of humour (or spelling idiosyncrasies), and that others will read everything that you write here.

`pkg/PLIST`:

This file governs the files that are installed on your system: all the binaries, manual pages, etc. There are other directives which may be entered in this file, to control the creation and deletion of directories, and the location of inserted files.

If you're updating a FreeBSD package to work for NetBSD, please pay special attention to the following things in `pkg/PLIST`:

- If there are any "`@exec ldconfig ...`" statements, or any "`@unexec ldconfig ...`", delete them. NetBSD works out automatically whether to call "`ldconfig`", since some NetBSD architectures do not have that command.
- Add any missing "`@dirrm`" statements
- Remove any `MAN`*x* definitions in the package `Makefile`.

You could also investigate the port2pkg package (`pkgsrc/pkgtools/port2pkg`), which does a lot of the donkey work for you.

5.4.2. Optional files

`pkg/INSTALL`:

Shell script invoked twice during pkg_add. First time after package extraction and before files are moved in place, the second time after the files to install are moved in place. This can be used to do any custom procedures not possible with "`@exec`" commands in `PLIST`. See pkg_add(1) and pkg_create(1) for more information.

`pkg/DEINSTALL`:

This script is executed before and after any files are removed. It is this script's responsibility to clean up any additional messy details around the package's installation, since all pkg_delete knows is how to delete the files created in the original distribution. See pkg_delete(1) and pkg_create(1) for more information.

`pkg/REQ`:

Require-script that is invoked before installation and de-installation to ensure things like certain accounts being available, user/sysadmin agreeing with usage policy, etc.

`pkg/MESSAGE`:

Display this file after installation of the package. Useful for things like legal notices on almost-free software, setup instructions etc.

5.5. `scripts/*`

This directory contains any files that are necessary for configuration of your software, etc. If a script with any of the following names is present, it will be executed at the appropriate time during the build process:

```
pre-fetch          post-fetch
pre-extract        post-extract
pre-patch          post-patch
pre-configure      post-configure     configure
pre-build          post-build
pre-install        post-install
pre-package        post-package
```

Note that you should *not* define a pre-* or post-* target in the Makefile which executes the matching `scripts/{pre|post{-*` script. `bsd.pkg.mk` runs any existing `Makefile` target first, then searches for `scripts/*` and runs it using sh(1). Running the script from the `Makefile` would cause it to be run twice.

See *The build process* for a description of the build process.

5.6. `work/*`

When you type "`make`" the distribution files are unpacked into this directory. It can be removed by typing

```
make clean
```

at the shell prompt. Also, this directory is used to keep various timestamp files.

5.7. Importing the package into CVS

This section is mostly of interrest to persons with write access to the NetBSD CVS repository, you can ignore it if you use AnonCVS, SUP, ... to update your sources. Please see *Submitting* for how to submit a package instead.

Newly created packages should be imported with a vendor tag of `TNF` and a release tag of `pkgsrc-base`, e.g.:

```
cvs import pkgsrc/category/frobnitz TNF pkgsrc-base
```

Packages derived from a FreeBSD port should be imported with a vendor tag of `FREEBSD` and a release tag of `FreeBSD-current-`*`YYYY-MM-DD`* (*`YYYY-MM-DD`* being the date when the snapshot of the port were taken form the FreeBSD tree), and then doing the necessary modifications by normal CVS operations. E.g:

```
cvs import pkgsrc/category/mumbler FREEBSD FreeBSD-current-1998-04-01
cvs rm patches/patch-a
cvs add patches/patch-aa
cvs ci
```

Please note all package updates/additions in `doc/pkg-CHANGES`. It's very important to keep this file up to date and conforming to the existing format, because it will be used by scripts to automatically update pages on www.netbsd.org (http://www.netbsd.org/).

Notes

1. The `MANZ` variable can be set in `/etc/mk.conf` to install compressed manpages.

Chapter 6. PLIST* issues

This section addresses some special issues that one needs to pay attention to when dealing with the PLIST file (or files, see below).

6.1. Miscellaneous

NetBSD RCS Id:

Be sure to add a RCS ID line as the first thing in any PLIST file you write:

```
@comment $NetBSD$
```

"ranlib":

Don't put any "ranlib" commands into your PLIST files, as they will cause troubles when the package is removed. Just make sure the build-process does run "ranlib" - it usually does - and you can leave this out. This is usually only a problem when using ports from FreeBSD.

"ldconfig":

Don't put any "ldconfig" commands into your PLIST files, as they will cause problems. All shared object caching is done automatically in NetBSD (this takes place when you see the Automatic shared object handling message), and so you can leave this out. If any shared objects are found in the package, they will be dealt with automatically, running "ldconfig" on platforms that need it. This is usually only a problem when using ports from FreeBSD.

${MACHINE_ARCH}, ${MACHINE_GNU_ARCH}:

Some packages like emacs and perl embed information about which architecture they were built on into the pathnames where they install their file. To handle this case, PLIST will be pre-processed before actually used, and the symbol "${MACHINE_ARCH}" will be replaced by what "sysctl -n hw.machine_arch" gives. The same is done if the string "${MACHINE_GNU_ARCH}" is embedded in PLIST somewhere - use this on packages that use GNU autoconfigure. There are a few more variables that are expanded, please see the PLIST_SUBST variable in bsd.pkg.mk.

Legacy note: There used to be a symbol "<$ARCH>" that was replaced by the output of "uname -m", but that's no longer supported and has been removed.

${OPSYS}, ${OS_VERSION}:

Some packages want to embed the OS name and version into some paths. to do this, use these two variables in PLIST. "${OPSYS}" will be replaced by output from "uname -s", "${OS_VERSION}" will be set to what "uname -r" gives.

Manpage-compression:

The package should install manpages in compressed form if MANZ is set (in /etc/mk.conf), and uncompressed otherwise. To handle this in the PLIST file, the suffix ".gz" is appended/removed automatically for manpages according to MANZ and MANCOMPRESSED being set or not, see above for details. This modification of the PLIST file is done on a copy of it, not pkg/PLIST itself.

Semi-automatic PLIST generation:

You can use the "make print-PLIST" command to output a PLIST that matches any new files since the package was extracted. If the package installs files via tar(1) or other methods that don't update file access times, be sure to add these files manually to your pkg/PLIST!

6.2. MD/MI vs. general PLIST

Sometimes the packaging list in pkg/PLIST differs between platforms, e.g. if one of them supports shared libs and the other does not. To address this, a hook has been introduced into the NetBSD packages system to provide a PLIST file defined on conditions set freely in the package's Makefile.

6.2.1. PLIST_SRC

To use one or more files as source for the PLIST used in generating the binary package, set the variable PLIST_SRC to the names of that file(s). The files are later concatenated using cat(1), and order of things is an important issue, see below.

