The pkgsrc guide

Documentation on the NetBSD packages system

(2015/01/01)

Alistair Crooks

agc@NetBSD.org

Hubert Feyrer

hubertf@NetBSD.org

The pkgsrc Developers

The pkgsrc guide: Documentation on the NetBSD packages system

by Alistair Crooks, Hubert Feyrer, The pkgsrc Developers

Published 2015/01/01 05:19:02

Copyright © 1994-2015 The NetBSD Foundation, Inc

pkgsrc is a centralized package management system for Unix-like operating systems. This guide provides information for users and developers of pkgsrc. It covers installation of binary and source packages, creation of binary and source packages and a high-level overview about the infrastructure.

Table of Contents

1. What is pkgsrc? ... 1

1.1. Introduction ... 1

1.1.1. Why pkgsrc? ... 1

1.1.2. Supported platforms ... 2

1.2. Overview ... 3

1.3. Terminology ... 3

1.3.1. Roles involved in pkgsrc ... 4

1.4. Typography ... 4

I. The pkgsrc user's guide ... 1

2. Where to get pkgsrc and how to keep it up-to-date ... 2

2.1. Getting pkgsrc for the first time ... 2

2.1.1. As tar archive ... 2

2.1.2. Via anonymous CVS ... 3

2.2. Keeping pkgsrc up-to-date ... 4

2.2.1. Via tar files ... 4

2.2.2. Via CVS ... 4

2.2.2.1. Switching between different pkgsrc branches ... 4

2.2.2.2. What happens to my changes when updating? ... 4

3. Using pkgsrc on systems other than NetBSD ... 5

3.1. Binary distribution ... 5

3.2. Bootstrapping pkgsrc ... 5

3.3. Platform-specific notes ... 5

3.3.1. Cygwin ... 6

3.3.2. Darwin (Mac OS X) ... 6

3.3.3. FreeBSD ... 6

3.3.4. GNU/kFreeBSD ... 6

3.3.5. Interix ... 7

3.3.5.1. When installing Interix/SFU ... 7

3.3.5.2. What to do if Interix/SFU is already installed ... 8

3.3.5.3. Important notes for using pkgsrc ... 8

3.3.5.4. Limitations of the Interix platform ... 8

3.3.5.5. Known issues for pkgsrc on Interix ... 9

3.3.6. IRIX ... 9

3.3.7. Linux ... 10

3.3.8. MirBSD ... 11

3.3.9. OpenBSD ... 11

3.3.10. Solaris ... 12

3.3.10.1. If you are using gcc ... 12

3.3.10.2. If you are using Sun WorkShop ... 12

3.3.10.3. Building 64-bit binaries with SunPro ... 13

3.3.10.4. Common problems ... 13

4. Using pkgsrc ... 14

4.1. Using binary packages ... 14

4.1.1. Finding binary packages ... 14

4.1.2. Installing binary packages ... 14

4.1.3. Deinstalling packages 15
4.1.4. Getting information about installed packages 15
4.1.5. Checking for security vulnerabilities in installed packages 15
4.1.6. Finding if newer versions of your installed packages are in pkgsrc 16
4.1.7. Other administrative functions 16
4.1.8. A word of warning 16
4.2. Building packages from source 17
4.2.1. Requirements 17
4.2.2. Fetching distfiles 17
4.2.3. How to build and install 18
5. Configuring pkgsrc 20
5.1. General configuration 20
5.2. Variables affecting the build process 21
5.3. Variables affecting the installation process 21
5.4. Selecting and configuring the compiler 22
5.4.1. Selecting the compiler 22
5.4.2. Additional flags to the compiler (`CFLAGS`) 23
5.4.3. Additional flags to the linker (`LDFLAGS`) 23
5.5. Developer/advanced settings 23
5.6. Selecting Build Options 23
6. Creating binary packages 25
6.1. Building a single binary package 25
6.2. Settings for creation of binary packages 25
7. Creating binary packages for everything in pkgsrc (bulk builds) 26
7.1. Preparations 26
7.2. Running a pbulk-style bulk build 26
7.2.1. Configuration 26
7.3. Requirements of a full bulk build 27
7.4. Creating a multiple CD-ROM packages collection 28
7.4.1. Example of cdpack 28
8. Directory layout of the installed files 29
8.1. File system layout in `${LOCALBASE}` 29
8.2. File system layout in `${VARBASE}` 31
9. Frequently Asked Questions 32
9.1. Are there any mailing lists for pkg-related discussion? 32
9.2. Utilities for package management (pkgtools) 32
9.3. How to use pkgsrc as non-root 33
9.4. How to resume transfers when fetching distfiles? 34
9.5. How can I install/use modular X.org from pkgsrc? 34
9.6. How to fetch files from behind a firewall 34
9.7. How to fetch files from HTTPS sites 35
9.8. How do I tell **make fetch** to do passive FTP? 35
9.9. How to fetch all distfiles at once 35
9.10. What does “Don’t know how to make /usr/share/tmac/tmac.andoc” mean? 36
9.11. What does “Could not find bsd.own.mk” mean? 36
9.12. Using ’sudo’ with pkgsrc 36
9.13. How do I change the location of configuration files? 36
9.14. Automated security checks 37

9.15. Why do some packages ignore my CFLAGS? 37
9.16. A package does not build. What shall I do? 38
9.17. What does "Makefile appears to contain unresolved cvs/rcs/??? merge conflicts" mean? 38

II. The pkgsrc developer's guide 39

10. Creating a new pkgsrc package from scratch 40
10.1. Common types of packages 41
10.1.1. Perl modules 41
10.1.2. KDE3 applications 41
10.1.3. Python modules and programs 41
10.2. Examples 42
10.2.1. How the www/nvu package came into pkgsrc 42
10.2.1.1. The initial package 42
10.2.1.2. Fixing all kinds of problems to make the package work 43
10.2.1.3. Installing the package 45
11. Package components - files, directories and contents 47
11.1. Makefile 47
11.2. distinfo 49
11.3. patches/* 49
11.3.1. Structure of a single patch file 49
11.3.2. Creating patch files 49
11.3.3. Sources where the patch files come from 50
11.3.4. Patching guidelines 50
11.3.5. Feedback to the author 51
11.4. Other mandatory files 51
11.5. Optional files 52
11.5.1. Files affecting the binary package 52
11.5.2. Files affecting the build process 53
11.5.3. Files affecting nothing at all 53
11.6. work* 53
11.7. files/* 54
12. Programming in Makefiles 55
12.1. Caveats 55
12.2. Makefile variables 55
12.2.1. Naming conventions 56
12.3. Code snippets 56
12.3.1. Adding things to a list 56
12.3.2. Converting an internal list into an external list 57
12.3.3. Passing variables to a shell command 57
12.3.4. Quoting guideline 58
12.3.5. Workaround for a bug in BSD Make 59
13. PLIST issues 60
13.1. RCS ID 60
13.2. Semi-automatic PLIST generation 60
13.3. Tweaking output of **make print-PLIST** 60
13.4. Variable substitution in PLIST 60
13.5. Man page compression 62

13.6. Changing PLIST source with PLIST_SRC 62
13.7. Platform-specific and differing PLISTs 62
13.8. Build-specific PLISTs 62
13.9. Sharing directories between packages 63
14. Buildlink methodology 64
14.1. Converting packages to use buildlink3 64
14.2. Writing buildlink3.mk files 65
14.2.1. Anatomy of a buildlink3.mk file 65
14.2.2. Updating BUILDLINK_API_DEPENDS.*pkg* and BUILDLINK_ABI_DEPENDS.*pkg* in buildlink3 67
14.3. Writing builtin.mk files 68
14.3.1. Anatomy of a builtin.mk file 68
14.3.2. Global preferences for native or pkgsrc software 69
15. The pkginstall framework 71
15.1. Files and directories outside the installation prefix 71
15.1.1. Directory manipulation 71
15.1.2. File manipulation 72
15.2. Configuration files 72
15.2.1. How PKG_SYSCONFDIR is set 72
15.2.2. Telling the software where configuration files are 73
15.2.3. Patching installations 73
15.2.4. Disabling handling of configuration files 74
15.3. System startup scripts 74
15.3.1. Disabling handling of system startup scripts 75
15.4. System users and groups 75
15.5. System shells 75
15.5.1. Disabling shell registration 75
15.6. Fonts 76
15.6.1. Disabling automatic update of the fonts databases 76
16. Options handling 77
16.1. Global default options 77
16.2. Converting packages to use bsd.options.mk 77
16.3. Option Names 79
16.4. Determining the options of dependencies 80
17. The build process 81
17.1. Introduction 81
17.2. Program location 81
17.3. Directories used during the build process 82
17.4. Running a phase 83
17.5. The *fetch* phase 83
17.5.1. What to fetch and where to get it from 83
17.5.2. How are the files fetched? 85
17.6. The *checksum* phase 85
17.7. The *extract* phase 85
17.8. The *patch* phase 86
17.9. The *tools* phase 86
17.10. The *wrapper* phase 86
17.11. The *configure* phase 87

17.12. The *build* phase....88
17.13. The *test* phase....88
17.14. The *install* phase....88
17.15. The *package* phase....90
17.16. Cleaning up....90
17.17. Other helpful targets....90
18. Tools needed for building or running....96
18.1. Tools for pkgsrc builds....96
18.2. Tools needed by packages....96
18.3. Tools provided by platforms....96
18.4. Questions regarding the tools....97
19. Making your package work....98
19.1. General operation....98
19.1.1. Portability of packages....98
19.1.2. How to pull in user-settable variables from `mk.conf`....98
19.1.3. User interaction....98
19.1.4. Handling licenses....99
19.1.5. Restricted packages....101
19.1.6. Handling dependencies....102
19.1.7. Handling conflicts with other packages....103
19.1.8. Packages that cannot or should not be built....104
19.1.9. Packages which should not be deleted, once installed....104
19.1.10. Handling packages with security problems....105
19.1.11. How to handle incrementing versions when fixing an existing package....105
19.1.12. Substituting variable text in the package files (the SUBST framework)....106
19.2. Fixing problems in the *fetch* phase....106
19.2.1. Packages whose distfiles aren't available for plain downloading....107
19.2.2. How to handle modified distfiles with the 'old' name....107
19.2.3. Packages hosted on github.com....107
19.2.3.1. Fetch based on a tagged release....107
19.2.3.2. Fetch based on a specific commit....108
19.2.3.3. Fetch based on release....108
19.3. Fixing problems in the *configure* phase....108
19.3.1. Shared libraries - libtool....108
19.3.2. Using libtool on GNU packages that already support libtool....110
19.3.3. GNU Autoconf/Automake....110
19.4. Programming languages....111
19.4.1. C, C++, and Fortran....111
19.4.2. Java....111
19.4.3. Packages containing perl scripts....112
19.4.4. Packages containing shell scripts....112
19.4.5. Other programming languages....112
19.5. Fixing problems in the *build* phase....112
19.5.1. Compiling C and C++ code conditionally....113
19.5.1.1. C preprocessor macros to identify the operating system....113
19.5.1.2. C preprocessor macros to identify the hardware architecture....113
19.5.1.3. C preprocessor macros to identify the compiler....114
19.5.2. How to handle compiler bugs....114

19.5.3. Undefined reference to "..." 114
19.5.3.1. Special issue: The SunPro compiler 114
19.5.4. Running out of memory 115
19.6. Fixing problems in the *install* phase 115
19.6.1. Creating needed directories 115
19.6.2. Where to install documentation 115
19.6.3. Installing highscore files 116
19.6.4. Adding DESTDIR support to packages 116
19.6.5. Packages with hardcoded paths to other interpreters 116
19.6.6. Packages installing perl modules 117
19.6.7. Packages installing info files 117
19.6.8. Packages installing man pages 118
19.6.9. Packages installing GConf data files 118
19.6.10. Packages installing scrollkeeper/rarian data files 118
19.6.11. Packages installing X11 fonts 119
19.6.12. Packages installing GTK2 modules 119
19.6.13. Packages installing SGML or XML data 119
19.6.14. Packages installing extensions to the MIME database 120
19.6.15. Packages using intltool 120
19.6.16. Packages installing startup scripts 120
19.6.17. Packages installing TeX modules 121
19.6.18. Packages supporting running binaries in emulation 121
19.6.19. Packages installing hicolor theme icons 121
19.6.20. Packages installing desktop files 122
19.7. Marking packages as having problems 122
20. Debugging 123
21. Submitting and Committing 125
21.1. Submitting binary packages 125
21.2. Submitting source packages (for non-NetBSD-developers) 125
21.3. General notes when adding, updating, or removing packages 125
21.4. Committing: Adding a package to CVS 126
21.5. Updating a package to a newer version 126
21.6. Renaming a package in pkgsrc 127
21.7. Moving a package in pkgsrc 127
22. Frequently Asked Questions 129
23. GNOME packaging and porting 131
23.1. Meta packages 131
23.2. Packaging a GNOME application 132
23.3. Updating GNOME to a newer version 133
23.4. Patching guidelines 134
III. The pkgsrc infrastructure internals 135
24. Design of the pkgsrc infrastructure 136
24.1. The meaning of variable definitions 136
24.2. Avoiding problems before they arise 136
24.3. Variable evaluation 137
24.3.1. At load time 137
24.3.2. At runtime 137

24.4. How can variables be specified? 137
24.5. Designing interfaces for Makefile fragments 138
24.5.1. Procedures with parameters 138
24.5.2. Actions taken on behalf of parameters 138
24.6. The order in which files are loaded 138
24.6.1. The order in `bsd.prefs.mk` 139
24.6.2. The order in `bsd.pkg.mk` 139
25. Regression tests 140
25.1. The regression tests framework 140
25.2. Running the regression tests 140
25.3. Adding a new regression test 140
25.3.1. Overridable functions 140
25.3.2. Helper functions 141
26. Porting pkgsrc 142
26.1. Porting pkgsrc to a new operating system 142
26.2. Adding support for a new compiler 142
A. A simple example package: bison 143
A.1. files 143
A.1.1. Makefile 143
A.1.2. DESCR 143
A.1.3. PLIST 143
A.1.4. Checking a package with **pkglint** 144
A.2. Steps for building, installing, packaging 144
B. Build logs 147
B.1. Building figlet 147
B.2. Packaging figlet 148
C. Directory layout of the pkgsrc FTP server 150
C.1. `distfiles`: The distributed source files 150
C.2. `misc`: Miscellaneous things 150
C.3. `packages`: Binary packages 150
C.4. `reports`: Bulk build reports 151
C.5. `current`, `pkgsrc-20`*xx*`Q`*y*: source packages 151
D. Editing guidelines for the pkgsrc guide 152
D.1. Make targets 152
D.2. Procedure 152

List of Tables

1-1. Platforms supported by pkgsrc 2
11-1. Patching examples 51
23-1. PLIST handling for GNOME packages 132

Chapter 1.
What is pkgsrc?

1.1. Introduction

There is a lot of software freely available for Unix-based systems, which is usually available in form of the source code. Before such software can be used, it needs to be configured to the local system, compiled and installed, and this is exactly what The NetBSD Packages Collection (pkgsrc) does. pkgsrc also has some basic commands to handle binary packages, so that not every user has to build the packages for himself, which is a time-costly task.

pkgsrc currently contains several thousand packages, including:

- `www/apache` - The Apache web server
- `www/firefox` - The Firefox web browser
- `meta-pkgs/gnome` - The GNOME Desktop Environment
- `meta-pkgs/kde3` - The K Desktop Environment

...just to name a few.

pkgsrc has built-in support for handling varying dependencies, such as pthreads and X11, and extended features such as IPv6 support on a range of platforms.

1.1.1. Why pkgsrc?

pkgsrc provides the following key features:

- Easy building of software from source as well as the creation and installation of binary packages. The source and latest patches are retrieved from a master or mirror download site, checksum verified, then built on your system. Support for binary-only distributions is available for both native platforms and NetBSD emulated platforms.
- All packages are installed in a consistent directory tree, including binaries, libraries, man pages and other documentation.
- Package dependencies, including when performing package updates, are handled automatically. The configuration files of various packages are handled automatically during updates, so local changes are preserved.
- Like NetBSD, pkgsrc is designed with portability in mind and consists of highly portable code. This allows the greatest speed of development when porting to a new platform. This portability also ensures that pkgsrc is *consistent across all platforms*.