6.2.2. PLIST-mi, PLIST-md.shared, PLIST-md.static

If PLIST_SRC is not set (the usual case), and if there is no pkg/PLIST, the packages system looks for pkg/PLIST-mi, and pkg/PLIST-md.shared or pkg/PLIST-md.static to handle differences due to the platform being able to handle shared libs or not. PLIST-mi contains machine independent files, PLIST-md.* contain machine dependent files, which may differ between architectures that don't support dynamic libs/shared loading.

Currently, this is only used in the perl-packages, and as perl5 on alpha doesn't support dynamic loading of extensions like perl/Tk yet, PLIST.mi-static is also used on the alpha (besides pmax and powerpc). Alpha will hopefully be removed soon when perl's fixed for dynamic loading.

(This handling of MI/MD PLIST files is implemented by setting PLIST_SRC to either "PLIST-mi PLIST-md.static" or "PLIST-mi PLIST-md.shared", see /usr/pkgsrc/mk/bsd.pkg.mk).

6.2.3. Order in the PLIST* file(s)

There is one gotcha regarding the ordering of "@dirrm" statements: any MI "@dirrm" directives that follow any MD "@dirrm"s *must* go into the PLIST.md-* files, as the files PLIST-mi and PLIST.md-{shared/static} are concatenated in exactly this order. If the MI directory would be listed in PLIST-mi, it would be removed before the MD directory, which wouldn't work.

E.g. if you have the following dirs:

```
foo/mi
foo/mi/md
```

then PLIST-mi contains:

```
nothing
```

and PLIST-md.* contain:

```
@dirrm foo/mi/md
@dirrm foo/mi
```

This will lead to some "@dirrm" statements being duplicated, but it's the only way to ensure everything is properly removed. The same care must be taken when PLIST_SRC is set to some package-specific settings.

Chapter 7. Notes on fixes for packages

7.1. CPP defines

To port an application to NetBSD, it's usually necessary for the compiler to be able to judge the system on which it's compiling, and we use definitions so that the C pre-processor can do this.

The really impatient should just note that a number of the FreeBSD ports (which are called packages in the NetBSD world) rely on the CPP definition `__FreeBSD__`. This should be used sparingly, for FreeBSD-specific features, but unfortunately this is not always the case. A number also rely on the fact that the CPU type is an Intel-based CPU with little-endian byte order.

To test whether you are working on a 4.4 BSD-derived system, you should use the BSD definition, which is defined in `<sys/param.h>` on said systems.

```
#include <sys/param.h>
```

and then you can surround the BSD-specific parts of your port using the conditional:

```
#if (defined(BSD) && BSD >= 199306)
...
#endif
```

Please use the `__NetBSD__` definition sparingly - it should only apply to features of NetBSD that are not present in other 4.4-lite derived BSDs.

You should also avoid defining `__FreeBSD__=1` and then simply using the FreeBSD port, if only from an aesthetic viewpoint.

7.2. Shared libraries - libtool

NetBSD supports many different machines, with different object formats like a.out and ELF, and varying abilities to do shared library and dynamic loading at all. To accompany this, varying commands and options have to be passed to the compiler, linker etc. to get the Right Thing, which can be pretty annoying especially if you don't have all the machines at your hand to test things. The libtool package (pkgsrc/devel/libtool) can help here, as it just "knows" how to build both static and dynamic libraries from a set our source files, thus being platform independent.

Here's how to use libtool in a package in six simple steps:

1. Add `USE_LIBTOOL=yes` to the package Makefile.

2. For library objects, use "`${LIBTOOL} -mode=compile ${CC}`" in place of "`${CC}`". You could even add it to the definition of `CC`, if only libraries are being built in a given Makefile. This one command will build both PIC and non-PIC library objects, so you need not have separate shared and non-shared library rules.

3. For the linking of the library, remove any "`ar`", "`ranlib`", and "`ld -Bshareable`" commands, and use instead:

   ```
   ${LIBTOOL} -mode=link cc -o ${.TARGET:.a=.la} ${OBJS:.o=.lo} -rpath ${PRE-
   FIX}/lib -version-info major:minor
   ```

   Note that the library is changed to have a `.la` extension, and the objects are changed to have a `.lo` extension. Change the `OBJS` variable as necessary. This automatically creates all of the `.a`, `.so.major.minor`, and ELF symlinks (if necessary) in the build directory.

4. When linking programs that depend on these libraries *before* they are installed, preface the "`cc`" or "`ld`" line with "`${LIBTOOL} -mode=link`", and it will find the correct libraries (static or shared), but please be aware that libtool will not allow you to specify a relative path in `-L` (such as `-L../somelib`), because it is trying to force you to change that argument to be the `.la` file. For example

   ```
   ${LIBTOOL} -mode=link ${CC} -o someprog -L../somelib -lsomelib
   ```

   won't work; it needs to be changed to:

   ```
   ${LIBTOOL} -mode=link ${CC} -o someprog ../somelib/somelib.la
   ```

   and it will DTRT with the libraries. If you *must* use a relative path with `-L`, and you are not going to run this program before installing it, you can omit the use of libtool during link and install of this program if you add the subdirectory `.libs` in your "`-L`" command:

   ```
   ${CC} -o someprog -L../somelib/.libs -lsomelib
   ```

5. When installing libraries, preface the "`install`" or "`cp`" command with "`${LIBTOOL} -mode=install`", and change the library name to `.la`. For example:

   ```
   ${LIBTOOL} -mode=install ${BSD_INSTALL_DATA} ${SOMELIB:.a=.la} ${PREFIX}/lib
   ```

   This will install the static `.a`, shared library, any needed symlinks, and run "`ldconfig`".

6. In your `PLIST`, include the `.a`, `.la`, and `.so.major.minor` files. Don't include the ELF symlink files (`.so.major`, `.so`); those are added automatically.

Do *not* use pkglibtool! Previously, the package system used its own version of libtool from pkgtools. However, over time, this version became outdated and is now deprecated. You may see some definitions of `USE_PKGLIBTOOL` in existing packages that still use this outdated version of libtool. Please do not use this definition in new packages!

7.3. Using libtool on GNU packages that already support libtool

Add `USE_LIBTOOL=yes` and `LTCONFIG_OVERRIDE=${WRKSRC}/ltconfig` to the package `Makefile` as the quick way to bypass the package's own libtool. The package's own libtool is made by the "`ltconfig`" script at "`do-configure`" time. If `USE_LIBTOOL` and `LTCONFIG_OVERRIDE` are defined, the specified `ltconfig` is overridden, using the "`devel/libtool`" command instead of the package's own libtool. If the pkg already has an original "`libtool`" which we can replace with the pkgsrc/devel/libtool you may have to specify `LIBTOOL_OVERRIDE` to the package `Makefile`.

7.4. Gotchas of FreeBSD ports

See the *Makefile* section for Makefile issues (`MAN`*x*, `CAT`*x*, `MANCOMPRESSED`, ldconfig, RCS IDs) and *patches/** for gotchas on using patches from FreeBSD ports.

One of the biggest problems with FreeBSD ports is that too many of them assume they will install into `/usr/local`, instead of honouring any `${PREFIX}` setting properly. To change this, add something like the following into your package Makefile:

```
pre-configure:
        for f in ‘find ${WRKDIR} -type f -print \
          |xargs grep -l ’/usr/local’’; do \
                ${SED} -e ’s:/usr/local:’${PREFIX}’:g’ < $$f > $$f.pdone \
&& ${MV} $$f.pdone $$f; \
        done
```

This is taken from the sysutils/rtty package; be sure this works for your package - it may actually make sense to look for some things in `/usr/local`, for example. So don't blindly replace all occurrences of `/usr/local`!

FreeBSD has decided to list manual pages in the package `Makefile`, with no corresponding entry in the `PLIST` file. You will thus need to add any `MAN[1-8ln]` files to the `PLIST` file before deleting the `MAN[1-8ln]` definition. Similarly with `MLINKS` and `CAT[1-8ln]` entries.

Side note on manpages in PLIST: we don't take any notice of any `.gz` suffix there, as many FreeBSD ports seem to have `.gz` pages in PLIST even when they install manpages without compressing them;

rather, we add our own `.gz` suffix there according to `MANZ`. In short, it does not matter whether the manual page name in the `PLIST` file has a `.gz` suffix or not - if it needs one which is not already there, it will be appended automatically, and if there is a `.gz` suffix which is not needed, it will be deleted automatically.

Some packages use bsd-style `.mk` files when building, and so any manual pages that are installed will be gzip-compressed, if `MANZ` is set, or not if `MANZ` is not set. If the package uses bsd-style `.mk` files, the variable `MANCOMPRESSED_IF_MANZ` should be set to a value of `yes` in the package `Makefile`.

7.5. Feedback to the author

If you have found any bugs in the package you make available, if you had to do special steps to make it run under NetBSD or if you enhanced the software in various other ways, be sure to report these changes back to the original author of the program! With that kind of support, the next release of the program can incorporate these fixes, and people not using the NetBSD packages system can win from your efforts.

Support the idea of free software!

Chapter 8. The build process

The basic steps for building a program are always the same. First the program's source (distfile) must be brought to the local system and then extracted. After any patches to compile properly on NetBSD are applied, the software can be configured, then built (usually by compiling), and finally the generated binaries etc. can be put into place on the system. These are exactly the steps performed by the NetBSD package system, which is implemented as a series of targets in a central Makefile, `/usr/pkgsrc/mk/bsd.pkg.mk`.

8.1. Program locations

Before outlining the process performed by the NetBSD package system in the next section, here's a brief discussion on where programs are installed, and which variables influence this.

The automatic variable `PREFIX` indicates where all files of the final program shall be installed. It is usually set to `$LOCALBASE` (`/usr/pkg`), or `$CROSSBASE` for pkgs in the `cross` category, though its value becomes that of `$X11BASE` if `USE_IMAKE`, `USE_MOTIF`, or `USE_X11BASE` is set. The value `${PREFIX}` needs to be put into the various places in the program's source where paths to these files are encoded; see sections *patches/** and *Shared libraries - libtool* for details on this.

When choosing which of these variables to use, follow the following rules:

- `${PREFIX}` always points to the location where the current package will be installed. When referring to a package's own installation path, use `${PREFIX}`.
- `${LOCALBASE}` is where all non-X11 pkgs are installed. If you need to construct a `-I` or `-L` argument to the compiler to find includes and libraries installed by another non-X11 pkg, use `${LOCALBASE}`.
- `${X11BASE}` is where the actual X11 distribution is installed. When looking for *standard* X11 includes (not those installed by a pkg), use `${X11BASE}`.
- X11 based pkgs are special in that they may be installed in either `X11BASE` or `LOCALBASE`. To install X11 packages in `LOCALBASE`, simply install the xpkgwedge package (`pkgsrc/pkgtools/xpkgwedge`). If you need to find includes or libraries installed by a pkg that has `USE_IMAKE`, `USE_MOTIF`, or `USE_X11BASE` in its pkg `Makefile`, you need to use *both* `${X11BASE}` and `${LOCALBASE}`.
- `${X11BASE}` points to the root of the installed X11 tree. To refer to the installed location of an X11 package, use the `${X11PREFIX}` definition (this will be `${LOCALBASE}` if xpkgwedge is installed, and `${X11BASE}` if not).

8.2. Main targets

The main targets used during the build process defined in `bsd.pkg.mk` are:

fetch:

This will check if the file(s) given in the variables `DISTFILES` and `PATCHFILES` (as defined in the package's `Makefile`) are present on the local system in `/usr/pkgsrc/distfiles`. If they are not present, they will be fetched using ftp(1) from the site(s) given in the variable `PATCH_SITES`. The location(s) in `PATCH_SITES` are in the form of URLs and can be `ftp://`- and `http://`-URLs, as ftp(1) understands both of them.

checksum:

After the distfile(s) are fetched, their MD5 checksum is generated and compared with the checksums stored in the `files/md5` file. If the checksums don't match, the build is aborted. This is to ensure the same distfile is used for building, and that the distfile wasn't changed, e.g. by some malign force, deliberately changed distfiles on the master distribution site or network lossage.

extract:

When the distfiles are present on the local system, they need to be extracted, as they are usually in the form of some compressed archive format, most commonly `.tar.gz`. If only some of the distfiles need to be uncompressed, the files to be uncompressed should be put into `EXTRACT_ONLY`. If the distfiles are not in `.tar.gz` format, they can be extracted by setting `EXTRACT_CMD`, `EXTRACT_BEFORE_ARGS` and/or `EXTRACT_AFTER_ARGS`.

patch:

After extraction, all the patches named by the `PATCHFILES` and those present in the `patches` subdirectory of the package are applied. Patchfiles ending in `.Z` or `.gz` are uncompressed before they are applied, files ending in `.orig` or `.rej` are ignored. Any special options to patch(1) can be handed in `PATCH_DIST_ARGS`. See *patches/** for more details.

If the variable `PKG_DEVELOPER` is set in `/etc/mk.conf`, patch is given special args to make it fail if the patches with some lines of fuzz. Please fix (regenerate) the patches so that they apply cleanly. The rationale behind this is that patches that apply cleanly may end up being applied in the wrong place, and cause severe harm there.

configure:

Most pieces of software need information on the header files, system calls, and library routines which are available in NetBSD. This is the process known as configuration, and is usually automated. In most cases, a script is supplied with the source, and its invocation results in generation of header files, Makefiles, etc.