- The installation prefix, acceptable software licenses, international encryption requirements and build-time options for a large number of packages are all set in a simple, central configuration file.
- The entire source (not including the distribution files) is freely available under a BSD license, so you may extend and adapt pkgsrc to your needs. Support for local packages and patches is available right out of the box, so you can configure it specifically for your environment.

The following principles are basic to pkgsrc:

- "It should only work if it's right." — That means, if a package contains bugs, it's better to find them and to complain about them rather than to just install the package and hope that it works. There are numerous checks in pkgsrc that try to find such bugs: Static analysis tools (`pkgtools/pkglint`), build-time checks (portability of shell scripts), and post-installation checks (installed files, references to shared libraries, script interpreters).
- "If it works, it should work everywhere" — Like NetBSD has been ported to many hardware architectures, pkgsrc has been ported to many operating systems. Care is taken that packages behave the same on all platforms.

1.1.2. Supported platforms

pkgsrc consists of both a source distribution and a binary distribution for these operating systems. After retrieving the required source or binaries, you can be up and running with pkgsrc in just minutes!

pkgsrc was derived from FreeBSD's ports system, and initially developed for NetBSD only. Since then, pkgsrc has grown a lot, and now supports the following platforms:

Table 1-1. Platforms supported by pkgsrc

Platform	Date Support Added
NetBSD (http://www.NetBSD.org/)	Aug 1997
Solaris (http://wwws.sun.com/software/solaris/)	Mar 1999
Linux (http://www.kernel.org/)	Jun 1999
Darwin (http://developer.apple.com/darwin/) (Mac OS X (http://developer.apple.com/macosx/))	Oct 2001
FreeBSD (http://www.freebsd.org/)	Nov 2002
OpenBSD (http://www.openbsd.org/)	Nov 2002
IRIX (http://www.sgi.com/software/irix/)	Dec 2002
BSD/OS	Dec 2003
AIX (http://www-1.ibm.com/servers/aix/)	Dec 2003
Interix (http://www.microsoft.com/windows/sfu/) (Microsoft Windows Services for Unix)	Mar 2004
DragonFlyBSD (http://www.dragonflybsd.org/)	Oct 2004
OSF/1 (http://www.tru64.org/)	Nov 2004
HP-UX (http://www.hp.com/products1/unix/)	Apr 2007

Platform	Date Support Added
Haiku (http://www.haiku-os.org/)	Sep 2010
MirBSD (http://www.mirbsd.org/)	Jan 2011
Minix3 (http://www.minix3.org/)	Nov 2011
Cygwin (http://cygwin.com/)	Mar 2013
GNU/kFreeBSD (http://www.debian.org/ports/kfreebsd-gnu/)	Jul 2013

1.2. Overview

This document is divided into three parts. The first, *The pkgsrc 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 one's own copy using the NetBSD package system. The second part, *The pkgsrc developer'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. The third part, *The pkgsrc infrastructure internals* is intended for those who want to understand how pkgsrc is implemented.

This document is available in various formats: HTML (index.html), PDF (pkgsrc.pdf), PS (pkgsrc.ps), TXT (pkgsrc.txt).

1.3. 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 pkgsrc. Packages are traditionally stored under `/usr/pkgsrc`.

The NetBSD package system

This is the former name of "pkgsrc". It is part of the NetBSD operating system and can be bootstrapped to run on non-NetBSD operating systems as well. It handles 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 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 usually stored below `/usr/pkgsrc/distfiles`.

Port

This is the term used by FreeBSD and OpenBSD people for what we call a package. In NetBSD terminology, "port" refers to a different architecture.

Precompiled/binary package

A set of binaries built with pkgsrc from a distfile 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 usually generated in `/usr/pkgsrc/packages`; there is also an archive on ftp.NetBSD.org (ftp://ftp.NetBSD.org/pub/pkgsrc/packages/).

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.

1.3.1. Roles involved in pkgsrc

pkgsrc users

The pkgsrc users are people who use the packages provided by pkgsrc. Typically they are system administrators. The people using the software that is inside the packages (maybe called "end users") are not covered by the pkgsrc guide.

There are two kinds of pkgsrc users: Some only want to install pre-built binary packages. Others build the pkgsrc packages from source, either for installing them directly or for building binary packages themselves. For pkgsrc users Part I in *The pkgsrc guide* should provide all necessary documentation.

package maintainers

A package maintainer creates packages as described in Part II in *The pkgsrc guide*.

infrastructure developers

These people are involved in all those files that live in the `mk/` directory and below. Only these people should need to read through Part III in *The pkgsrc guide*, though others might be curious, too.

1.4. Typography

When giving examples for commands, shell prompts are used to show if the command should/can be issued as root, or if "normal" user privileges are sufficient. We use a `#` for root's shell prompt, and a `%` for users' shell prompt, assuming they use the C-shell or tcsh.

I. The pkgsrc user's guide

Chapter 2.
Where to get pkgsrc and how to keep it up-to-date

Before you download and extract the files, you need to decide where you want to extract them. When using pkgsrc as root user, pkgsrc is usually installed in `/usr/pkgsrc`. You are though free to install the sources and binary packages wherever you want in your filesystem, provided that the pathname does not contain white-space or other characters that are interpreted specially by the shell and some other programs. A safe bet is to use only letters, digits, underscores and dashes.

2.1. Getting pkgsrc for the first time

Before you download any pkgsrc files, you should decide whether you want the *current* branch or the *stable* branch. The latter is forked on a quarterly basis from the current branch and only gets modified for security updates. The names of the stable branches are built from the year and the quarter, for example `2014Q3`.

The second step is to decide *how* you want to download pkgsrc. You can get it as a tar file or via CVS. Both ways are described here.

Note that tar archive contains CVS working copy. Thus you can switch to using CVS at any later time.

2.1.1. As tar archive

The primary download location for all pkgsrc files is http://ftp.NetBSD.org/pub/pkgsrc/ or ftp://ftp.NetBSD.org/pub/pkgsrc/ (it points to the same location). There are a number of subdirectories for different purposes, which are described in detail in Appendix C.

The tar archive for the current branch is in the directory `current` and is called `pkgsrc.tar.gz` (http://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.gz). It is autogenerated daily.

To save download time we provide bzip2- and xz-compressed archives which are published at `pkgsrc.tar.bz2` (http://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.bz2) and `pkgsrc.tar.xz` (http://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.xz) respectively.

You can fetch the same files using FTP.

The tar file for the stable branch 2014Q3 is in the directory `pkgsrc-2014Q3` and is also called `pkgsrc.tar.gz` (ftp://ftp.NetBSD.org/pub/pkgsrc/pkgsrc-2014Q3/pkgsrc.tar.gz).

To download a pkgsrc stable tarball, run:

```
$ ftp ftp://ftp.NetBSD.org/pub/pkgsrc/pkgsrc-20xxQy/pkgsrc.tar.gz
```

Where *`pkgsrc-20xxQy`* is the stable branch to be downloaded, for example, “pkgsrc-2014Q3”.

If you prefer, you can also fetch it using "wget", "curl", or your web browser.

Then, extract it with:

```
$ tar -xzf pkgsrc.tar.gz -C /usr
```

This will create the directory `pkgsrc/` in `/usr/` and all the package source will be stored under `/usr/pkgsrc/`.

To download pkgsrc-current, run:

```
$ ftp ftp://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.gz
```

2.1.2. Via anonymous CVS

To fetch a specific pkgsrc stable branch, run:

```
$ cd /usr && cvs -q -z2 -d anoncvs@anoncvs.NetBSD.org:/cvsroot checkout -r pkgsrc-20xxQy -P pkgsrc
```

Where *`pkgsrc-20xxQy`* is the stable branch to be checked out, for example, “pkgsrc-2014Q3”

This will create the directory `pkgsrc/` in your `/usr/` directory and all the package source will be stored under `/usr/pkgsrc/`.

To fetch the pkgsrc current branch, run:

```
$ cd /usr && cvs -q -z2 -d anoncvs@anoncvs.NetBSD.org:/cvsroot checkout -P pkgsrc
```

Refer to the list of available mirrors (http://www.NetBSD.org/mirrors/#anoncvs) to choose a faster CVS mirror, if needed.

If you get error messages from `rsh`, you need to set CVS_RSH variable. E.g.:

```
$ cd /usr && env CVS_RSH=ssh cvs -q -z2 -d anoncvs@anoncvs.NetBSD.org:/cvsroot checkout -P pkgsrc
```

Refer to documentation on your command shell how to set CVS_RSH=ssh permanently. For Bourne shells, you can set it in your `.profile` or better globally in `/etc/profile`:

```
# set CVS remote shell command
CVS_RSH=ssh
export CVS_RSH
```

By default, CVS doesn’t do things like most people would expect it to do. But there is a way to convince CVS, by creating a file called `.cvsrc` in your home directory and saving the following lines to it. This file will save you lots of headache and some bug reports, so we strongly recommend it. You can find an explanation of this file in the CVS documentation.

```
# recommended CVS configuration file from the pkgsrc guide
cvs -q -z2
checkout -P
update -dP
diff -upN
rdiff -u
release -d
```

2.2. Keeping pkgsrc up-to-date

The preferred way to keep pkgsrc up-to-date is via CVS (which also works if you have first installed it via a tar file). It saves bandwidth and hard disk activity, compared to downloading the tar file again.

2.2.1. Via tar files

Warning

When updating from a tar file, you first need to completely remove the old pkgsrc directory. Otherwise those files that have been removed from pkgsrc in the mean time will not be removed on your local disk, resulting in inconsistencies. When removing the old files, any changes that you have done to the pkgsrc files will be lost after updating. Therefore updating via CVS is strongly recommended.

Note that by default the distfiles and the binary packages are saved in the pkgsrc tree, so don't forget to rescue them before updating. You can also configure pkgsrc to store distfiles and packages in directories outside the pkgsrc tree by setting the `DISTDIR` and `PACKAGES` variables. See Chapter 5 for the details.

To update pkgsrc from a tar file, download the tar file as explained above. Then, make sure that you have not made any changes to the files in the pkgsrc directory. Remove the pkgsrc directory and extract the new tar file. Done.

2.2.2. Via CVS

To update pkgsrc via CVS, change to the `pkgsrc` directory and run cvs:

```
$ cd /usr/pkgsrc && cvs update -dP
```

If you get error messages from `rsh`, you need to set CVS_RSH variable as described above. E.g.:

```
$ cd /usr/pkgsrc && env CVS_RSH=ssh cvs up -dP
```

2.2.2.1. Switching between different pkgsrc branches

When updating pkgsrc, the CVS program keeps track of the branch you selected. But if you, for whatever reason, want to switch from the stable branch to the current one, you can do it by adding the option "-A" after the "update" keyword. To switch from the current branch back to the stable branch, add the "-rpkgsrc-2014Q3" option.

2.2.2.2. What happens to my changes when updating?

When you update pkgsrc, the CVS program will only touch those files that are registered in the CVS repository. That means that any packages that you created on your own will stay unmodified. If you change files that are managed by CVS, later updates will try to merge your changes with those that have been done by others. See the CVS manual, chapter "update" for details.

Chapter 3.
Using pkgsrc on systems other than NetBSD

3.1. Binary distribution

See Section 4.1.

3.2. Bootstrapping pkgsrc

pkgsrc can be bootstrapped for use in two different modes: privileged and unprivileged one. In unprivileged mode in contrast to privileged one all programs are installed under one particular user and cannot utilise privileged operations (packages don't create special users and all special file permissions like setuid are ignored).

Installing the bootstrap kit from source should be as simple as:

```
# env CVS_RSH=ssh cvs -d anoncvs@anoncvs.NetBSD.org:/cvsroot checkout -P pkgsrc
# cd pkgsrc/bootstrap
# ./bootstrap
```

To bootstrap in unprivileged mode pass "--unprivileged" flag to **bootstrap**

By default, in privileged mode pkgsrc uses `/usr/pkg` for *prefix* where programs will be installed in, and `/var/db/pkg` for the package database directory where pkgsrc will do its internal bookkeeping, `/var` is used as *varbase*, where packages install their persistent data. In unprivileged mode pkgsrc uses `~/pkg` for *prefix*, `~/pkg/var/db/pkg` for the package database, and `~/pkg/var` for *varbase*.

You can change default layout using command-line arguments. Run "./bootstrap --help" to get details.

> **Note:** The bootstrap installs a **bmake** tool. Use this **bmake** when building via pkgsrc. For examples in this guide, use **bmake** instead of "make".

> **Note:** It is possible to bootstrap multiple instances of pkgsrc using non-intersecting directories. Use **bmake** corresponding to the installation you're working with to build and install packages.

3.3. Platform-specific notes

Here are some platform-specific notes you should be aware of.

3.3.1. Cygwin

Cygwin 1.7.x and later are supported.

You need to install minimal base packages in 'Base' category plus any of compiler, gcc, gcc4, and/or clang. For gcc and gcc4, C and C++ compiler will be installed by default, but you can install Fortran compiler additionally because it will be required to use libtool. If it is not installed (or too old), Fortran compiler will be installed with pkgsrc automatically.

As noted in Cygwin FAQ: 'Why doesn't su work?' (http://cygwin.com/faq-nochunks.html#faq.using.su), su(1) command has been in Cygwin distribution, but it has never worked. Unless you bootstrap pkgsrc with the --unprivileged option, workaround is:

- Right click "Cygwin Terminal" in your Start Menu, then pick "Run as administrator".

3.3.2. Darwin (Mac OS X)

Darwin 5.x and up are supported.

Before you start, you need to download and install the Mac OS X Developer Tools from Apple's Developer Connection. This requires (free) membership. See http://developer.apple.com/macosx/ for details. Also, make sure you install X11 (an optional package included with the Developer Tools) if you intend to build packages that use the X11 Window System. (If you don't want or need the full Xcode GUI, download and install Command Line Tools for Xcode.)

3.3.3. FreeBSD

FreeBSD 8.3 and 9.0 have been tested and are supported, other versions may work.

Care should be taken so that the tools that this kit installs do not conflict with the FreeBSD userland tools. There are several steps:

1. FreeBSD stores its ports pkg database in `/var/db/pkg`. It is therefore recommended that you choose a different location (e.g. `/usr/pkgdb`) by using the --pkgdbdir option to the bootstrap script.
2. If you do not intend to use the FreeBSD ports tools, it's probably a good idea to move them out of the way to avoid confusion, e.g.

   ```
   # cd /usr/sbin
   # mv pkg_add pkg_add.orig
   # mv pkg_create pkg_create.orig
   # mv pkg_delete pkg_delete.orig
   # mv pkg_info pkg_info.orig
   ```

3. An example `mk.conf` file will be placed in `/etc/mk.conf.example` file when you use the bootstrap script.

3.3.4. GNU/kFreeBSD

Debian GNU/kFreeBSD is the only GNU/kFreeBSD distribution now. Debian GNU/kFreeBSD 7.0 or later is tested and supported.

You should install ncurses (libncurses and libncurses-dev) packages.

3.3.5. Interix

Interix is a POSIX-compatible subsystem for the Windows NT kernel, providing a Unix-like environment with a tighter kernel integration than available with Cygwin. It is part of the Windows Services for Unix package, available for free for any licensed copy of Windows 2000, XP (not including XP Home), or 2003. SFU can be downloaded from http://www.microsoft.com/windows/sfu/.

Services for Unix 3.5 has been tested. 3.0 or 3.1 may work, but are not officially supported. (The main difference in 3.0/3.1 is lack of pthreads, but other parts of libc may also be lacking.)