If the program doesn't come with its own configure script, one can be placed in the package's `scripts` directory, called `configure`. If so, it is executed using sh(1).

If the program's distfile contains its own configure script, this can be invoked by setting `HAS_CONFIGURE`. If the configure script is a GNU autoconf script, `GNU_CONFIGURE` should be specified instead. In either case, any arguments to the configure script can be specified in the `CONFIGURE_ARGS` variable, and the configure script's name can be set in `CONFIGURE_SCRIPT` if it differs from the default `configure`.

If the program uses an `Imakefile` for configuration, the appropriate steps can be invoked by setting `USE_IMAKE` to `yes`. (If you only want the package installed in `$X11PREFIX` but xmkmf not being run, set `USE_X11BASE` instead!)

build:

Once configuration has taken place, the software can be built on NetBSD by invoking $MAKE_PROGRAM on `$MAKEFILE` with $ALL_TARGET as the target to build. The default MAKE_PROGRAM is "`gmake`" if `USE_GMAKE` is set, "`make`" otherwise. `MAKEFILE` is set to `Makefile` by default, and `ALL_TARGET` defaults to `all`. Any of these variables can be set to change the default build process.

install:

Once the build stage has completed, the final step is to install the software in public directories, for users. As in the build-target, $MAKE_PROGRAM is invoked on `$MAKEFILE` here, but with the $INSTALL_TARGET instead, the latter defaulting to "install" (plus "install.man", if `USE_IMAKE` is set).

If no target is specified, the default is "build". If a subsequent stage is requested, all prior stages are made: e.g. "make build" will perform the equivalent of:

```
make fetch
make checksum
make extract
make patch
make configure
make build
```

8.3. Other helpful targets

pre/post-*:

For any of the main targets described in the previous section, two auxiliary targets exist with "pre-" and "post-" used as a prefix for the main target's name. These targets are invoked before and after the main target is called, allowing extra configuration or installation steps, for example, which program's configure script or install target omitted. For any of these auxiliary targets, scripts of the same name can be placed in the package's `scripts`-subdirectory that will be executed at the given time, see *scripts/**.

do-*:

Should one of the main targets do the wrong thing, and should there be no variable to fix this, you can redefine it with the do-* target. (Note that redefining the target itself instead of the do-* target is a bad idea, as the pre-* and post-* targets won't be called anymore, etc.) You will not usually need to do this.

reinstall:

If you did a "`make install`" and you noticed some file was not installed properly, you can repeat the installation with this target, which will ignore the "already installed" flag.

deinstall:

This target does a pkg_delete(1) in the current directory, effectively de-installing the package. The following variables can be used either on the command line or in `/etc/mk.conf` to tune the behaviour:

`PKG_VERBOSE`:

Add a "-v" flag to the pkg_delete(1) command.

`DEINSTALLDEPENDS`:

Remove all packages that require (depend on) the given package. This can be used to remove any packages that may have been pulled in by a given package, e.g. if `make deinstall DEINSTALLDEPENDS=1` is done in `x11/kde`, this is likely to remove whole KDE. Works by adding a "-R" to the pkg_delete command line.

update:

This target causes the current package to be updated to the latest version. The package and all depending packages first get deinstalled, then current versions of the corresponding packages get compiled and installed. This is similar to manually noting which packages are currently installed, then performing a series of “`make deinstall`” and “`make install`” for these packages.

You can use the "update" target to resume package updating in case a previous “`make update`” was interrupted for some reason. However, in this case, make sure you don’t call “`make clean`” or otherwise remove the list of dependent packages in `${WRKDIR}`. Otherwise you lose the ability to automatically update the current package along with the dependent packages you have installed.

Resuming an interrupted “`make update`” will only work as long as the package tree remains unchanged. If the source code for one of the packages to be updated has been changed, resuming “`make update`” will most certainly fail!

The following variables can be used either on the command line or in `/etc/mk.conf` to alter the behaviour of “`make update`”:

`DEPENDS_TARGET`:

Install target to use for the updated package and the dependent packages. Defaults to `install`. E.g. “`make update DEPENDS_TARGET=package`”

`NOCLEAN`:

Don’t clean up after updating. Useful if you want to leave the work sources of the updated packages around for inspection or other purposes. Be sure you eventually clean up the source tree (see the "clean-update" target below) or you may run into troubles with old source code still lying around on your next “`make`” or `make update`.

`REINSTALL`:

Use "reinstall" instead of ${DEPENDS_TARGET} for every package that gets updated. Be sure you know the implications of using the "reinstall" target when using this variable.

clean-update:

Clean the source tree for all packages that would get updated if “`make update`” was called from the current directory. This target should not be used if the current package (or any of its depending packages) have already been deinstalled (e.g., after calling “`make update`”) or you may lose some packages you intended to update. As a rule of thumb: only use this target *before* the first time you call “`make update`” and only if you have a dirty package tree (e.g., if you used `NOCLEAN`). The following variables can be used either on the command line or in `/etc/mk.conf` to alter the behaviour of “`make clean-update`”:

`CLEAR_DIRLIST`:

After "`make clean`", do not reconstruct the list of directories to update for this package. Only use this if "`make update`" successfully installed all packages you wanted to update. Normally, this is done automatically on "`make update`", but may have been suppressed by the `NOCLEAN` variable (see above).

readme:

This target generates a `README.html` file, which can be viewed using a browser such as netscape (`pkgsrc/www/mozilla`) or lynx (`pkgsrc/www/lynx`). The generated files contain references to any packages which are in the `${PACKAGES}` directory on the local host. The generated files can be made to refer to URLs based on `FTP_PKG_URL_HOST` and `FTP_PKG_URL_DIR`. (For example, if I wanted to generate `README.html` files which pointed to binary packages on the local machine, in the directory `/usr/packages`, set `FTP_PKG_URL_HOST=file://localhost` and `FTP_PKG_URL_DIR=/usr/packages`. The `${PACKAGES}` directory and its subdirectories will be searched for all the binary packages.)

readme-all:

Use this target to create a file `README-all.html` which contains a list of all packages currently available in the NetBSD Packages Collection, together with the category they belong to and a short description. This file is compiled from the `pkgsrc/*/README.html` files, so be sure to run this *after* a "`make readme`".

cdrom-readme:

This is very much the same as the "readme" target (see above), but is to be used when generating a pkgsrc tree to be written to a CD-ROM. This target also produces `README.html` files, and can be made to refer to URLs based on `CDROM_PKG_URL_HOST` and `CDROM_PKG_URL_DIR`.

show-distfiles:

This target shows which distfiles and patchfiles are needed to build the package. (`DISTFILES` and `PATCHFILES`, but not `patches/*`)

show-downlevel:

This target shows nothing if the package is not installed. If a version of this package is installed, but is not the version provided in this version of pkgsrc, then a warning message is displayed. This target can be used to show which of your installed packages are downlevel, and so the old versions can be deleted, and the current ones added.

show-pkgsrc-dir:

This target shows the directory in the pkgsrc hierarchy from which the package can be built and installed. This may not be the same directory as the one from which the package was installed. This target is intended to be used by people who may wish to upgrade many packages on a single host, and can be invoked from the top-level pkgsrc Makefile by using the target "show-host-specific-pkgs".

check-shlibs:

After a package is installed, check all it’s binaries and (on ELF platforms) shared libraries if they find the shared libs they need. Run by default if `PKG_DEVELOPER` is set in `/etc/mk.conf`.