Services for Unix Applications (aka SUA) is an integrated component of Windows Server 2003 R2 (5.2), Windows Vista and Windows Server 2008 (6.0), Windows 7 and Windows Server 2008 R2 (6.1). As of this writing, the SUA's Interix 6.0 (32bit) and 6.1 (64bit) subsystems have been tested. Other versions may work as well. The Interix 5.x subsystem has not yet been tested with pkgsrc.

3.3.5.1. When installing Interix/SFU

At an absolute minimum, the following packages must be installed from the Windows Services for Unix 3.5 distribution in order to use pkgsrc:

- Utilities -> Base Utilities
- Interix GNU Components -> (all)
- Remote Connectivity
- Interix SDK

When using pkgsrc on Interix, DO NOT install the Utilities subcomponent "UNIX Perl". That is Perl 5.6 without shared module support, installed to /usr/local, and will only cause confusion. Instead, install Perl 5.8 from pkgsrc (or from a binary package).

The Remote Connectivity subcomponent "Windows Remote Shell Service" does not need to be installed, but Remote Connectivity itself should be installed in order to have a working inetd.

During installation you may be asked whether to enable setuid behavior for Interix programs, and whether to make pathnames default to case-sensitive. Setuid should be enabled, and case-sensitivity MUST be enabled. (Without case-sensitivity, a large number of packages including perl will not build.)

NOTE: Newer Windows service packs change the way binary execution works (via the Data Execution Prevention feature). In order to use pkgsrc and other gcc-compiled binaries reliably, a hotfix containing POSIX.EXE, PSXDLL.DLL, PSXRUN.EXE, and PSXSS.EXE (899522 or newer) must be installed. Hotfixes are available from Microsoft through a support contract; however, Debian Interix Port has made most Interix hotfixes available for personal use from http://www.debian-interix.net/hotfixes/.

In addition to the hotfix noted above, it may be necessary to disable Data Execution Prevention entirely to make Interix functional. This may happen only with certain types of CPUs; the cause is not fully understood at this time. If gcc or other applications still segfault repeatedly after installing one of the

hotfixes note above, the following option can be added to the appropriate "boot.ini" line on the Windows boot drive: /NoExecute=AlwaysOff (WARNING, this will disable DEP completely, which may be a security risk if applications are often run as a user in the Administrators group!)

3.3.5.2. What to do if Interix/SFU is already installed

If SFU is already installed and you wish to alter these settings to work with pkgsrc, note the following things.

- To uninstall UNIX Perl, use Add/Remove Programs, select Microsoft Windows Services for UNIX, then click Change. In the installer, choose Add or Remove, then uncheck Utilities->UNIX Perl.
- To enable case-sensitivity for the file system, run REGEDIT.EXE, and change the following registry key:

  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel

  Set the DWORD value "obcaseinsensitive" to 0; then reboot.
- To enable setuid binaries (optional), run REGEDIT.EXE, and change the following registry key:

  HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Services for UNIX

  Set the DWORD value "EnableSetuidBinaries" to 1; then reboot.

3.3.5.3. Important notes for using pkgsrc

The package manager (either the pkgsrc "su" user, or the user running "pkg_add") must be a member of the local Administrators group. Such a user must also be used to run the bootstrap. This is slightly relaxed from the normal pkgsrc requirement of "root".

The package manager should use a umask of 002. "make install" will automatically complain if this is not the case. This ensures that directories written in /var/db/pkg are Administrators-group writeable.

The popular Interix binary packages from http://www.interopsystems.com/ use an older version of pkgsrc's pkg_* tools. Ideally, these should NOT be used in conjunction with pkgsrc. If you choose to use them at the same time as the pkgsrc packages, ensure that you use the proper pkg_* tools for each type of binary package.

The TERM setting used for DOS-type console windows (including those invoked by the csh and ksh startup shortcuts) is "interix". Most systems don't have a termcap/terminfo entry for it, but the following .termcap entry provides adequate emulation in most cases:

```
interix:kP=\E[S:kN=\E[T:kH=\E[U:dc@:DC@:tc=pcansi:
```

3.3.5.4. Limitations of the Interix platform

Though Interix suffices as a familiar and flexible substitute for a full Unix-like platform, it has some drawbacks that should be noted for those desiring to make the most of Interix.

- **X11:**

Interix comes with the standard set of X11R6 client libraries, and can run X11 based applications, but it does *not* come with an X server. Some options are StarNet X-Win32 (http://www.starnet.com/products/xwin32/), Hummingbird Exceed (http://connectivity.hummingbird.com/products/nc/exceed/) (available in a trimmed version for Interix from Interop Systems as the Interop X Server (http://www.interopsystems.com/InteropXserver.htm)), and the free X11 server included with Cygwin (http://x.cygwin.com/).

- **X11 acceleration:**

  Because Interix runs in a completely different NT subsystem from Win32 applications, it does not currently support various X11 protocol extensions for acceleration (such as MIT-SHM or DGA). Most interactive applications to a local X server will run reasonably fast, but full motion video and other graphics intensive applications may require a faster-than-expected CPU.

- **Audio:**

  Interix has no native support for audio output. For audio support, pkgsrc uses the **esound** client/server audio system on Interix. Unlike on most platforms, the `audio/esound` package does *not* contain the **esd** server component. To output audio via an Interix host, the `emulators/cygwin_esound` package must also be installed.

- **CD/DVDs, USB, and SCSI:**

  Direct device access is not currently supported in Interix, so it is not currently possible to access CD/DVD drives, USB devices, or SCSI devices through non-filesystem means. Among other things, this makes it impossible to use Interix directly for CD/DVD burning.

- **Tape drives:**

  Due to the same limitations as for CD-ROMs and SCSI devices, tape drives are also not directly accessible in Interix. However, support is in work to make tape drive access possible by using Cygwin as a bridge (similarly to audio bridged via Cygwin's esound server).

3.3.5.5. Known issues for pkgsrc on Interix

It is not necessary, in general, to have a "root" user on the Windows system; any member of the local Administrators group will suffice. However, some packages currently assume that the user named "root" is the privileged user. To accommodate these, you may create such a user; make sure it is in the local group Administrators (or your language equivalent).

pkg_add creates directories of mode 0755, not 0775, in `$PKG_DBDIR`. For the time being, install packages as the local Administrator (or your language equivalent), or run the following command after installing a package to work around the issue:

```
# chmod -R g+w $PKG_DBDIR
```

3.3.6. IRIX

You will need a working C compiler, either gcc or SGI's MIPS and MIPSpro compiler (cc/c89). Please set the `CC` environment variable according to your preference. If you do not have a license for the MIPSpro compiler suite, you can download a gcc tardist file from http://freeware.sgi.com/.

Please note that you will need IRIX 6.5.17 or higher, as this is the earliest version of IRIX providing support for if_indextoname(3), if_nametoindex(3), etc.

At this point in time, pkgsrc only supports one ABI at a time. That is, you cannot switch between the old 32-bit ABI, the new 32-bit ABI and the 64-bit ABI. If you start out using "abi=n32", that's what all your packages will be built with.

Therefore, please make sure that you have no conflicting `CFLAGS` in your environment or the `mk.conf`. Particularly, make sure that you do not try to link n32 object files with lib64 or vice versa. Check your `/etc/compiler.defaults`!

If you have the actual pkgsrc tree mounted via NFS from a different host, please make sure to set `WRKOBJDIR` to a local directory, as it appears that IRIX linker occasionally runs into issues when trying to link over a network-mounted file system.

The bootstrapping process should set all the right options for programs such as imake(1), but you may want to set some options depending on your local setup. Please see `pkgsrc/mk/defaults/mk.conf` and, of course, your compiler's man pages for details.

If you are using SGI's MIPSPro compiler, please set

```
PKGSRC_COMPILER=        mipspro
```

in `mk.conf`. Otherwise, pkgsrc will assume you are using gcc and may end up passing invalid flags to the compiler. Note that bootstrap should create an appropriate `mk.conf.example` by default.

If you have both the MIPSPro compiler chain installed as well as gcc, but want to make sure that MIPSPro is used, please set your `PATH` to *not* include the location of gcc (often `/usr/freeware/bin`), and (important) pass the '--preserve-path' flag.

3.3.7. Linux

Some versions of Linux (for example Debian GNU/Linux) need either libtermcap or libcurses (libncurses). Installing the distributions libncurses-dev package (or equivalent) should fix the problem.

pkgsrc supports both gcc (GNU Compiler Collection) and icc (Intel C++ Compiler). gcc is the default. icc 8.0 and 8.1 on i386 have been tested.

To bootstrap using icc, assuming the default icc installation directory:

```
env ICCBASE=/opt/intel/cc/10.1.008 ./bootstrap --compiler=icc
```

> **Note:** For icc 8.0 you must add 'LDFLAGS=-static-libcxa' to this.
>
> For icc 8.1 you must add 'LDFLAGS=-i-static' instead.
>
> For icc 10.1 neither of these appears to be necessary.

Use a value for ICCBASE that corresponds to the directory where icc is installed. After bootstrapping, set `ICCBASE` in `mk.conf`:

```
ICCBASE=                /opt/intel/cc/10.1.008
```

The pkgsrc default for `ICCBASE` is `/opt/intel_cc_80`. This is the default install directory for icc 8.0. If you are using a more recent version, be sure to set the correct path explicitly.

pkgsrc uses the static linking method of the runtime libraries provided by icc, so binaries can be run on other systems which do not have the shared libraries installed.

Libtool, however, extracts a list of libraries from the ld(1) command run when linking a C++ shared library and records it, throwing away the -Bstatic and -Bdynamic options interspersed between the libraries. This means that libtool-linked C++ shared libraries will have a runtime dependency on the icc libraries until this is fixed in libtool.

3.3.8. MirBSD

pkgsrc has been tested on MirBSD #10-current (2011 and newer). Older versions might also work. Releases before #10 are not supported.

The package tools of the (older) native ports tree, MirPorts (//www.mirbsd.org/ports.htm), have the same names as the ones used by pkgsrc. Care should be taken that the right tools are used. When installing packages from source, use the `bmake` command for pkgsrc and `mmake` for MirPorts.

pkgsrc and MirPorts use the same location for the package database, `/var/db/pkg`. It is strongly recommended to use `/usr/pkg/db` instead, so that the pkgsrc tree is self-contained. This is also the default setting used in the binary package builds.

Binary packages for MirBSD/i386 can be found on the pkgsrc ftp server. The bootstrap kit there already contains the **pkgin** package manager. See the pkgsrc on MirOS (https://www.mirbsd.org/pkgsrc.htm) page for more details.

3.3.9. OpenBSD

OpenBSD 5.1 has been tested and supported, other versions may work.

Care should be taken so that the tools that this kit installs do not conflict with the OpenBSD userland tools. There are several steps:

1. OpenBSD stores its ports pkg database in `/var/db/pkg`. It is therefore recommended that you choose a different location (e.g. `/usr/pkgdb`) by using the --pkgdbdir option to the bootstrap script.
2. If you do not intend to use the OpenBSD ports tools, it's probably a good idea to move them out of the way to avoid confusion, e.g.

```
# cd /usr/sbin
# mv pkg_add pkg_add.orig
# mv pkg_create pkg_create.orig
# mv pkg_delete pkg_delete.orig
# mv pkg_info pkg_info.orig
```

3. An example `mk.conf` file will be placed in `/etc/mk.conf.example` file when you use the bootstrap script. OpenBSD's make program uses `mk.conf` as well. You can work around this by enclosing all the pkgsrc-specific parts of the file with:

```
.ifdef BSD_PKG_MK
# pkgsrc stuff, e.g. insert defaults/mk.conf or similar here
.else
# OpenBSD stuff
.endif
```

3.3.10. Solaris

Solaris 2.6 through 10 are supported on both x86 and sparc. You will need a working C compiler. Both gcc 4.5.3 and Sun WorkShop 5 have been tested.

The following packages are required on Solaris 8 for the bootstrap process and to build packages.

- SUNWsprot
- SUNWarc
- SUNWbtool
- SUNWtoo
- SUNWlibm

Please note that the use of GNU binutils on Solaris is *not* supported, as of June 2006.

Whichever compiler you use, please ensure the compiler tools and your $prefix are in your `PATH`. This includes `/usr/ccs/{bin,lib}` and e.g. `/usr/pkg/{bin,sbin}`.

3.3.10.1. If you are using gcc

It makes life much simpler if you only use the same gcc consistently for building all packages.

It is recommended that an external gcc be used only for bootstrapping, then either build gcc from `lang/gcc46` or install a binary gcc package, then remove gcc used during bootstrapping.

Binary packages of gcc can be found through http://www.sunfreeware.com/.

3.3.10.2. If you are using Sun WorkShop

You will need at least the following packages installed (from WorkShop 5.0)

- SPROcc - Sun WorkShop Compiler C 5.0
- SPROcpl - Sun WorkShop Compiler C++ 5.0
- SPROild - Sun WorkShop Incremental Linker
- SPROlang - Sun WorkShop Compilers common components

You should set the following variables in your `mk.conf` file:

```
CC=     cc
CXX=    CC
CPP=    cc -E
CXXCPP= CC -E
```

Note: The `CPP` setting might break some packages that use the C preprocessor for processing things other than C source code.

3.3.10.3. Building 64-bit binaries with SunPro

To build 64-bit packages, you just need to have the following lines in your `mk.conf` file:

```
PKGSRC_COMPILER=        sunpro
ABI=                    64
```

Note: This setting has been tested for the SPARC architecture. Intel and AMD machines need some more work.

3.3.10.4. Common problems

Sometimes, when using **libtool**, `/bin/ksh` crashes with a segmentation fault. The workaround is to use another shell for the configure scripts, for example by installing `shells/bash` and adding the following lines to your `mk.conf`:

```
CONFIG_SHELL=   ${LOCALBASE}/bin/bash
WRAPPER_SHELL=  ${LOCALBASE}/bin/bash
```

Then, rebuild the `devel/libtool-base` package.

Chapter 4.
Using pkgsrc

Basically, there are two ways of using pkgsrc. The first is to only install the package tools and to use binary packages that someone else has prepared. This is the "pkg" in pkgsrc. The second way is to install the "src" of pkgsrc, too. Then you are able to build your own packages, and you can still use binary packages from someone else.

4.1. Using binary packages

On the ftp.NetBSD.org (ftp://ftp.NetBSD.org/) server and its mirrors, there are collections of binary packages, ready to be installed. These binary packages have been built using the default settings for the directories, that is:

- `/usr/pkg` for `LOCALBASE`, where most of the files are installed,
- `/usr/pkg/etc` for configuration files,
- `/var` for `VARBASE`, where those files are installed that may change after installation.

If you cannot use these directories for whatever reasons (maybe because you're not root), you cannot use these binary packages, but have to build the packages yourself, which is explained in Section 3.2.

4.1.1. Finding binary packages

To install binary packages, you first need to know from where to get them. The first place where you should look is on the main pkgsrc FTP server in the directory `/pub/pkgsrc/packages` (ftp://ftp.NetBSD.org/pub/pkgsrc/packages/).

This directory contains binary packages for multiple platforms. First, select your operating system. (Ignore the directories with version numbers attached to it, they just exist for legacy reasons.) Then, select your hardware architecture, and in the third step, the OS version and the "version" of pkgsrc.

In this directory, you often find a file called `bootstrap.tar.gz` which contains the package management tools. If the file is missing, it is likely that your operating system already provides those tools. Download the file and extract it in the `/` directory. It will create the directories `/usr/pkg` (containing the tools for managing binary packages) and `/var/db/pkg` (the database of installed packages).

4.1.2. Installing binary packages

In the directory from the last section, there is a subdirectory called `All/`, which contains all the binary packages that are available for the platform, excluding those that may not be distributed via FTP or CDROM (depending on which medium you are using).

To install packages directly from an FTP or HTTP server, run the following commands in a Bourne-compatible shell (be sure to **su** to root first):

```
# PATH="/usr/pkg/sbin:$PATH"
# PKG_PATH="ftp://ftp.NetBSD.org/pub/pkgsrc/packages/OPSYS/ARCH/VERSIONS/All/"
# export PATH PKG_PATH
```

Instead of URLs, you can also use local paths, for example if you are installing from a set of CDROMs, DVDs or an NFS-mounted repository. If you want to install packages from multiple sources, you can separate them by a semicolon in `PKG_PATH`.