Chapter 9. Debugging

To check out all the gotchas when building a package (either from a FreeBSD port, or from scratch), here are the steps that I do in order to get a package working. Please note this is basically the same as what was explained in the previous sections, only with some debugging aids.

1. Make sure `PKG_DEVELOPER=1` is in `/etc/mk.conf`
2. Retrieve port from FreeBSD collection
3. Fix RCS-ID in the package's `Makefile`, see *Makefile*.
4. Import unchanged FreeBSD source (only if you have CVS write access, not needed otherwise):

```
    (cd .../pkgsrc/category/pkgname ; cvs import pkgsrc/category/pkgname \
  FREEBSD FreeBSD-current-yyyy-mm-dd)
```

5. If you did a CVS import, check it out to apply the following fixes (not needed if you don't have CVS access!)
6. Look at `Makefile`, fix if necessary; see *Makefile*.
7. Look at patches, remember if not appropriate
8. Have a look at `pkg/PLIST`, add a "`@comment $NetBSD$`" line at the beginning of any `PLIST` file (see *PLIST* issues*).
9. make
10. If something is not ok, fix; for patches: fix the file, then re-generate the diff: "`diff -bu foo.orig foo >../../patches/patch-xx`" ("`mv patch-xx patch-xx.orig`" before); If there's no `foo.orig` from a previous patch, be sure to have an old version of the file somewhere; re-iterate :)
11. If all builds OK: "`touch /tmp/bla`"
12. make install
13. "`find /usr/pkg/ /usr/X11R6/ -newer /tmp/bla >/tmp/x`" (or whatever you set `LOCALBASE` and `X11BASE` to)
14. "`pkg_delete blub`"
15. "`find /usr/pkg/ /usr/X11R6/ -newer /tmp/bla`" (or diff against output of "`make print-PLIST`"): if this brings up any files, that are missing in `pkg/PLIST*`; add them.
16. Compare `pkg/PLIST*` against `/tmp/x`, fix the former one ("`sort /tmp/x >/tmp/x2 ; sort pkg/PLIST >/tmp/P ; sdiff /tmp/x2 /tmp/P`")
17. "`make reinstall && make package`"
18. "`pkg_delete blub`"
19. "`find /usr/pkg/ /usr/X11R6/ -type f -newer /tmp/bla`" shouldn't find anything now

20. “`pkg_add .../blub.tgz`”
21. Play with it :)
22. “`pkg_delete`” - still no file should be left (re-run above “`find`”)
23. “`make clean && touch /tmp/bla && make install && make clean && make deinstall`” then run the “`find`” again. Yes, some software authors write Makefiles that install files during the build target. Sigh. Re-run the “`find`”, and fix the `PLIST`. Repeat until certain the software does not install any files that aren’t in `PLIST`.
24. submit (or commit, if you have CVS access); see *Submitting*.

Chapter 10. FAQs & features of the package system

10.1. Packages using GNU autoconfig

If your package uses GNU autoconf, add the following to your package's Makefile:

```
GNU_CONFIGURE= yes
```

Note that this appends `-prefix=${PREFIX}` to `CONFIGURE_ARGS`, so you don't have to do that yourself, and this may not be what you want.

10.2. Other distrib methods than `.tar.gz`

If your package uses a different distribution method from `.tar.gz`, take a look at the package for `plan9/sam`, which uses a gzipped shell archive (shar), but the quick solution is to set `EXTRACT_SUFX` to the name after the `DISTNAME` field, and add the following to your package's `Makefile`:

```
EXTRACT_SUFX=    .msg.gz
EXTRACT_CMD=             zcat
EXTRACT_BEFORE_ARGS=
EXTRACT_AFTER_ARGS=      |sh
```

10.3. Packages not creating their own subdirectory

Your package doesn't create a subdirectory for itself (like GNU software does, for instance), but extracts itself in the current directory: see `plan9/sam` again, but the quick answer is:

```
NO_WRKSUBDIR=    yes
```

10.4. Custom configuration process

If your package uses a weird Configure script see the `top` package, but the quick answer is:

```
HAS_CONFIGURE=          yes
CONFIGURE_SCRIPT=       Configure
CONFIGURE_ARGS+=        netbsd13
```

10.5. Packages not building in their `DISTNAME` directory

If your package builds in a different directory from its base `DISTNAME`, see the `tcl80` and `tk80` packages:

```
WRKSRC=         ${WRKDIR}/${DISTNAME}/unix
```

10.6. How to fetch all distfiles at once

You would like to download all the distfiles in a single batch from work or school, where you can't run a "`make fetch`". But there's no archive of the distfiles on ftp.netbsd.org and the one on ftp.freebsd.org contains many distfiles for which there are no ports (yet).

The answer here is to do a "`make fetch-list`" in `/usr/pkgsrc` and use the resulting list.

10.7. How to fetch files from behind a firewall

If you are sitting behind a firewall which does not allow direct connections to Internet hosts (i.e. non-NAT), you may specify the relevant proxy hosts. This is done using an environment variable in the form of a URL e.g. if the machine www-proxy.myisp.com is one of the firewalls, and it uses port 80 as the proxy port number, the proxy environment variables look like:

```
ftp_proxy=ftp://www-proxy.myisp.com:80/
http_proxy=http://www-proxy.myisp.com:80/
```

10.8. If your patch contains an RCS ID

See *patches/** on how to remove RCS IDs from patch files.

10.9. How to pull in variables from `/etc/mk.conf`

The problem with package-defined variables that can be overridden via `MAKECONF` or `/etc/mk`.conf is that make(1) expands a variable as it is used, but evaluates preprocessor like statements (`.if`, `.ifdef` and `.ifndef`) as they are read. So, to use any variable (which may be set in `/etc/mk.conf`) in one of the `.if*` statements, the file `/etc/mk.conf` must be included before that `.if*` statement.

Rather than have a number of ad-hoc ways of including `/etc/mk.conf`, should it exist, or `MAKECONF`, should it exist, include the `pkgsrc/mk/bsd.prefs.mk` file in the package `Makefile` before any preprocessor-like `.if`, `.ifdef`, or `.ifndef` statements:

```
.include "../../mk/bsd.prefs.mk"

.if defined(USE_MENUS)
...
.endif
```

10.10. Is there a mailing list for pkg-related discussion?

Yes. We are using <`tech-pkg@netbsd.org`> for discussing package related issues. To subscribe do:

```
echo subscribe tech-pkg | mail majordomo@netbsd.org
```

10.11. How do I tell "`make fetch`" to do passive FTP?

This depends on which utility is used to retrieve distfiles. From `bsd.pkg.mk`, `FETCH_CMD` is assigned the first available command from the following list:

```
/usr/bin/fetch
${LOCALBASE}/bsd/bin/ftp
```

```
/usr/bin/ftp
```

On a default NetBSD install, this will be `/usr/bin/ftp`, which automatically tries passive connections first, and falls back to active connections if the server refuses to do passive. For the other tools, add the following to your `/etc/mk.conf` file:

```
PASSIVE_FETCH=1
```

Having that option present will prevent `/usr/bin/ftp` from falling back to active transfers.

10.12. Dependencies on other packages

Your package may depend on some other package being present - and there are various ways of expressing this dependency. NetBSD supports the `BUILD_DEPENDS` and `DEPENDS` definitions (beware: the `DEPENDS` definition is not the same as FreeBSD's deprecated one, and NetBSD does not use the FreeBSD `LIB_DEPENDS` definition any more - it proved problematic on ELF NetBSD platforms).

In the following examples, the `BUILD_DEPENDS` dependencies have the format: *`file`*:*`directory`*[:*`stage`*]. If the *`stage`* isn't specified, it defaults to "install". If the file contains a *'/'*, it is interpreted as a regular file - otherwise, the name is taken to be an executable file, and the shell's search `PATH` is searched for *`file`*. If the regular file is not found, or the executable file is not in the path, then the pre-requisite package will be built from the sources in *`directory`*, which is usually relative to the current package's directory. The `DEPENDS` definition specifies a package name (which contains its version number), and the directory containing the package to build if this version of the package is not installed.

- If your package needs files from another package to build, see the `print/ghostscript5` package (it relies on the jpeg sources being present in source form during the build):

  ```
  BUILD_DEPENDS+=  ../../graphics/jpeg/${WRKDIR:T}/jpeg-6a:../../graphics/jpeg:extract
  ```

- If your package needs to have another package installed to build itself, this is specified using the `BUILD_DEPENDS` definition, but without specifying the stage "`:extract`" as above. An example is the `print/lyx` package, which uses the "`latex`" binary during its build process:

  ```
  BUILD_DEPENDS+=  latex:../../print/teTeX
  ```

- If your package needs a library with which to link, this is specified using the `DEPENDS` definition. An example of this is the `print/lyx` package, which uses the xpm library, version 3.4j to build.

```
DEPENDS+=       xpm-3.4j:../../graphics/xpm
```

You can also use wildcards in package dependences:

```
DEPENDS+= xpm-*:../../graphics/xpm
```

Note that such wildcard dependencies are retained when creating binary package. The dependency is checked when installing the binary package and any package which matches the pattern would be used. Beware that wildard dependencies should be used with a bit of care. Simple example for package which needs some version of Tk installed, but doesn't care which exactly - dependency

```
DEPENDS+= tk-*:../../x11/tk80
```

would also match e.g. tk-postgresql-6.5.3, which is not what was needed. ALWAYS ensure that the wildcard doesn't match more than it should. For this example, use:

```
DEPENDS+= tk-[0-9]*:../../x11/tk80
```

This is safe because package names don't include digits.

- If your package needs some executable to be able to run correctly, this is specified using the `DEPENDS` definition. The `print/lyx` package needs to be able to execute the latex binary from the teTex package when it runs, and that is specified:

  ```
  DEPENDS+=        teTex-*:../../print/teTeX
  ```

  The comment about wildcard dependencies from previous paragraph applies here, too.

10.13. Conflicts with other packages

Your package may conflict with other packages a user might already have installed on his system, e.g. if your package installs the same set of files like another package in our pkgsrc tree.

In this case you can set `CONFLICTS` to a space separated list of packages (including version string) your package conflicts with.

For example `pkgsrc/x11/Xaw3d` and `pkgsrc/x11/Xaw-Xpm` install provide the same shared library, thus you set in `pkgsrc/x11/Xaw3d/Makefile`:

```
CONFLICTS=      Xaw-Xpm-*
```

and in pkgsrc/x11/Xaw-Xpm/Makefile:

```
CONFLICTS=      Xaw3d-*
```

Packages will automatically conflict with other packages with the name prefix and a different version string. "Xaw3d-1.5" e.g. will automatically conflict with the older version "Xaw3d-1.3".

10.14. Software which has a WWW Home Page

The NetBSD packages system supports a variable called HOMEPAGE. If the software being packaged has a home page, the Makefile should include the URL for that page in the HOMEPAGE variable:

```
HOMEPAGE= http://www.netpedia.net/hosting/gqview/mpeg-index.html
```

The definition of the package should be placed immediately after the MAINTAINER variable.

10.15. How to handle modified distfiles with the 'old' name

Sometimes authors of a software package make some modifications after the software was released, and they put up a new distfile without changing the package's version number. If a package is already in pkgsrc at that time, the md5 checksum will no longer match. The correct way to work around this is to update the package's md5 checksum to match the package on the master site (beware, any mirrors may not be up to date yet!), and to remove the old distfile from ftp.netbsd.org's /pub/NetBSD/packages/distfiles directory. Furthermore, a mail to the package's author seems appropriate making sure the distfile was really updated on purpose, and that no trojan horse or so crept in.

10.16. What does `Don't know how to make /usr/share/tmac/tmac.andoc` mean?

When compiling the pkgsrc/pkgtools/pkg_install package, you get the error from make that it doesn't know how to make /usr/share/tmac/tmac.andoc? This indicates that you don't have

installed the "text" set on your machine (nroff, ...). Please do so:

```
tar -unlink -pvx -C / -f .../text.tgz
```

10.17. How to handle incrementing versions when fixing an existing package

When making fixes to an existing package it can be useful to change the version number in `PKGNAME`. To avoid conflicting with future versions by the original author, use a '`nb1`' suffix (later versions should increment this to give '`nb2`' and so on).

10.18. `Could not find bsd.own.mk` - what's wrong?