After these preparations, installing a package is very easy:

```
# pkg_add openoffice2
# pkg_add kde-3.5.7
# pkg_add ap2-php5-*
```

Note that any prerequisite packages needed to run the package in question will be installed, too, assuming they are present where you install from.

Adding packages might install vulnerable packages. Thus you should run **pkg_admin audit** regularly, especially after installing new packages, and verify that the vulnerabilities are acceptable for your configuration.

After you've installed packages, be sure to have `/usr/pkg/bin` and `/usr/pkg/sbin` in your `PATH` so you can actually start the just installed program.

4.1.3. Deinstalling packages

To deinstall a package, it does not matter whether it was installed from source code or from a binary package. The **pkg_delete** command does not know it anyway. To delete a package, you can just run **pkg_delete** ***package-name***. The package name can be given with or without version number. Wildcards can also be used to deinstall a set of packages, for example `*emacs*`. Be sure to include them in quotes, so that the shell does not expand them before `pkg_delete` sees them.

The `-r` option is very powerful: it removes all the packages that require the package in question and then removes the package itself. For example:

```
# pkg_delete -r jpeg
```

will remove jpeg and all the packages that used it; this allows upgrading the jpeg package.

4.1.4. Getting information about installed packages

The **pkg_info** shows information about installed packages or binary package files.

4.1.5. Checking for security vulnerabilities in installed packages

The NetBSD Security-Officer and Packages Groups maintain a list of known security vulnerabilities to packages which are (or have been) included in pkgsrc. The list is available from the NetBSD FTP site at ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/vulnerabilities.

Through **pkg_admin fetch-pkg-vulnerabilities**, this list can be downloaded automatically, and a security audit of all packages installed on a system can take place.

There are two components to auditing. The first step, **pkg_admin fetch-pkg-vulnerabilities**, is for downloading the list of vulnerabilities from the NetBSD FTP site. The second step, **pkg_admin audit**, checks to see if any of your installed packages are vulnerable. If a package is vulnerable, you will see output similar to the following:

```
Package samba-2.0.9 has a local-root-shell vulnerability, see
    http://www.samba.org/samba/whatsnew/macroexploit.html
```

You may wish to have the vulnerabilities (ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/vulnerabilities) file downloaded daily so that it remains current. This may be done by adding an appropriate entry to the root users crontab(5) entry. For example the entry

```
# download vulnerabilities file
0 3 * * * /usr/sbin/pkg_admin fetch-pkg-vulnerabilities >/dev/null 2>&1
```

will update the vulnerability list every day at 3AM. You may wish to do this more often than once a day. In addition, you may wish to run the package audit from the daily security script. This may be accomplished by adding the following line to `/etc/security.local`:

```
/usr/sbin/pkg_admin audit
```

4.1.6. Finding if newer versions of your installed packages are in pkgsrc

Install `pkgtools/lintpkgsrc` and run **lintpkgsrc** with the "-i" argument to check if your packages are up-to-date, e.g.

```
% lintpkgsrc -i
...
Version mismatch: 'tcsh' 6.09.00 vs 6.10.00
```

You can then use **make update** to update the package on your system and rebuild any dependencies.

4.1.7. Other administrative functions

The **pkg_admin** executes various administrative functions on the package system.

4.1.8. A word of warning

Please pay very careful attention to the warnings expressed in the pkg_add(1) 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.

The same warning of course applies to every package you install from source when you haven't completely read and understood the source code of the package, the compiler that is used to build the package and all the other tools that are involved.

4.2. Building packages from source

After obtaining pkgsrc, the `pkgsrc` directory now contains a set of packages, organized into categories. You can browse the online index of packages, or run **make readme** from the `pkgsrc` directory to build local `README.html` files for all packages, viewable with any web browser such as `www/lynx` or `www/firefox`.

The default *prefix* for installed packages is `/usr/pkg`. If you wish to change this, you should do so by setting `LOCALBASE` in `mk.conf`. You should not try to use multiple different `LOCALBASE` definitions on the same system (inside a chroot is an exception).

The rest of this chapter assumes that the package is already in pkgsrc. If it is not, see Part II in *The pkgsrc guide* for instructions how to create your own packages.

4.2.1. Requirements

To build packages from source, you need a working C compiler. On NetBSD, you need to install the "comp" and the "text" distribution sets. If you want to build X11-related packages, the "xbase" and "xcomp" distribution sets are required, too.

4.2.2. Fetching distfiles

The first step for building a package is downloading the distfiles (i.e. the unmodified source). If they have not yet been downloaded, pkgsrc will fetch them automatically.

If you have all files that you need in the `distfiles` directory, you don't need to connect. If the distfiles are on CD-ROM, you can mount the CD-ROM on `/cdrom` and add:

```
DISTDIR=/cdrom/pkgsrc/distfiles
```

to your `mk.conf`.

By default a list of distribution sites will be randomly intermixed to prevent huge load on servers which holding popular packages (for example, SourceForge.net mirrors). Thus, every time when you need to fetch yet another distfile all the mirrors will be tried in new (random) order. You can turn this feature off by setting `MASTER_SORT_RANDOM=NO` (for `PKG_DEVELOPER`s it's already disabled).

You can overwrite some of the major distribution sites to fit to sites that are close to your own. By setting one or two variables you can modify the order in which the master sites are accessed. `MASTER_SORT` contains a whitespace delimited list of domain suffixes. `MASTER_SORT_REGEX` is even more flexible, it

contains a whitespace delimited list of regular expressions. It has higher priority than `MASTER_SORT`. Have a look at `pkgsrc/mk/defaults/mk.conf` to find some examples. This may save some of your bandwidth and time.

You can change these settings either in your shell's environment, or, if you want to keep the settings, by editing the `mk.conf` file, and adding the definitions there.

If a package depends on many other packages (such as `meta-pkgs/kde3`), the build process may alternate between periods of downloading source, and compiling. To ensure you have all the source downloaded initially you can run the command:

```
% make fetch-list | sh
```

which will output and run a set of shell commands to fetch the necessary files into the `distfiles` directory. You can also choose to download the files manually.

4.2.3. How to build and install

Once the software has downloaded, any patches will be applied, then it will be compiled for you. This may take some time depending on your computer, and how many other packages the software depends on and their compile time.

> **Note:** If using bootstrap or pkgsrc on a non-NetBSD system, use the pkgsrc **bmake** command instead of "make" in the examples in this guide.

For example, type

```
% cd misc/figlet
% make
```

at the shell prompt to build the various components of the package.

The next stage is to actually install the newly compiled program onto your system. Do this by entering:

```
% make install
```

while you are still in the directory for whatever package you are installing.

Installing the package on your system may require you to be root. However, pkgsrc has a *just-in-time-su* feature, which allows you to only become root for the actual installation step.

That's it, the software should now be installed and setup for use. You can now enter:

```
% make clean
```

to remove the compiled files in the work directory, as you shouldn't need them any more. If other packages were also added to your system (dependencies) to allow your program to compile, you can tidy these up also with the command:

```
% make clean-depends
```

Taking the figlet utility as an example, we can install it on our system by building as shown in Appendix B.

The program is installed under the default root of the packages tree - `/usr/pkg`. Should this not conform to your tastes, set the `LOCALBASE` variable in your environment, and it will use that value as the root of your packages tree. So, to use `/usr/local`, set `LOCALBASE=/usr/local` in your environment. Please note that you should use a directory which is dedicated to packages and not shared with other programs (i.e., do not try and use `LOCALBASE=/usr`). Also, you should not try to add any of your own files or directories (such as `src/`, `obj/`, or `pkgsrc/`) below the `LOCALBASE` tree. This is to prevent possible conflicts between programs and other files installed by the package system and whatever else may have been installed there.

Some packages look in `mk.conf` to alter some configuration options at build time. Have a look at `pkgsrc/mk/defaults/mk.conf` to get an overview of what will be set there by default. Environment variables such as `LOCALBASE` can be set in `mk.conf` to save having to remember to set them each time you want to use pkgsrc.

Occasionally, people want to “look under the covers” to see what is going on when a package is building or being installed. This may be for debugging purposes, or out of simple curiosity. A number of utility values have been added to help with this.

1. If you invoke the make(1) command with `PKG_DEBUG_LEVEL=2`, then a huge amount of information will be displayed. For example,

   ```
   make patch PKG_DEBUG_LEVEL=2
   ```

   will show all the commands that are invoked, up to and including the “patch” stage.

2. If you want to know the value of a certain make(1) definition, then the `VARNAME` definition should be used, in conjunction with the show-var target. e.g. to show the expansion of the make(1) variable `LOCALBASE`:

   ```
   % make show-var VARNAME=LOCALBASE
   /usr/pkg
   %
   ```

If you want to install a binary package that you’ve either created yourself (see next section), that you put into pkgsrc/packages manually or that is located on a remote FTP server, you can use the "bin-install" target. This target will install a binary package - if available - via pkg_add(1), else do a **make package**. The list of remote FTP sites searched is kept in the variable `BINPKG_SITES`, which defaults to ftp.NetBSD.org. Any flags that should be added to pkg_add(1) can be put into `BIN_INSTALL_FLAGS`. See `pkgsrc/mk/defaults/mk.conf` for more details.

A final word of warning: If you set up a system that has a non-standard setting for `LOCALBASE`, be sure to set that before any packages are installed, as you cannot use several directories for the same purpose. Doing so will result in pkgsrc not being able to properly detect your installed packages, and fail miserably. Note also that precompiled binary packages are usually built with the default `LOCALBASE` of `/usr/pkg`, and that you should *not* install any if you use a non-standard `LOCALBASE`.

Chapter 5.
Configuring pkgsrc

The whole pkgsrc system is configured in a single file, usually called `mk.conf`. In which directory pkgsrc looks for that file depends on the installation. On NetBSD, when you use make(1) from the base system, it is in the directory `/etc/`. In all other cases the default location is `${PREFIX}/etc/`, depending on where you told the bootstrap program to install the binary packages.

During the bootstrap, an example configuration file is created. To use that, you have to create the directory `${PREFIX}/etc` and copy the example file there.

The format of the configuration file is that of the usual BSD-style `Makefile`s. The whole pkgsrc configuration is done by setting variables in this file. Note that you can define all kinds of variables, and no special error checking (for example for spelling mistakes) takes place, so you have to try it out to see if it works.

5.1. General configuration

In this section, you can find some variables that apply to all pkgsrc packages. A complete list of the variables that can be configured by the user is available in `mk/defaults/mk.conf`, together with some comments that describe each variable's intent.

- `LOCALBASE`: Where packages will be installed. The default is `/usr/pkg`. Do not mix binary packages with different `LOCALBASE`s!
- `CROSSBASE`: Where "cross" category packages will be installed. The default is `${LOCALBASE}/cross`.
- `X11BASE`: Where X11 is installed on the system. The default is `/usr/X11R7`.
- `DISTDIR`: Where to store the downloaded copies of the original source distributions used for building pkgsrc packages. The default is `${PKGSRCDIR}/distfiles`.
- `PKG_DBDIR`: Where the database about installed packages is stored. The default is `/var/db/pkg`.
- `MASTER_SITE_OVERRIDE`: If set, override the packages' `MASTER_SITES` with this value.
- `MASTER_SITE_BACKUP`: Backup location(s) for distribution files and patch files if not found locally or in `${MASTER_SITES}` or `${PATCH_SITES}` respectively. The defaults are `ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/${DIST_SUBDIR}/` and `ftp://ftp.freebsd.org/pub/FreeBSD/distfiles/${DIST_SUBDIR}/`.
- `BINPKG_SITES`: List of sites carrying binary pkgs. *`rel`* and *`arch`* are replaced with OS release ("2.0", etc.) and architecture ("mipsel", etc.).
- `ACCEPTABLE_LICENSES`: List of acceptable licenses. License names are case-sensitive. Whenever you try to build a package whose license is not in this list, you will get an error message. If the license

condition is simple enough, the error message will include specific instructions on how to change this variable.

5.2. Variables affecting the build process

XXX

- `PACKAGES`: The top level directory for the binary packages. The default is `${PKGSRCDIR}/packages`.
- `WRKOBJDIR`: The top level directory where, if defined, the separate working directories will get created, and symbolically linked to from `${WRKDIR}` (see below). This is useful for building packages on several architectures, then `${PKGSRCDIR}` can be NFS-mounted while `${WRKOBJDIR}` is local to every architecture. (It should be noted that `PKGSRCDIR` should not be set by the user — it is an internal definition which refers to the root of the pkgsrc tree. It is possible to have many pkgsrc tree instances.)
- `LOCALPATCHES`: Directory for local patches that aren't part of pkgsrc. See Section 11.3 for more information.
- `PKGMAKECONF`: Location of the `mk.conf` file used by a package's BSD-style Makefile. If this is not set, `MAKECONF` is set to `/dev/null` to avoid picking up settings used by builds in `/usr/src`.
- `DEPENDS_TARGET`: By default, dependencies are only installed, and no binary package is created for them. You can set this variable to `package-install` to automatically create binary packages after installing dependencies. Please note that the `PKGSRC_KEEP_BIN_PKGS` can be set to `yes` to preserve binary packages when using the `install` as well.

5.3. Variables affecting the installation process

Most packages support installation into a subdirectory of `WRKDIR`. This allows a package to be built, before the actual filesystem is touched. DESTDIR support exists in two variations:

- Basic DESTDIR support means that the package installation and packaging is still run as root.
- Full DESTDIR support can run the complete build, installation and packaging as normal user. Root privileges are only needed to add packages.

DESTDIR support is now the default. To switch back to non-DESTDIR, you can set `USE_DESTDIR=no`; this setting will be deprecated though, so it's preferable to convert a package to DESTDIR instead.

With basic DESTDIR support, **`make clean`** needs to be run as root.

Considering the `foo/bar` package, DESTDIR full support can be tested using the following commands

```
$ id
uid=1000(myusername) gid=100(users) groups=100(users),0(wheel)
$ mkdir $HOME/packages
$ cd $PKGSRCDIR/foo/bar
```

Verify DESTDIR full support, no root privileges should be needed

```
$ make stage-install
```

Create a package without root privileges

```
$ make PACKAGES=$HOME/packages package
```

For the following command, you must be able to gain root privileges using su(1)

```
$ make PACKAGES=$HOME/packages install
```

Then, as a simple user

```
$ make clean
```

5.4. Selecting and configuring the compiler

5.4.1. Selecting the compiler

By default, pkgsrc will use GCC to build packages. This may be overridden by setting the following variables in /etc/mk.conf:

PKGSRC_COMPILER:

This is a list of values specifying the chain of compilers to invoke when building packages. Valid values are:

- ccc: Compaq C Compilers (Tru64)
- ccache: compiler cache (chainable)
- clang: Clang C and Objective-C compiler
- distcc: distributed C/C++ (chainable)
- f2c: Fortran 77 to C compiler (chainable)
- icc: Intel C++ Compiler (Linux)
- ido: SGI IRIS Development Option cc (IRIX 5)
- gcc: GNU C/C++ Compiler
- hp: HP-UX C/aC++ compilers
- mipspro: Silicon Graphics, Inc. MIPSpro (n32/n64)
- mipspro-ucode: Silicon Graphics, Inc. MIPSpro (o32)
- sunpro: Sun Microsystems, Inc. WorkShip/Forte/Sun ONE Studio
- xlc: IBM's XL C/C++ compiler suite (Darwin/MacOSX)

The default is "gcc". You can use ccache and/or distcc with an appropriate PKGSRC_COMPILER setting, e.g. "ccache gcc". This variable should always be terminated with a value for a real compiler. Note that only one real compiler should be listed (e.g. "sunpro gcc" is not allowed).

GCC_REQD:

This specifies the minimum version of GCC to use when building packages. If the system GCC doesn't satisfy this requirement, then pkgsrc will build and install one of the GCC packages to use instead.

5.4.2. Additional flags to the compiler (CFLAGS)

If you wish to set the CFLAGS variable, please make sure to use the += operator instead of the = operator:

```
CFLAGS+=        -your -flags
```

Using CFLAGS= (i.e. without the "+") may lead to problems with packages that need to add their own flags. You may want to take a look at the devel/cpuflags package if you're interested in optimization specifically for the current CPU.

5.4.3. Additional flags to the linker (LDFLAGS)

If you want to pass flags to the linker, both in the configure step and the build step, you can do this in two ways. Either set LDFLAGS or LIBS. The difference between the two is that LIBS will be appended to the command line, while LDFLAGS come earlier. LDFLAGS is pre-loaded with rpath settings for ELF machines depending on the setting of USE_IMAKE or the inclusion of mk/x11.buildlink3.mk. As with CFLAGS, if you do not wish to override these settings, use the += operator:

```
LDFLAGS+=        -your -linkerflags
```

5.5. Developer/advanced settings

- PKG_DEVELOPER: Run some sanity checks that package developers want:
  - make sure patches apply with zero fuzz
  - run check-shlibs to see that all binaries will find their shared libs.
- PKG_DEBUG_LEVEL: The level of debugging output which is displayed whilst making and installing the package. The default value for this is 0, which will not display the commands as they are executed (normal, default, quiet operation); the value 1 will display all shell commands before their invocation, and the value 2 will display both the shell commands before their invocation, and their actual execution progress with **set -x** will be displayed.

5.6. Selecting Build Options

Some packages have build time options, usually to select between different dependencies, enable optional support for big dependencies or enable experimental features.

To see which options, if any, a package supports, and which options are mutually exclusive, run **make show-options**, for example:

```
The following options are supported by this package:
    ssl      Enable SSL support.
Exactly one of the following gecko options is required:
    firefox  Use firefox as gecko rendering engine.
    mozilla  Use mozilla as gecko rendering engine.
At most one of the following database options may be selected:
    mysql    Enable support for MySQL database.
    pgsql    Enable support for PostgreSQL database.

These options are enabled by default: firefox
These options are currently enabled: mozilla ssl
```

The following variables can be defined in `mk.conf` to select which options to enable for a package: `PKG_DEFAULT_OPTIONS`, which can be used to select or disable options for all packages that support them, and `PKG_OPTIONS.`*`pkgbase`*, which can be used to select or disable options specifically for package *`pkgbase`*. Options listed in these variables are selected, options preceded by "-" are disabled. A few examples:

```
$ grep "PKG.*OPTION" mk.conf
PKG_DEFAULT_OPTIONS=    -arts -dvdread -esound
PKG_OPTIONS.kdebase=    debug -sasl
PKG_OPTIONS.apache=     suexec
```

It is important to note that options that were specifically suggested by the package maintainer must be explicitly removed if you do not wish to include the option. If you are unsure you can view the current state with **make show-options**.

The following settings are consulted in the order given, and the last setting that selects or disables an option is used:

1. the default options as suggested by the package maintainer
2. the options implied by the settings of legacy variables (see below)
3. `PKG_DEFAULT_OPTIONS`
4. `PKG_OPTIONS.`*`pkgbase`*

For groups of mutually exclusive options, the last option selected is used, all others are automatically disabled. If an option of the group is explicitly disabled, the previously selected option, if any, is used. It is an error if no option from a required group of options is selected, and building the package will fail.

Before the options framework was introduced, build options were selected by setting a variable (often named `USE_`*`FOO`*) in `mk.conf` for each option. To ease transition to the options framework for the user, these legacy variables are converted to the appropriate options setting (`PKG_OPTIONS.`*`pkgbase`*) automatically. A warning is issued to prompt the user to update `mk.conf` to use the options framework directly. Support for the legacy variables will be removed eventually.

Chapter 6.
Creating binary packages

6.1. Building a single binary package

Once you have built and installed a package, you can create a *binary package* which can be installed on another system with pkg_add(1). This saves having to build the same package on a group of hosts and wasting CPU time. It also provides a simple means for others to install your package, should you distribute it.

To create a binary package, change into the appropriate directory in pkgsrc, and run **make package**:

```
# cd misc/figlet
# make package
```

This will build and install your package (if not already done), and then build a binary package from what was installed. You can then use the **pkg_*** tools to manipulate it. Binary packages are created by default in `/usr/pkgsrc/packages`, in the form of a gzipped tar file. See Section B.2 for a continuation of the above `misc/figlet` example.

See Chapter 21 for information on how to submit such a binary package.

6.2. Settings for creation of binary packages

See Section 17.17.

Chapter 7.
Creating binary packages for everything in pkgsrc (bulk builds)

For a number of reasons you may want to build binary packages for a large selected set of packages in pkgsrc or even for all pkgsrc packages. For instance, when you have multiple machines that should run the same software, it is wasted time if they all build their packages themselves from source. Or you may want to build a list of packages you want and check them before deploying onto production system. There is a way of getting a set of binary packages: The bulk build system, or pbulk ("p" stands for "parallel). This chapter describes how to set it up so that the packages are most likely to be usable later.

7.1. Preparations

First of all, you have to decide whether you build all packages or a limited set of them. Full bulk builds usually consume a lot more resources, both space and time, than builds for some practical sets of packages. There exists a number of particularly heavy packages that are not actually interesting to a wide audience. For a limited bulk builds you need to make a list of packages you want to build. Note, that all their dependencies will be built, so you don't need to track them manually.

During bulk builds various packages are installed and deinstalled in `/usr/pkg` (or whatever `LOCALBASE` is), so make sure that you don't need any package during the builds. Essentially, you should provide fresh system, either a chroot environment or something even more restrictive, depending on what the operating system provides, or dedicate the whole physical machine. As a useful side effect this makes sure that bulk builds cannot break anything in your system. There have been numerous cases where certain packages tried to install files outside the `LOCALBASE` or wanted to edit some files in `/etc`.

7.2. Running a pbulk-style bulk build

Running a pbulk-style bulk build works roughly as follows:

- First, build the pbulk infrastructure in a fresh pkgsrc location.
- Then, build each of the packages from a clean installation directory using the infrastructure.

7.2.1. Configuration

To simplify configuration we provide helper script `mk/pbulk/pbulk.sh`.

In order to use it, prepare a clear system (real one, chroot environment, jail, zone, virtual machine). Configure network access to fetch distribution files. Create user with name "pbulk".

Fetch and extract pkgsrc. Use a command like one of these:

```
# (cd /usr && ftp -o - http://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.gz | tar -zxf-)
# (cd /usr && fetch -o - http://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc.tar.gz | tar -zxf-)
# (cd /usr && cvs -Q -z3 -d anoncvs@anoncvs.NetBSD.org:/cvsroot get -P pkgsrc)
```

Or any other way that fits (e.g., curl, wget).

Deploy and configure pbulk tools, e.g.:

```
# sh pbulk.sh -n # native (NetBSD)
# sh pbulk.sh -n -c mk.conf.frag # native, apply settings from given mk.conf fragment
# sh pbulk.sh -nlc mk.conf.frag # native, apply settings, configure for limited build
```

Note: `mk.conf.frag` is a fragment of `mk.conf` that contains settings you want to apply to packages you build. For instance,

```
PKG_DEVELOPER=  yes  # perform more checks
X11_TYPE=  modular  # use pkgsrc X11
SKIP_LICENSE_CHECK= yes  # accept all licences (useful when building all packages)
```

If configured for limited list, replace the list in `/usr/pbulk/etc/pbulk.list` with your list of packages one per line without empty lines or comments. E.g.:

```
www/firefox
mail/thunderbird
misc/libreoffice4
```

At this point you can also review configuration in `/usr/pbulk/etc` and make final amendments, if wanted.

Start it:

```
# /usr/pbulk/bin/bulkbuild
```

After it finishes, you'll have `/mnt` filled with distribution files, binary packages, and reports, plain text summary in `/mnt/bulklog/meta/report.txt`

Note: The `pbulk.sh` script does not cover all possible use cases. While being ready to run, it serves as a good starting point to understand and build more complex setups. The script is kept small enough for better understanding.

Note: The `pbulk.sh` script supports running unprivileged bulk build and helps configuring distributed bulk builds.

7.3. Requirements of a full bulk build

A complete bulk build requires lots of disk space. Some of the disk space can be read-only, some other must be writable. Some can be on remote filesystems (such as NFS) and some should be local. Some can be temporary filesystems, others must survive a sudden reboot.

- 40 GB for the distfiles (read-write, remote, temporary)
- 30 GB for the binary packages (read-write, remote, permanent)
- 1 GB for the pkgsrc tree (read-only, remote, permanent)
- 5 GB for `LOCALBASE` (read-write, local, temporary)
- 10 GB for the log files (read-write, remote, permanent)
- 5 GB for temporary files (read-write, local, temporary)

7.4. Creating a multiple CD-ROM packages collection

After your pkgsrc bulk-build has completed, you may wish to create a CD-ROM set of the resulting binary packages to assist in installing packages on other machines. The `pkgtools/cdpack` package provides a simple tool for creating the ISO 9660 images. **cdpack** arranges the packages on the CD-ROMs in a way that keeps all the dependencies for a given package on the same CD as that package.

7.4.1. Example of cdpack

Complete documentation for cdpack is found in the cdpack(1) man page. The following short example assumes that the binary packages are left in `/usr/pkgsrc/packages/All` and that sufficient disk space exists in `/u2` to hold the ISO 9660 images.

```
# mkdir /u2/images
# pkg_add /usr/pkgsrc/packages/All/cdpack
# cdpack /usr/pkgsrc/packages/All /u2/images
```

If you wish to include a common set of files (`COPYRIGHT`, `README`, etc.) on each CD in the collection, then you need to create a directory which contains these files. e.g.

```
# mkdir /tmp/common
# echo "This is a README" > /tmp/common/README
# echo "Another file" > /tmp/common/COPYING
# mkdir /tmp/common/bin
# echo "#!/bin/sh" > /tmp/common/bin/myscript
# echo "echo Hello world" >> /tmp/common/bin/myscript
# chmod 755 /tmp/common/bin/myscript
```

Now create the images:

```
# cdpack -x /tmp/common /usr/pkgsrc/packages/All /u2/images
```

Each image will contain `README`, `COPYING`, and `bin/myscript` in their root directories.

Chapter 8. *Directory layout of the installed files*

The files that are installed by pkgsrc are organized in a way that is similar to what you find in the `/usr` directory of the base system. But some details are different. This is because pkgsrc initially came from FreeBSD and had adopted its file system hierarchy. Later it was largely influenced by NetBSD. But no matter which operating system you are using pkgsrc with, you can expect the same layout for pkgsrc.

There are mainly four root directories for pkgsrc, which are all configurable in the `bootstrap/bootstrap` script. When pkgsrc has been installed as root, the default locations are:

```
LOCALBASE=              /usr/pkg
PKG_SYSCONFBASE=        /usr/pkg/etc
VARBASE=                /var
PKG_DBDIR=              /var/db/pkg
```

In unprivileged mode (when pkgsrc has been installed as any other user), the default locations are:

```
LOCALBASE=              ${HOME}/pkg
PKG_SYSCONFBASE=        ${HOME}/pkg/etc
VARBASE=                ${HOME}/pkg/var
PKG_DBDIR=              ${HOME}/pkg/var/db/pkg
```

What these four directories are for, and what they look like is explained below.

- `LOCALBASE` corresponds to the `/usr` directory in the base system. It is the "main" directory where the files are installed and contains the well-known subdirectories like `bin`, `include`, `lib`, `share` and `sbin`.
- `VARBASE` corresponds to `/var` in the base system. Some programs (especially games, network daemons) need write access to it during normal operation.
- `PKG_SYSCONFDIR` corresponds to `/etc` in the base system. It contains configuration files of the packages, as well as pkgsrc's `mk.conf` itself.

8.1. File system layout in `${LOCALBASE}`

The following directories exist in a typical pkgsrc installation in `${LOCALBASE}`.

`bin`

Contains executable programs that are intended to be directly used by the end user.

`emul`

Contains files for the emulation layers of various other operating systems, especially for NetBSD.

`etc` (the usual location of `${PKG_SYSCONFDIR}`)

Contains the configuration files.

`include`

Contains headers for the C and C++ programming languages.

`info`

Contains GNU info files of various packages.

`lib`

Contains shared and static libraries.

`libdata`

Contains data files that don't change after installation. Other data files belong into `${VARBASE}`.

`libexec`

Contains programs that are not intended to be used by end users, such as helper programs or network daemons.

`libexec/cgi-bin`

Contains programs that are intended to be executed as CGI scripts by a web server.

`man` (the usual value of `${PKGMANDIR}`)

Contains brief documentation in form of manual pages.

`sbin`

Contains programs that are intended to be used only by the super-user.

`share`

Contains platform-independent data files that don't change after installation.

`share/doc`

Contains documentation files provided by the packages.

`share/examples`

Contains example files provided by the packages. Among others, the original configuration files are saved here and copied to `${PKG_SYSCONFDIR}` during installation.

`share/examples/rc.d`

Contains the original files for rc.d scripts.

`var` (the usual location of `${VARBASE}`)

Contains files that may be modified after installation.

8.2. File system layout in `${VARBASE}`

`db/pkg` (the usual location of `${PKG_DBDIR}`)

Contains information about the currently installed packages.

`games`

Contains highscore files.

`log`

Contains log files.

`run`

Contains informational files about daemons that are currently running.

Chapter 9.
Frequently Asked Questions

This section contains hints, tips & tricks on special things in pkgsrc that we didn't find a better place for in the previous chapters, and it contains items for both pkgsrc users and developers.

9.1. Are there any mailing lists for pkg-related discussion?

The following mailing lists may be of interest to pkgsrc users:

- pkgsrc-users (http://www.NetBSD.org/mailinglists/index.html#pkgsrc-users): This is a general purpose list for most issues regarding pkgsrc, regardless of platform, e.g. soliciting user help for pkgsrc configuration, unexpected build failures, using particular packages, upgrading pkgsrc installations, questions regarding the pkgsrc release branches, etc. General announcements or proposals for changes that impact the pkgsrc user community, e.g. major infrastructure changes, new features, package removals, etc., may also be posted.
- pkgsrc-bulk (http://www.NetBSD.org/mailinglists/index.html#pkgsrc-bulk): A list where the results of pkgsrc bulk builds are sent and discussed.
- pkgsrc-changes (http://www.NetBSD.org/mailinglists/index.html#pkgsrc-changes): This list is for those who are interested in getting a commit message for every change committed to pkgsrc. It is also available in digest form, meaning one daily message containing all commit messages for changes to the package source tree in that 24 hour period.

To subscribe, do:

```
% echo subscribe listname | mail majordomo@NetBSD.org
```

Archives for all these mailing lists are available from http://mail-index.NetBSD.org/.

9.2. Utilities for package management (pkgtools)

The directory `pkgsrc/pkgtools` contains a number of useful utilities for both users and developers of pkgsrc. This section attempts only to make the reader aware of the utilities and when they might be useful, and not to duplicate the documentation that comes with each package.

Utilities used by pkgsrc (automatically installed when needed):

- `pkgtools/x11-links`: Symlinks for use by buildlink.

OS tool augmentation (automatically installed when needed):

- `pkgtools/digest`: Calculates various kinds of checksums (including SHA1).

- `pkgtools/libnbcompat`: Compatibility library for pkgsrc tools.
- `pkgtools/mtree`: Installed on non-BSD systems due to lack of native mtree.
- `pkgtools/pkg_install`: Up-to-date replacement for `/usr/sbin/pkg_install`, or for use on operating systems where pkg_install is not present.

Utilities used by pkgsrc (not automatically installed):

- `pkgtools/pkg_tarup`: Create a binary package from an already-installed package. Used by **make replace** to save the old package.
- `pkgtools/dfdisk`: Adds extra functionality to pkgsrc, allowing it to fetch distfiles from multiple locations. It currently supports the following methods: multiple CD-ROMs and network FTP/HTTP connections.
- `devel/cpuflags`: Determine the best compiler flags to optimise code for your current CPU and compiler.