You didn't install the compiler set, `comp.tgz`, when you installed your NetBSD machine. Please get it and install it, by extracting it in /:

```
tar -unlink -pvx -C / -f .../comp.tgz
```

`comp.tgz` is part of every NetBSD release, please get the one matching the release you have installed (determine via "`uname -r`").

Chapter 11. Submitting

11.1. Precompiled binary packages

Our policy is that we accept binaries only from NetBSD developers to guarantee that the packages don’t contain any trojan horses etc. This is not to annoy anyone but rather to protect our users! You’re still free to put up your home-made binary packages and tell the world where to get them.

11.2. packages

First, check that your package is complete, compiles and runs well; see *Debugging* and the rest of this document. Then, generate a gzipped `tar`-file of all the files needed for the package, preferably with all files in a single directory. Place this `tar`-file to a place where the package maintainers can fetch it using FTP or HTTP (WWW). Finally, “`send-pr`” with category "`pkg`", a synopsis which includes the package name and version number, a short description of your package (contents of `pkg/COMMENT` are OK) and the URL of your `tar`-file.

You will be notified if your send-pr has been addressed so you can remove the `tar`-file.

Chapter 12. A simple example of a package: bison

I checked to find a piece of software that isn't in the FreeBSD ports collection, and picked GNU bison. Quite why someone would want to have bison when Berkeley yacc is already present in the tree is beyond me, but it's useful for the purposes of this exercise.

12.1. Files

The file contents in this section must be used verbatime.

12.1.1. `Makefile`

```
# $NetBSD$

DISTNAME=       bison-1.25
CATEGORIES=     devel
MASTER_SITES=   ${MASTER_SITE_GNU}

MAINTAINER=     thorpej@netbsd.org
HOMEPAGE=       http://www.gnu.org/software/bison/bison.html

GNU_CONFIGURE=  yes
INFO_FILES=     bison.info

.include "../../mk/bsd.pkg.mk"
```

12.1.2. `pkg/COMMENT`

```
GNU yacc clone.
```

12.1.3. `pkg/DESCR`

```
GNU version of yacc.  Can make re-entrant parsers, and numerous other
```

```
improvements.  Why you would want this when Berkeley yacc(1) is part
of the NetBSD source tree is beyond me.
```

12.1.4. pkg/PLIST

```
@comment $NetBSD$
bin/bison
man/man1/bison.1.gz
@unexec install-info -delete %D/info/bison.info %D/info/dir
info/bison.info
info/bison.info-1
info/bison.info-2
info/bison.info-3
info/bison.info-4
info/bison.info-5
@exec install-info %D/info/bison.info %D/info/dir
share/bison.simple
share/bison.hairy
```

12.2. Checking a package with “pkglint”

The NetBSD package system comes with a tool called "pkglint" (located in the directory `pkgsrc/pkgtools/pkglint`) which helps to check the contents of these files. After installation it is quite easy to use, just change to the directory of the package you which to examine and execute “`pkglint`”:

```
tron@lyssa:/usr/pkgsrc/devel/bison> pkglint
OK: checking pkg/COMMENT.
OK: checking pkg/DESCR.
OK: checking Makefile.
OK: checking files/md5.
OK: checking patches/patch-aa.
looks fine.
```

Depending on the supplied command line arguments (see “`man pkglint`”) more intensive checks will be performed. Use e.g. “`pkglint -a -v`” for a very detailed and verbose check.

12.3. Steps for building, installing, packaging

Create the directory where the package lives, plus any auxiliary directories:

```
root@pumpy:/u/pkgsrc/lang(1765)# cd /usr/pkgsrc/lang
root@pumpy:/u/pkgsrc/lang(1765)# mkdir bison
root@pumpy:/u/pkgsrc/lang(1766)# cd bison
root@pumpy:/u/pkgsrc/lang/bison(1768)# mkdir files patches pkg
```

Create `Makefile`, `pkg/COMMENT`, `pkg/DESCR` and `pkg/PLIST` as in *Files* above, then continue with fetching the distfile:

```
root@pumpy:/u/pkgsrc/lang/bison(1769)# make fetch
>> bison-1.25.tar.gz doesn't seem to exist on this system.
>> Attempting to fetch from ftp://prep.ai.mit.edu/pub/gnu//.
Requesting ftp://prep.ai.mit.edu/pub/gnu//bison-1.25.tar.gz (via ftp://www-
proxy.myisp.com:80/)
ftp: Error retrieving file: 500 Internal error

>> Attempting to fetch from ftp://wuarchive.wustl.edu/systems/gnu//.
Requesting ftp://wuarchive.wustl.edu/systems/gnu//bison-1.25.tar.gz (via ftp://www-
proxy.myisp.com:80/)
ftp: Error retrieving file: 500 Internal error

>> Attempting to fetch from ftp://ftp.freebsd.org/pub/FreeBSD/distfiles//.
Requesting ftp://ftp.freebsd.org/pub/FreeBSD/distfiles//bison-1.25.tar.gz (via ftp://www
proxy.myisp.com:80/)
Successfully retrieved file.
```

Generate the checksum of the distfile into `files/md5`:

```
root@pumpy:/u/pkgsrc/lang/bison(1770)# make makesum
root@pumpy:/u/pkgsrc/lang/bison(1771)#
```

Now compile:

```
root@pumpy:/u/pkgsrc/lang/bison(1777)# make
>> Checksum OK for bison-1.25.tar.gz.
===>  Extracting for bison-1.25
===>  Patching for bison-1.25
===>   Ignoring empty patch directory
===>  Configuring for bison-1.25
```

```
creating cache ./config.cache
checking for gcc... cc
checking whether we are using GNU C... yes
checking for a BSD compatible install... /usr/bin/install -c -o bin -g bin
checking how to run the C preprocessor... cc -E
checking for minix/config.h... no
checking for POSIXized ISC... no
checking whether cross-compiling... no
checking for ANSI C header files... yes
checking for string.h... yes
checking for stdlib.h... yes
checking for memory.h... yes
checking for working const... yes
checking for working alloca.h... no
checking for alloca... yes
checking for strerror... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
===>  Building for bison-1.25
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g LR0.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g allocate.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g closure.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g conflicts.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g derives.c
cc -c -DXPFILE=\"/usr/pkg/share/bison.simple\"  -DXPFILE1=\"/usr/pkg/share/bison.hairy\"
DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1 -
DHAVE_STRERROR=1  -g  ./files.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g getargs.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g gram.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g lalr.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g lex.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g main.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g nullable.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g output.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g print.c
```

```
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g reader.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g reduce.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g symtab.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g warshall.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g version.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g getopt.c
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -
DHAVE_ALLOCA=1 -DHAVE_STRERROR=1 -I./../include  -g getopt1.c
cc  -g -o bison LR0.o allocate.o closure.o conflicts.o derives.o files.o         getargs
lable.o output.o print.o reader.o reduce.o symtab.o   warshall.o version.o
./files.c:240: warning: mktemp() possibly used unsafely, consider using mkstemp()
rm -f bison.s1
sed -e "/^#line/ s|bison|/usr/pkg/share/bison|" < ./bison.simple > bison.s1
```