Utilities for keeping track of installed packages, being up to date, etc:

- `pkgtools/pkgin`: A package update tool similar to apt(1). Download, install, and upgrade binary packages easily.
- `pkgtools/pkg_chk`: Reports on packages whose installed versions do not match the latest pkgsrc entries.
- `pkgtools/pkgdep`: Makes dependency graphs of packages, to aid in choosing a strategy for updating.
- `pkgtools/pkgdepgraph`: Makes graphs from the output of `pkgtools/pkgdep` (uses graphviz).
- `pkgtools/pkglint`: The pkglint(1) program checks a pkgsrc entry for errors.
- `pkgtools/lintpkgsrc`: The lintpkgsrc(1) program does various checks on the complete pkgsrc system.
- `pkgtools/pkgsurvey`: Report what packages you have installed.

Utilities for people maintaining or creating individual packages:

- `pkgtools/pkgdiff`: Automate making and maintaining patches for a package (includes pkgdiff, pkgvi, mkpatches, etc.).
- `pkgtools/url2pkg`: Aids in converting to pkgsrc.

Utilities for people maintaining pkgsrc (or: more obscure pkg utilities)

- `pkgtools/pkg_comp`: Build packages in a chrooted area.
- `pkgtools/libkver`: Spoof kernel version for chrooted cross builds.

9.3. How to use pkgsrc as non-root

If you want to use pkgsrc as non-root user, you can set some variables to make pkgsrc work under these conditions. At the very least, you need to set `UNPRIVILEGED` to "yes"; this will turn on unprivileged mode and set multiple related variables to allow installation of packages as non-root.

In case the defaults are not enough, you may want to tune some other variables used. For example, if the automatic user/group detection leads to incorrect values (or not the ones you would like to use), you can change them by setting `UNPRIVILEGED_USER` and `UNPRIVILEGED_GROUP` respectively.

As regards bootstrapping, please note that the **bootstrap** script will ease non-root configuration when given the "--ignore-user-check" flag, as it will choose and use multiple default directories under `~/pkg` as the installation targets. These directories can be overridden by the "--prefix" flag provided by the script, as well as some others that allow finer tuning of the tree layout.

9.4. How to resume transfers when fetching distfiles?

By default, resuming transfers in pkgsrc is disabled, but you can enable this feature by adding the option `PKG_RESUME_TRANSFERS=YES` into `mk.conf`. If, during a fetch step, an incomplete distfile is found, pkgsrc will try to resume it.

You can also use a different program than the platform default program by changing the `FETCH_USING` variable. You can specify the program by using of ftp, fetch, wget or curl. Alternatively, fetching can be disabled by using the value manual. A value of custom disables the system defaults and dependency tracking for the fetch program. In that case you have to provide `FETCH_CMD`, `FETCH_BEFORE_ARGS`, `FETCH_RESUME_ARGS`, `FETCH_OUTPUT_ARGS`, `FETCH_AFTER_ARGS`.

For example, if you want to use `wget` to download, you'll have to use something like:

```
FETCH_USING= wget
```

9.5. How can I install/use modular X.org from pkgsrc?

If you want to use modular X.org from pkgsrc instead of your system's own X11 (`/usr/X11R6`, `/usr/openwin`, ...) you will have to add the following line into `mk.conf`:

```
X11_TYPE=modular
```

> **Note:** The DragonFly operating system defaults to using modular X.org from pkgsrc.

9.6. 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. in Amdahl, the machine "orpheus.amdahl.com" is one of the firewalls, and it uses port 80 as the proxy port number. So the proxy environment variables are:

```
ftp_proxy=ftp://orpheus.amdahl.com:80/
http_proxy=http://orpheus.amdahl.com:80/
```

9.7. How to fetch files from HTTPS sites

Some fetch tools are not prepared to support HTTPS by default (for example, the one in NetBSD 6.0), or the one installed by the pkgsrc bootstrap (to avoid an openssl dependency that low in the dependency graph).

Usually you won't notice, because distribution files are mirrored weekly to "ftp.NetBSD.org", but that might not be often enough if you are following pkgsrc-current. In that case, set `FETCH_USING` in your `mk.conf` file to "curl" or "wget", which are both compiled with HTTPS support by default. Of course, these tools need to be installed before you can use them this way.

9.8. 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:

- `${LOCALBASE}/bin/ftp`
- `/usr/bin/ftp`

On a default NetBSD installation, 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 `mk.conf` file: `PASSIVE_FETCH=1`.

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

9.9. How to fetch all distfiles at once

You would like to download all the distfiles in a single batch from work or university, where you can't run a **make fetch**. There is an archive of distfiles on ftp.NetBSD.org (ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/), but downloading the entire directory may not be appropriate.

The answer here is to do a **make fetch-list** in `/usr/pkgsrc` or one of its subdirectories, carry the resulting list to your machine at work/school and use it there. If you don't have a NetBSD-compatible ftp(1) (like tnftp) at work, don't forget to set `FETCH_CMD` to something that fetches a URL:

At home:

```
% cd /usr/pkgsrc
% make fetch-list FETCH_CMD=wget DISTDIR=/tmp/distfiles >/tmp/fetch.sh
% scp /tmp/fetch.sh work:/tmp
```

At work:

```
% sh /tmp/fetch.sh
```

then tar up `/tmp/distfiles` and take it home.

If you have a machine running NetBSD, and you want to get *all* distfiles (even ones that aren't for your machine architecture), you can do so by using the above-mentioned **make fetch-list** approach, or fetch the distfiles directly by running:

```
% make mirror-distfiles
```

If you even decide to ignore `NO_{SRC,BIN}_ON_{FTP,CDROM}`, then you can get everything by running:

```
% make fetch NO_SKIP=yes
```

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

When compiling the `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 (nroff, ...) from the NetBSD base distribution on your machine. It is recommended to do that to format man pages.

In the case of the `pkgtools/pkg_install` package, you can get away with setting `NOMAN=YES` either in the environment or in `mk.conf`.

9.11. What does "Could not find bsd.own.mk" mean?

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

```
# cd /
# tar --unlink -zxvpf .../comp.tgz
```

`comp.tgz` is part of every NetBSD release. Get the one that corresponds to your release (determine via **uname -r**).

9.12. Using 'sudo' with pkgsrc

When installing packages as non-root user and using the just-in-time su(1) feature of pkgsrc, it can become annoying to type in the root password for each required package installed. To avoid this, the sudo package can be used, which does password caching over a limited time. To use it, install sudo (either as binary package or from `security/sudo`) and then put the following into your `mk.conf`, somewhere *after* the definition of the `LOCALBASE` variable:

```
.if exists(${LOCALBASE}/bin/sudo)
SU_CMD=        ${LOCALBASE}/bin/sudo /bin/sh -c
.endif
```

9.13. How do I change the location of configuration files?

As the system administrator, you can choose where configuration files are installed. The default settings make all these files go into `${PREFIX}/etc` or some of its subdirectories; this may be suboptimal depending on your expectations (e.g., a read-only, NFS-exported `PREFIX` with a need of per-machine configuration of the provided packages).

In order to change the defaults, you can modify the `PKG_SYSCONFBASE` variable (in `mk.conf`) to point to your preferred configuration directory; some common examples include `/etc` or `/etc/pkg`.

Furthermore, you can change this value on a per-package basis by setting the `PKG_SYSCONFDIR.${PKG_SYSCONFVAR}` variable. `PKG_SYSCONFVAR`'s value usually matches the name of the package you would like to modify, that is, the contents of `PKGBASE`.

Note that after changing these settings, you must rebuild and reinstall any affected packages.

9.14. Automated security checks

Please be aware that there can often be bugs in third-party software, and some of these bugs can leave a machine vulnerable to exploitation by attackers. In an effort to lessen the exposure, the NetBSD packages team maintains a database of known-exploits to packages which have at one time been included in pkgsrc. The database can be downloaded automatically, and a security audit of all packages installed on a system can take place. To do this, refer to the following two tools (installed as part of the `pkgtools/pkg_install` package):

1. **pkg_admin fetch-pkg-vulnerabilities**, an easy way to download a list of the security vulnerabilities information. This list is kept up to date by the pkgsrc security team, and is distributed from the NetBSD ftp server:

   ftp://ftp.NetBSD.org/pkgsrc/distfiles/pkg-vulnerabilities

2. **pkg_admin audit**, an easy way to audit the current machine, checking each known vulnerability. If a vulnerable package is installed, it will be shown by output to stdout, including a description of the type of vulnerability, and a URL containing more information.

Use of these tools is strongly recommended! After "pkg_install" is installed, please read the package's message, which you can get by running **`pkg_info -D pkg_install`**.

If this package is installed, pkgsrc builds will use it to perform a security check before building any package. See Section 5.2 for ways to control this check.

9.15. Why do some packages ignore my `CFLAGS`?

When you add your own preferences to the `CFLAGS` variable in your `mk.conf`, these flags are passed in environment variables to the `./configure` scripts and to make(1). Some package authors ignore the `CFLAGS` from the environment variable by overriding them in the `Makefile`s of their package.

Currently there is no solution to this problem. If you really need the package to use your `CFLAGS` you should run **make patch** in the package directory and then inspect any `Makefile` and `Makefile.in` for whether they define `CFLAGS` explicitly. Usually you can remove these lines. But be aware that some

"smart" programmers write so bad code that it only works for the specific combination of `CFLAGS` they have chosen.

9.16. A package does not build. What shall I do?

1. Make sure that your copy of pkgsrc is consistent. A case that occurs often is that people only update pkgsrc in parts, because of performance reasons. Since pkgsrc is one large system, not a collection of many small systems, there are sometimes changes that only work when the whole pkgsrc tree is updated.
2. Make sure that you don't have any CVS conflicts. Search for "<<<<<<" or ">>>>>>" in all your pkgsrc files.
3. Make sure that you don't have old copies of the packages extracted. Run **make clean clean-depends** to verify this.
4. If the problem still exists, write a mail to the `pkgsrc-users` mailing list.

9.17. What does "Makefile appears to contain unresolved cvs/rcs/??? merge conflicts" mean?

You have modified a file from pkgsrc, and someone else has modified that same file afterwards in the CVS repository. Both changes are in the same region of the file, so when you updated pkgsrc, the `cvs` command marked the conflicting changes in the file. Because of these markers, the file is no longer a valid `Makefile`.

Have a look at that file, and if you don't need your local changes anymore, you can remove that file and run **cvs -q update -dP** in that directory to download the current version.

II. The pkgsrc developer's guide

This part of the book deals with creating and modifying packages. It starts with a "HOWTO"-like guide on creating a new package. The remaining chapters are more like a reference manual for pkgsrc.

Chapter 10.
Creating a new pkgsrc package from scratch

When you find a package that is not yet in pkgsrc, you most likely have a URL from where you can download the source code. Starting with this URL, creating a package involves only a few steps.

1. First, install the packages `pkgtools/url2pkg` and `pkgtools/pkglint`.
2. Then, choose one of the top-level directories as the category in which you want to place your package. You can also create a directory of your own (maybe called `local`). In that category directory, create another directory for your package and change into it.
3. Run the program **url2pkg**, which will ask you for a URL. Enter the URL of the distribution file (in most cases a `.tar.gz` file) and watch how the basic ingredients of your package are created automatically. The distribution file is extracted automatically to fill in some details in the `Makefile` that would otherwise have to be done manually.
4. Examine the extracted files to determine the dependencies of your package. Ideally, this is mentioned in some `README` file, but things may differ. For each of these dependencies, look where it exists in pkgsrc, and if there is a file called `buildlink3.mk` in that directory, add a line to your package `Makefile` which includes that file just before the last line. If the `buildlink3.mk` file does not exist, it must be created first. The `buildlink3.mk` file makes sure that the package's include files and libraries are provided.

   If you just need binaries from a package, add a `DEPENDS` line to the Makefile, which specifies the version of the dependency and where it can be found in pkgsrc. This line should be placed in the third paragraph. If the dependency is only needed for building the package, but not when using it, use `BUILD_DEPENDS` instead of `DEPENDS`. Your package may then look like this:

   ```
   [...]

   BUILD_DEPENDS+= libxslt-[0-9]*:../../textproc/libxslt
   DEPENDS+=       screen-[0-9]*:../../misc/screen
   DEPENDS+=       screen>=4.0:../../misc/screen

   [...]

   .include "../../category/package/buildlink3.mk"
   .include "../../devel/glib2/buildlink3.mk"
   .include "../../mk/bsd.pkg.mk"
   ```
5. Run **pkglint** to see what things still need to be done to make your package a "good" one. If you don't know what pkglint's warnings want to tell you, try **pkglint --explain** or **pkglint -e**, which outputs additional explanations.

6. In many cases the package is not yet ready to build. You can find instructions for the most common cases in the next section, Section 10.1. After you have followed the instructions over there, you can hopefully continue here.
7. Run **bmake clean** to clean the working directory from the extracted files. Besides these files, a lot of cache files and other system information has been saved in the working directory, which may become wrong after you edited the `Makefile`.
8. Now, run **bmake** to build the package. For the various things that can go wrong in this phase, consult Chapter 19.
9. When the package builds fine, the next step is to install the package. Run **bmake install** and hope that everything works.
10. Up to now, the file `PLIST`, which contains a list of the files that are installed by the package, is nearly empty. Run **bmake print-PLIST >PLIST** to generate a probably correct list. Check the file using your preferred text editor to see if the list of files looks plausible.
11. Run **pkglint** again to see if the generated `PLIST` contains garbage or not.
12. When you ran **bmake install**, the package has been registered in the database of installed files, but with an empty list of files. To fix this, run **bmake deinstall** and **bmake install** again. Now the package is registered with the list of files from `PLIST`.
13. Run **bmake package** to create a binary package from the set of installed files.

10.1. Common types of packages

10.1.1. Perl modules

Simple Perl modules are handled automatically by **url2pkg**, including dependencies.

10.1.2. KDE3 applications

KDE3 applications should always include `meta-pkgs/kde3/kde3.mk`, which contains numerous settings that are typical of KDE3 packages.

10.1.3. Python modules and programs

Python modules and programs packages are easily created using a set of predefined variables.

If some Python versions are not supported by the software, set the `PYTHON_VERSIONS_INCOMPATIBLE` variable to the Python versions that are not supported, e.g.

```
PYTHON_VERSIONS_INCOMPATIBLE=       26
```

If the packaged software is a Python module, include "`../../lang/python/extension.mk`". In this case, the package directory should be called "py-software" and `PKGNAME` should be set to "${PYPKGPREFIX}-${DISTNAME}", e.g.

```
DISTNAME=   foopymodule-1.2.10
PKGNAME=    ${PYPKGPREFIX}-${DISTNAME}
```

If it is an application, include "`../../lang/python/application.mk`". In order to correctly set the path to the Python interpreter, use the `REPLACE_PYTHON` variable and set it to the list of files (paths relative to `WRKSRC`) that must be corrected. For example:

```
REPLACE_PYTHON=   *.py
```

Most Python packages use either "distutils" or easy-setup ("eggs"). If the software uses "distutils", include "`../../lang/python/distutils.mk`". so pkgsrc will use this framework. "distutils" uses a script called `setup.py`, if the "distutils" driver is not called `setup.py`, set the `PYSETUP` variable to the name of the script.

Otherwise, if the packaged software is egg-aware, you only need to include "`../../lang/python/egg.mk`".

Some Python modules have separate distributions for Python-2.x and Python-3.x support. In pkgsrc this is handled by the `versioned_dependencies.mk` file. Set `PYTHON_VERSIONED_DEPENDENCIES` to the list of packages that should be depended upon and include "`../../lang/python/versioned_dependencies.mk`", then the pkgsrc infrastructure will depend on the appropriate package version. For example:

```
PYTHON_VERSIONED_DEPENDENCIES=dateutil dns
```

Look inside `versioned_dependencies.mk` for a list of supported packages.

10.2. Examples

10.2.1. How the www/nvu package came into pkgsrc

10.2.1.1. The initial package

Looking at the file `pkgsrc/doc/TODO`, I saw that the "nvu" package has not yet been imported into pkgsrc. As the description says it has to do with the web, the obvious choice for the category is "www".

```
$ mkdir www/nvu
$ cd www/nvu
```

The web site says that the sources are available as a tar file, so I fed that URL to the **url2pkg** program:

```
$ url2pkg http://cvs.nvu.com/download/nvu-1.0-sources.tar.bz2
```

My editor popped up, and I added a `PKGNAME` line below the `DISTNAME` line, as the package name should not have the word "sources" in it. I also filled in the `MAINTAINER`, `HOMEPAGE` and `COMMENT` fields. Then the package `Makefile` looked like that:

```
# $NetBSD: pkgsrc.ps,v 1.90 2015/03/07 21:32:40 tnn Exp $
#

DISTNAME=       nvu-1.0-sources
PKGNAME=        nvu-1.0
CATEGORIES=     www
MASTER_SITES=   http://cvs.nvu.com/download/
EXTRACT_SUFX=   .tar.bz2

MAINTAINER=     rillig@NetBSD.org
HOMEPAGE=       http://cvs.nvu.com/
COMMENT=        Web Authoring System

# url2pkg-marker (please do not remove this line.)
.include "../../mk/bsd.pkg.mk"
```

Then, I quit the editor and watched pkgsrc downloading a large source archive:

```
url2pkg> Running "make makesum" ...
=> Required installed package digest>=20010302: digest-20060826 found
=> Fetching nvu-1.0-sources.tar.bz2
Requesting http://cvs.nvu.com/download/nvu-1.0-sources.tar.bz2
100% |***********************************| 28992 KB  150.77 KB/s00:00 ETA
29687976 bytes retrieved in 03:12 (150.77 KB/s)
url2pkg> Running "make extract" ...
=> Required installed package digest>=20010302: digest-20060826 found
=> Checksum SHA1 OK for nvu-1.0-sources.tar.bz2
=> Checksum RMD160 OK for nvu-1.0-sources.tar.bz2
work.bacc -> /tmp/roland/pkgsrc/www/nvu/work.bacc
===> Installing dependencies for nvu-1.0
===> Overriding tools for nvu-1.0
===> Extracting for nvu-1.0
url2pkg> Adjusting the Makefile.

Remember to correct CATEGORIES, HOMEPAGE, COMMENT, and DESCR when you're done!

Good luck! (See pkgsrc/doc/pkgsrc.txt for some more help :-)
```

10.2.1.2. Fixing all kinds of problems to make the package work

Now that the package has been extracted, let's see what's inside it. The package has a `README.txt`, but that only says something about mozilla, so it's probably useless for seeing what dependencies this package has. But since there is a GNU configure script in the package, let's hope that it will complain about everything it needs.

```
$ bmake
=> Required installed package digest>=20010302: digest-20060826 found
=> Checksum SHA1 OK for nvu-1.0-sources.tar.bz2
=> Checksum RMD160 OK for nvu-1.0-sources.tar.bz2
===> Patching for nvu-1.0
===> Creating toolchain wrappers for nvu-1.0
===> Configuring for nvu-1.0
```

```
[...]
configure: error: Perl 5.004 or higher is required.
[...]
WARNING: Please add USE_TOOLS+=perl to the package Makefile.
[...]
```

That worked quite well. So I opened the package Makefile in my editor, and since it already has a `USE_TOOLS` line, I just appended "perl" to it. Since the dependencies of the package have changed now, and since a perl wrapper is automatically installed in the "tools" phase, I need to build the package from scratch.

```
$ bmake clean
===> Cleaning for nvu-1.0
$ bmake
[...]
*** /tmp/roland/pkgsrc/www/nvu/work.bacc/.tools/bin/make is not \
GNU Make.  You will not be able to build Mozilla without GNU Make.
[...]
```

So I added "gmake" to the `USE_TOOLS` line and tried again (from scratch).

```
[...]
checking for GTK - version >= 1.2.0... no
*** Could not run GTK test program, checking why...
[...]
```

Now to the other dependencies. The first question is: Where is the GTK package hidden in pkgsrc?

```
$ echo ../../*/gtk*
[many packages ...]
$ echo ../../*/gtk
../../x11/gtk
$ echo ../../*/gtk2
../../x11/gtk2
$ echo ../../*/gtk2/bui*
../../x11/gtk2/buildlink3.mk
```

The first try was definitely too broad. The second one had exactly one result, which is very good. But there is one pitfall with GNOME packages. Before GNOME 2 had been released, there were already many GNOME 1 packages in pkgsrc. To be able to continue to use these packages, the GNOME 2 packages were imported as separate packages, and their names usually have a "2" appended. So I checked whether this was the case here, and indeed it was.

Since the GTK2 package has a `buildlink3.mk` file, adding the dependency is very easy. I just inserted an `.include` line before the last line of the package `Makefile`, so that it now looks like this:

```
[...]
.include "../../x11/gtk2/buildlink3.mk"
.include "../../mk/bsd.pkg.mk
```

After another **bmake clean && bmake**, the answer was:

```
[...]
```

```
checking for gtk-config... /home/roland/pkg/bin/gtk-config
checking for GTK - version >= 1.2.0... no
*** Could not run GTK test program, checking why...
*** The test program failed to compile or link. See the file config.log for the
*** exact error that occured. This usually means GTK was incorrectly installed
*** or that you have moved GTK since it was installed. In the latter case, you
*** may want to edit the gtk-config script: /home/roland/pkg/bin/gtk-config
configure: error: Test for GTK failed.
[...]
```

In this particular case, the assumption that "every package prefers GNOME 2" had been wrong. The first of the lines above told me that this package really wanted to have the GNOME 1 version of GTK. If the package had looked for GTK2, it would have looked for **pkg-config** instead of **gtk-config**. So I changed the `x11/gtk2` to `x11/gtk` in the package `Makefile`, and tried again.

```
[...]
cc -o xpidl.o -c -DOSTYPE=\"NetBSD3\" -DOSARCH=\"NetBSD\"  [...]
In file included from xpidl.c:42:
xpidl.h:53:24: libIDL/IDL.h: No such file or directory
In file included from xpidl.c:42:
xpidl.h:132: error: parse error before "IDL_ns"
[...]
```

The package still does not find all of its dependencies. Now the question is: Which package provides the `libIDL/IDL.h` header file?

```
$ echo ../../*/*idl*
../../devel/py-idle ../../wip/idled ../../x11/acidlaunch
$ echo ../../*/*IDL*
../../net/libIDL
```

Let's take the one from the second try. So I included the `../../net/libIDL/buildlink3.mk` file and tried again. But the error didn't change. After digging through some of the code, I concluded that the build process of the package was broken and couldn't have ever worked, but since the Mozilla source tree is quite large, I didn't want to fix it. So I added the following to the package `Makefile` and tried again:

```
CPPFLAGS+=              -I${BUILDLINK_PREFIX.libIDL}/include/libIDL-2.0
BUILDLINK_TRANSFORM+=   -l:IDL:IDL-2
```

The latter line is needed because the package expects the library `libIDL.so`, but only `libIDL-2.so` is available. So I told the compiler wrapper to rewrite that on the fly.

The next problem was related to a recent change of the FreeType interface. I looked up in `www/seamonkey` which patch files were relevant for this issue and copied them to the `patches` directory. Then I retried, fixed the patches so that they applied cleanly and retried again. This time, everything worked.

10.2.1.3. Installing the package

```
$ bmake CHECK_FILES=no install
[...]
$ bmake print-PLIST >PLIST
```

```
$ bmake deinstall
$ bmake install
```

Chapter 11.
Package components - files, directories and contents

Whenever you're preparing a package, there are a number of files involved which are described in the following sections.

11.1. `Makefile`

Building, installation and creation of a binary package are all controlled by the package's `Makefile`. The `Makefile` describes various things about a package, for example from where to get it, how to configure, build, and install it.

A package `Makefile` contains several sections that describe the package.

In the first section there are the following variables, which should appear exactly in the order given here. The order and grouping of the variables is mostly historical and has no further meaning.

- `DISTNAME` is the basename of the distribution file to be downloaded from the package's website.
- `PKGNAME` is the name of the package, as used by pkgsrc. You need to provide it if `DISTNAME` (which is the default) is not a good name for the package in pkgsrc or `DISTNAME` is not provided (no distribution file is required). Usually it is the pkgsrc directory name together with the version number. It must match the regular expression `^[A-Za-z0-9][A-Za-z0-9-_.+]*$`, that is, it starts with a letter or digit, and contains only letters, digits, dashes, underscores, dots and plus signs.
- `CATEGORIES` is a list of categories which the package fits in. You can choose any of the top-level directories of pkgsrc for it.

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

  ```
  archivers     cross         geography     meta-pkgs     security
  audio         databases     graphics      misc          shells
  benchmarks    devel         ham           multimedia    sysutils
  biology       editors       inputmethod   net           textproc
  cad           emulators     lang          news          time
  chat          finance       mail          parallel      wm
  comms         fonts         math          pkgtools      www
  converters    games         mbone         print         x11
  ```

- `MASTER_SITES`, `DYNAMIC_MASTER_SITES`, `DIST_SUBDIR`, `EXTRACT_SUFX` and `DISTFILES` are discussed in detail in Section 17.5.

The second section contains information about separately downloaded patches, if any.

- `PATCHFILES`: Name(s) of additional files that contain distribution patches. There is no default. pkgsrc will look for them at `PATCH_SITES`. They will automatically be uncompressed before patching if the names end with `.gz` or `.Z`.
- `PATCH_SITES`: Primary location(s) for distribution patch files (see `PATCHFILES` above) if not found locally.
- `PATCH_DIST_STRIP`: an argument to patch(1) that sets the pathname strip count to help find the correct files to patch. It defaults to **-p0**.

The third section contains the following variables.

- `MAINTAINER` is the email address of the person who feels responsible for this package, and who is most likely to look at problems or questions regarding this package which have been reported with send-pr(1). Other developers may contact the `MAINTAINER` before making changes to the package, but are not required to do so. When packaging a new program, set `MAINTAINER` to yourself. If you really can't maintain the package for future updates, set it to <`pkgsrc-users@NetBSD.org`>.
- `OWNER` should be used instead of `MAINTAINER` when you do not want other developers to update or change the package without contacting you first. A package Makefile should contain one of `MAINTAINER` or `OWNER`, but not both.
- `HOMEPAGE` is a URL where users can find more information about the package.
- `COMMENT` is a one-line description of the package (should not include the package name).

Other variables that affect the build:

- `WRKSRC`: The directory where the interesting distribution files of the package are found. The default is `${WRKDIR}/${DISTNAME}`, which works for most packages.

  If a package doesn't create a subdirectory for itself (most GNU software does, for instance), but extracts itself in the current directory, you should set `WRKSRC=${WRKDIR}`.

  If a package doesn't create a subdirectory with the name of `DISTNAME` but some different name, set `WRKSRC` to point to the proper name in `${WRKDIR}`, for example `WRKSRC=${WRKDIR}/${DISTNAME}/unix`. See `lang/tcl` and `x11/tk` for other examples.

  The name of the working directory created by pkgsrc is taken from the `WRKDIR_BASENAME` variable. By default, its value is `work`. If you want to use the same pkgsrc tree for building different kinds of binary packages, you can change the variable according to your needs. Two other variables handle common cases of setting `WRKDIR_BASENAME` individually. If `OBJHOSTNAME` is defined in `mk.conf`, the first component of the host's name is attached to the directory name. If `OBJMACHINE` is defined, the platform name is attached, which might look like `work.i386` or `work.sparc`.

Please pay attention to the following gotchas:

- Add `MANCOMPRESSED` if man pages are installed in compressed form by the package. For packages using BSD-style makefiles which honor MANZ, there is `MANCOMPRESSED_IF_MANZ`.
- Replace `/usr/local` with "${PREFIX}" in all files (see patches, below).

- If the package installs any info files, see Section 19.6.7.

11.2. distinfo

The `distinfo` file contains the message digest, or checksum, of each distfile needed for the package. This ensures that the distfiles retrieved from the Internet have not been corrupted during transfer or altered by a malign force to introduce a security hole. Due to recent rumor about weaknesses of digest algorithms, all distfiles are protected using both SHA1 and RMD160 message digests, as well as the file size.

The `distinfo` file also contains the checksums for all the patches found in the `patches` directory (see Section 11.3).

To regenerate the `distinfo` file, use the **make makedistinfo** or **make mdi** command.

Some packages have different sets of distfiles depending on the platform, for example `lang/openjdk7`. These are kept in the same `distinfo` file and care should be taken when upgrading such a package to ensure distfile information is not lost.

11.3. patches/*

Many packages still don't work out-of-the box on the various platforms that are supported by pkgsrc. Therefore, a number of custom patch files are needed to make the package work. These patch files are found in the `patches/` directory.

In the *patch* phase, these patches are applied to the files in `WRKSRC` directory after extracting them, in alphabetic order (http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html#tag_02_13_03).

11.3.1. Structure of a single patch file

The `patch-*` files should be in **diff -bu** format, and apply without a fuzz to avoid problems. (To force patches to apply with fuzz you can set `PATCH_FUZZ_FACTOR=-F2`). Furthermore, each patch should contain only changes for a single file, and no file should be patched by more than one patch file. This helps to keep future modifications simple.

Each patch file is structured as follows: In the first line, there is the RCS Id of the patch itself. The second line should be empty for aesthetic reasons. After that, there should be a comment for each change that the patch does. There are a number of standard cases:

- Patches for commonly known vulnerabilities should mention the vulnerability ID (CAN, CVE).
- Patches that change source code should mention the platform and other environment (for example, the compiler) that the patch is needed for.

In all, the patch should be commented so that any developer who knows the code of the application can make some use of the patch. Special care should be taken for the upstream developers, since we generally want that they accept our patches, so we have less work in the future.

11.3.2. Creating patch files

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. Use the **pkgdiff** command from the `pkgtools/pkgdiff` package to avoid these problems.

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** or, easier, by using **pkgvi** again from the same package. If you upgrade a package this way, you can easily compare the new set of patches with the previously existing one with **patchdiff**. The files in `patches` are replaced by new files, so carefully check if you want to take all the changes.

When you have finished a package, remember to generate the checksums for the patch files by using the **make makepatchsum** command, see Section 11.2.

When adding a patch that corrects a problem in the distfile (rather than e.g. enforcing pkgsrc's view of where man pages should go), send the patch as a bug report to the maintainer. This benefits non-pkgsrc users of the package, and usually makes it possible to remove the patch in future version.

The file names of the patch files are usually of the form `patch-`*`path_to_file__with__underscores.c`*. Many packages still use the previous convention `patch-`*`[a-z][a-z]`*, but new patches should be of the form containing the filename. **mkpatches** included in `pkgtools/pkgdiff` takes care of the name automatically.

11.3.3. Sources where the patch files come from

If you want to share patches between multiple packages in pkgsrc, e.g. because they use the same distfiles, set `PATCHDIR` to the path where the patch files can be found, e.g.:

```
PATCHDIR= ${.CURDIR}/../xemacs/patches
```

Patch files that are distributed by the author or other maintainers can be listed in `PATCHFILES`.

If it is desired to store any patches that should not be committed into pkgsrc, they can be kept outside the pkgsrc tree in the `$LOCALPATCHES` directory. The directory tree there is expected to have the same "category/package" structure as pkgsrc, and patches are expected to be stored inside these dirs (also known as `$LOCALPATCHES/$PKGPATH`). For example, if you want to keep a private patch for `pkgsrc/graphics/png`, keep it in `$LOCALPATCHES/graphics/png/mypatch`. All files in the named directory are expected to be patch files, and *they are applied after pkgsrc patches are applied*.

11.3.4. Patching guidelines

When fixing a portability issue in the code do not use preprocessor magic to check for the current operating system nor platform. Doing so hurts portability to other platforms because the OS-specific details are not abstracted appropriately.