Everything seems OK, so install the files:

```
root@pumpy:/u/pkgsrc/lang/bison(1785)# make install
>> Checksum OK for bison-1.25.tar.gz.
===>  Installing for bison-1.25
sh ./mkinstalldirs /usr/pkg/bin /usr/pkg/share  /usr/pkg/info /usr/pkg/man/man1
rm -f /usr/pkg/bin/bison
cd /usr/pkg/share; rm -f bison.simple bison.hairy
rm -f /usr/pkg/man/man1/bison.1 /usr/pkg/info/bison.info*
install -c  -o bin -g bin -m 555 bison /usr/pkg/bin/bison
/usr/bin/install -c -o bin -g bin -m 644 bison.s1 /usr/pkg/share/bison.simple
/usr/bin/install -c -o bin -g bin -m 644 ./bison.hairy /usr/pkg/share/bison.hairy
cd .; for f in bison.info*;  do /usr/bin/install -c -o bin -g bin -m 644 $f /usr/pkg/inf
/usr/bin/install -c -o bin -g bin -m 644 ./bison.1 /usr/pkg/man/man1/bison.1
===>  Registering installation for bison-1.25
```

You can now use bison, and also - if you decide so - remove it with “`pkg_delete bison-1.25`”. Should you decide that you want a binary package, do this now:

```
root@pumpy:/u/pkgsrc/lang/bison(1786)# make package
>> Checksum OK for bison-1.25.tar.gz.
===>  Building package for bison-1.25
Creating package bison-1.25.tgz
Registering depends:.
```

```
Creating gzip'd tar ball in '/u/pkgsrc/lang/bison/bison-1.25.tgz'
```

Now that you don't need the source and object files any more, clean up:

```
root@pumpy:/u/pkgsrc/lang/bison(1787)# make clean
===>  Cleaning for bison-1.25
```

Appendix A. Build logs

A.1. Building top

```
Script started on Fri Oct  3 13:22:31 1997
root@pumpy:/u/pkgsrc/sysutils/top(1342)# make
>> top-3.5beta5.tar.gz doesn't seem to exist on this system.
>> Attempting to fetch from ftp://ftp.groupsys.com/pub/top/.
Requesting ftp://ftp.groupsys.com/pub/top/top-3.5beta5.tar.gz (via ftp://www-
proxy.myisp.com:80/)
Successfully retrieved file.
>> Checksum OK for top-3.5beta5.tar.gz.
===>  Extracting for top-3.5beta5
===>  Patching for top-3.5beta5
===>  Applying NetBSD patches for top-3.5beta5
===>  Configuring for top-3.5beta5
/bin/cp /u/pkgsrc/sysutils/top/files/defaults /u/pkgsrc/sysutils/top/work/top-
3.5beta5/.defaults
chmod a-x /u/pkgsrc/sysutils/top/work/top-3.5beta5/install

Reading configuration from last time...

Using these settings:
        Bourne Shell   /bin/sh
          C compiler   cc
    Compiler options   -DHAVE_GETOPT -O
         Awk command   awk
     Install command   /usr/bin/install

              Module   netbsd13
             LoadMax   5.0
        Default TOPN   -1
        Nominal TOPN   18
       Default Delay   2
Random passwd access   yes
          Table Size   47
               Owner   root
         Group Owner   kmem
                Mode   2755
       bin directory   $(PREFIX)/bin
       man directory   $(PREFIX)/man/man1
       man extension   1
       man style       man

Building Makefile...
Building top.local.h...
```

```
Building top.1...
Doing a "make clean".
rm -f *.o top core core.* sigdesc.h
To create the executable, type "make".
To install the executable, type "make install".
===>  Building for top-3.5beta5
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c top.c
awk -f sigconv.awk /usr/include/sys/signal.h >sigdesc.h
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c commands.c
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c display.c
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c screen.c
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c username.c
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c utils.c
utils.c: In function ‘errmsg’:
utils.c:348: warning: return discards ‘const’ from pointer target type
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c version.c
cc -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O  -c getopt.c
cc "-DOSREV=12G" -DHAVE_GETOPT -DORDER -DHAVE_GETOPT -O -c machine.c
rm -f top
cc -o top top.o commands.o display.o screen.o username.o  utils.o version.o getopt.o ma-
chine.o -ltermcap -lm -lkvm
root@pumpy:/u/pkgsrc/sysutils/top(1343)# make install
>> Checksum OK for top-3.5beta5.tar.gz.
===>  Installing for top-3.5beta5
/usr/bin/install -o root -m 2755 -g kmem top /usr/pkg/bin
/usr/bin/install top.1 /usr/pkg/man/man1/top.1
strip /usr/pkg/bin/top
===>  Registering installation for top-3.5beta5
root@pumpy:/u/pkgsrc/sysutils/top(1344)#
```

A.2. Packaging top

```
root@pumpy:/u/pkgsrc/sysutils/top(1344)# make package
>> Checksum OK for top-3.5beta5.tar.gz.
===>  Building package for top-3.5beta5
Creating package top-3.5beta5.tgz
Registering depends:.
Creating gzip’d tar ball in ’/u/pkgsrc/sysutils/top/top-3.5beta5.tgz’
root@pumpy:/u/pkgsrc/sysutils/top(1345)#
```

Appendix B. Layout of the FTP server's package archive

Layout for precompiled binary packages on ftp.netbsd.org:

```
/pub/NetBSD/packages/
README
        distfiles/
pkgsrc -> /pub/NetBSD/NetBSD-current/pkgsrc
        1.3/
         i386/
                        All/
                        archivers/
                                foo -> ../All/foo
                        ...
                m68k/
                        All/
                        archivers/
                                foo -> ../All/foo
                        ...
                amiga -> m68k
                atari -> m68k
                ...
```

To create:

- "`cd /usr/pkgsrc ; make install ; make package`"
- upload /usr/pkgsrc/packages to ftp://ftp.netbsd.org/pub/NetBSD/packages/‘uname -r‘/‘sysctl -n hw.machine_arch‘
- if necessary, create appropriate symlinks for architectures sharing the same packages: "`ln -s ‘sysctl -n hw.machine‘ ‘sysctl -n hw.machine_arch‘`"

Disk space needed: approx. 1.3GB for one architecture (as of Jul 2000).