The general rule to follow is: instead of checking for the operating system the application is being built on, check for the specific *features* you need. For example, instead of assuming that kqueue is available under NetBSD and using the `__NetBSD__` macro to conditionalize kqueue support, add a check that detects kqueue itself — yes, this generally involves patching the **configure** script. There is absolutely

nothing that prevents some OSes from adopting interfaces from other OSes (e.g. Linux implementing kqueue), something that the above checks cannot take into account.

Of course, checking for features generally involves more work on the developer's side, but the resulting changes are cleaner and there are chances they will work on many other platforms. Not to mention that there are higher chances of being later integrated into the mainstream sources. Remember: *It doesn't work unless it is right!*

Some typical examples:

Table 11-1. Patching examples

Where	Incorrect	Correct

configure script	`case ${target_os} in`
	`netbsd*) have_kvm=yes ;;`
	`*) have_kvm=no ;;`
	`esac`
		`AC_CHECK_LIB(kvm, kvm_open, have_kvm=ye`

C source file	`#if defined(__NetBSD__)`
	`# include <sys/event.h>`
	`#endif`
		`#if defined(HAVE_SYS_EVENT_H)`
		`# include <sys/event.h>`
		`#endif`

C source file	`int monitor_file(...) {`
	`#if defined(__NetBSD__)`
	`int fd = kqueue();`
	`... #else ...`
	`#endif }`
		`int monitor_file(...) {`
		`#if defined(HAVE_KQUEUE)`
		`int fd = kqueue();`
		`... #else ... #endif }`

For more information, please read the *Making packager-friendly software* article (part 1 (http://www.onlamp.com/pub/a/onlamp/2005/03/31/packaging.html), part 2 (http://www.oreillynet.com/pub/a/onlamp/2005/04/28/packaging2.html)). It summarizes multiple details on how to make software easier to package; all the suggestions in it were collected from our experience in pkgsrc work, so they are possibly helpful when creating patches too.

11.3.5. Feedback to the author

Always, always, **always** feed back any *portability fixes* or improvements you do to a package to the mainstream developers. This is the only way to get their attention on portability issues and to ensure that future versions can be built out-of-the box on NetBSD. Furthermore, any user that gets newer distfiles will get the fixes straight from the packaged code.

This generally involves cleaning up the patches (because sometimes the patches that are added to pkgsrc are quick hacks), filing bug reports in the appropriate trackers for the projects and working with the mainstream authors to accept your changes. It is *extremely important* that you do it so that the packages in pkgsrc are kept simple and thus further changes can be done without much hassle.

When you have done this, please add a URL to the upstream bug report to the patch comment.

Support the idea of free software!

11.4. Other mandatory files

`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.

`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. See Chapter 13 for more information.

11.5. Optional files

11.5.1. Files affecting the binary package

`INSTALL`

This shell script is invoked twice by pkg_add(1). 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. See also Section 15.1. Please note that you can modify variables in it easily by using `FILES_SUBST` in the package's `Makefile`:

```
FILES_SUBST+=  SOMEVAR="somevalue"
```

replaces "@SOMEVAR@" with "somevalue" in the `INSTALL`. By default, substitution is performed for `PREFIX`, `LOCALBASE`, `X11BASE`, `VARBASE`, and a few others, type **make help topic=FILES_SUBST** for a complete list.

`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. The same methods to replace variables can be used as for the `INSTALL` file.

`MESSAGE`

This file is displayed after installation of the package. Useful for things like legal notices on almost-free software and hints for updating config files after installing modules for apache, PHP etc. Please note that you can modify variables in it easily by using `MESSAGE_SUBST` in the package's `Makefile`:

```
MESSAGE_SUBST+=  SOMEVAR="somevalue"
```

replaces "${SOMEVAR}" with "somevalue" in `MESSAGE`. By default, substitution is performed for `PKGNAME`, `PKGBASE`, `PREFIX`, `LOCALBASE`, `X11PREFIX`, `X11BASE`, `PKG_SYSCONFDIR`, `ROOT_GROUP`, and `ROOT_USER`.

You can display a different or additional files by setting the `MESSAGE_SRC` variable. Its default is `MESSAGE`, if the file exists.

`ALTERNATIVES`

This file is used by the alternatives framework. It creates, configures, and destroys generic wrappers used to run programs with similar interfaces. See pkg_alternatives(8) from pkgtools/pkg_alternatives for more information.

Each line of the file contains two filenames, first the wrapper and then the alternative provided by the package. Both paths are relative to `PREFIX`.

11.5.2. Files affecting the build process

`Makefile.common`

This file contains arbitrary things that could also go into a `Makefile`, but its purpose is to be used by more than one package. This file should only be used when the packages that will use the file are known in advance. For other purposes it is often better to write a `*.mk` file and give it a good name that describes what it does.

`buildlink3.mk`

This file contains the dependency information for the buildlink3 framework (see Chapter 14).

`hacks.mk`

This file contains workarounds for compiler bugs and similar things. It is included automatically by the pkgsrc infrastructure, so you don't need an extra `.include` line for it.

`options.mk`

This file contains the code for the package-specific options (see Chapter 16) that can be selected by the user. If a package has only one or two options, it is equally acceptable to put the code directly into the `Makefile`.

11.5.3. Files affecting nothing at all

`README*`

These files do not take place in the creation of a package and thus are purely informative to the package developer.

`TODO`

This file contains things that need to be done to make the package even better.

11.6. work*

When you type **make**, the distribution files are unpacked into the directory denoted by `WRKDIR`. It can be removed by running **make clean**. Besides the sources, this directory is also used to keep various timestamp files. The directory gets *removed completely* on clean. The default is `${.CURDIR}/work` or `${.CURDIR}/work.${MACHINE_ARCH}` if `OBJMACHINE` is set.

11.7. files/*

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.

If you want to share files in this way with other packages, set the `FILESDIR` variable to point to the other package's `files` directory, e.g.:

```
FILESDIR=${.CURDIR}/../xemacs/files
```

Chapter 12.
Programming in `Makefiles`

Pkgsrc consists of many `Makefile` fragments, each of which forms a well-defined part of the pkgsrc system. Using the make(1) system as a programming language for a big system like pkgsrc requires some discipline to keep the code correct and understandable.

The basic ingredients for `Makefile` programming are variables (which are actually macros) and shell commands. Among these shell commands may even be more complex ones like awk(1) programs. To make sure that every shell command runs as intended it is necessary to quote all variables correctly when they are used.

This chapter describes some patterns, that appear quite often in `Makefiles`, including the pitfalls that come along with them.

12.1. Caveats

- When you are creating a file as a target of a rule, always write the data to a temporary file first and finally rename that file. Otherwise there might occur an error in the middle of generating the file, and when the user runs make(1) for the second time, the file exists and will not be regenerated properly. Example:

  ```
  wrong:
          @echo "line 1" > ${.TARGET}
          @echo "line 2" >> ${.TARGET}
          @false

  correct:
          @echo "line 1" > ${.TARGET}.tmp
          @echo "line 2" >> ${.TARGET}.tmp
          @false
          @mv ${.TARGET}.tmp ${.TARGET}
  ```

  When you run **make wrong** twice, the file `wrong` will exist, although there was an error message in the first run. On the other hand, running **make correct** gives an error message twice, as expected.

  You might remember that make(1) sometimes removes `${.TARGET}` in case of error, but this only happens when it is interrupted, for example by pressing `^C`. This does *not* happen when one of the commands fails (like false(1) above).

12.2. `Makefile` variables

`Makefile` variables contain strings that can be processed using the five operators “=”, “+=”, “?=”, “:=”, and “!=”, which are described in the make(1) man page.

When a variable's value is parsed from a `Makefile`, the hash character "#" and the backslash character "\" are handled specially. If a backslash is followed by a newline, any whitespace immediately in front of the backslash, the backslash, the newline, and any whitespace immediately behind the newline are replaced with a single space. A backslash character and an immediately following hash character are replaced with a single hash character. Otherwise, the backslash is passed as is. In a variable assignment, any hash character that is not preceded by a backslash starts a comment that continues upto the end of the logical line.

Note: Because of this parsing algorithm the only way to create a variable consisting of a single backslash is using the "!=" operator, for example: `BACKSLASH!=echo "\\"`.

So far for defining variables. The other thing you can do with variables is evaluating them. A variable is evaluated when it is part of the right side of the ":=" or the "!=" operator, or directly before executing a shell command which the variable is part of. In all other cases, make(1) performs lazy evaluation, that is, variables are not evaluated until there's no other way. The "modifiers" mentioned in the man page also evaluate the variable.

Some of the modifiers split the string into words and then operate on the words, others operate on the string as a whole. When a string is split into words, it is split as you would expect it from sh(1).

No rule without exception—the **.for** loop does not follow the shell quoting rules but splits at sequences of whitespace.

There are several types of variables that should be handled differently. Strings and two types of lists.

- *Strings* can contain arbitrary characters. Nevertheless, you should restrict yourself to only using printable characters. Examples are `PREFIX` and `COMMENT`.
- *Internal lists* are lists that are never exported to any shell command. Their elements are separated by whitespace. Therefore, the elements themselves cannot have embedded whitespace. Any other characters are allowed. Internal lists can be used in **.for** loops. Examples are `DEPENDS` and `BUILD_DEPENDS`.
- *External lists* are lists that may be exported to a shell command. Their elements can contain any characters, including whitespace. That's why they cannot be used in **.for** loops. Examples are `DISTFILES` and `MASTER_SITES`.

12.2.1. Naming conventions

- All variable names starting with an underscore are reserved for use by the pkgsrc infrastructure. They shall not be used by package `Makefiles`.
- In **.for** loops you should use lowercase variable names for the iteration variables.
- All list variables should have a "plural" name, e.g. `PKG_OPTIONS` or `DISTFILES`.

12.3. Code snippets

This section presents you with some code snippets you should use in your own code. If you don't find anything appropriate here, you should test your code and add it here.

12.3.1. Adding things to a list

```
STRING=                 foo * bar `date`
INT_LIST=               # empty
ANOTHER_INT_LIST=       apache-[0-9]*:../../www/apache
EXT_LIST=               # empty
ANOTHER_EXT_LIST=       a=b c=d

INT_LIST+=              ${STRING}               # 1
INT_LIST+=              ${ANOTHER_INT_LIST}     # 2
EXT_LIST+=              ${STRING:Q}             # 3
EXT_LIST+=              ${ANOTHER_EXT_LIST}     # 4
```

When you add a string to an external list (example 3), it must be quoted. In all other cases, you must not add a quoting level. You must not merge internal and external lists, unless you are sure that all entries are correctly interpreted in both lists.

12.3.2. Converting an internal list into an external list

```
EXT_LIST=       # empty
.for i in ${INT_LIST}
EXT_LIST+=      ${i:Q}""
.endfor
```

This code converts the internal list `INT_LIST` into the external list `EXT_LIST`. As the elements of an internal list are unquoted they must be quoted here. The reason for appending `""` is explained below.

12.3.3. Passing variables to a shell command

Sometimes you may want to print an arbitrary string. There are many ways to get it wrong and only few that can handle every nastiness.

```
STRING=         foo bar <    > * `date` $$HOME ' "
EXT_LIST=       string=${STRING:Q} x=second\ item

all:
        echo ${STRING}                  # 1
        echo "${STRING}"                # 2
        echo "${STRING:Q}"              # 3
        echo ${STRING:Q}                # 4
        echo x${STRING:Q} | sed 1s,.,,  # 5
        printf "%s\\n" ${STRING:Q}""    # 6
        env ${EXT_LIST} /bin/sh -c 'echo "$$string"; echo "$$x"'
```

Example 1 leads to a syntax error in the shell, as the characters are just copied.

Example 2 leads to a syntax error too, and if you leave out the last " character from `${STRING}`, date(1) will be executed. The `$HOME` shell variable would be evaluated, too.

Example 3 outputs each space character preceded by a backslash (or not), depending on the implementation of the echo(1) command.

Example 4 handles correctly every string that does not start with a dash. In that case, the result depends on the implementation of the echo(1) command. As long as you can guarantee that your input does not start with a dash, this form is appropriate.

Example 5 handles even the case of a leading dash correctly.

Example 6 also works with every string and is the light-weight solution, since it does not involve a pipe, which has its own problems.

The `EXT_LIST` does not need to be quoted because the quoting has already been done when adding elements to the list.

As internal lists shall not be passed to the shell, there is no example for it.

12.3.4. Quoting guideline

There are many possible sources of wrongly quoted variables. This section lists some of the commonly known ones.

- Whenever you use the value of a list, think about what happens to leading or trailing whitespace. If the list is a well-formed shell expression, you can apply the `:M*` modifier to strip leading and trailing whitespace from each word. The `:M` operator first splits its argument according to the rules of the shell, and then creates a new list consisting of all words that match the shell glob expression `*`, that is: all. One class of situations where this is needed is when adding a variable like `CPPFLAGS` to `CONFIGURE_ARGS`. If the configure script invokes other configure scripts, it strips the leading and trailing whitespace from the variable and then passes it to the other configure scripts. But these configure scripts expect the (child) `CPPFLAGS` variable to be the same as the parent `CPPFLAGS`. That's why we better pass the `CPPFLAGS` value properly trimmed. And here is how we do it:

```
CPPFLAGS=               # empty
CPPFLAGS+=              -Wundef -DPREFIX=\"${PREFIX:Q}\"
CPPFLAGS+=              ${MY_CPPFLAGS}

CONFIGURE_ARGS+=        CPPFLAGS=${CPPFLAGS:M*:Q}

all:
        echo x${CPPFLAGS:Q}x            # leading and trailing whitespace
        echo x${CONFIGURE_ARGS}x        # properly trimmed
```

- The example above contains one bug: The `${PREFIX}` is a properly quoted shell expression, but there is the C compiler after it, which also expects a properly quoted string (this time in C syntax). The version above is therefore only correct if `${PREFIX}` does not have embedded backslashes or double quotes. If you want to allow these, you have to add another layer of quoting to each variable that is used as a C string literal. You cannot use the `:Q` operator for it, as this operator only works for the shell.
- Whenever a variable can be empty, the `:Q` operator can have surprising results. Here are two completely different cases which can be solved with the same trick.

```
EMPTY=                  # empty
empty_test:
        for i in a ${EMPTY:Q} c; do \
            echo "$$i"; \
```

```
        done

for_test:
.for i in a:\ a:\test.txt
        echo ${i:Q}
        echo "foo"
.endfor
```

The first example will only print two of the three lines we might have expected. This is because `${EMPTY:Q}` expands to the empty string, which the shell cannot see. The workaround is to write `${EMPTY:Q}""`. This pattern can be often found as `${TEST} -z ${VAR:Q}` or as `${TEST} -f ${FNAME:Q}` (both of these are wrong).

The second example will only print three lines instead of four. The first line looks like `a:\ echo foo`. This is because the backslash of the value `a:\` is interpreted as a line-continuation by make(1), which makes the second line the arguments of the echo(1) command from the first line. To avoid this, write `${i:Q}""`.

12.3.5. Workaround for a bug in BSD Make

The pkgsrc bmake program does not handle the following assignment correctly. In case `_othervar_` contains a "-" character, one of the closing braces is included in `${VAR}` after this code executes.

```
VAR:=   ${VAR:N${_othervar_:C/-//}}
```

For a more complex code snippet and a workaround, see the package `regress/make-quoting`, testcase `bug1`.

Chapter 13.
PLIST issues

The PLIST file contains a package's "packing list", i.e. a list of files that belong to the package (relative to the ${PREFIX} directory it's been installed in) plus some additional statements - see the pkg_create(1) man page for a full list. This chapter addresses some issues that need attention when dealing with the PLIST file (or files, see below!).

13.1. RCS ID

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

```
@comment $NetBSD: pkgsrc.ps,v 1.90 2015/03/07 21:32:40 tnn Exp $
```

13.2. 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. See Section 17.17 for more information on this target.

13.3. Tweaking output of make print-PLIST

If you have used any of the *-dirs packages, as explained in Section 13.9, you may have noticed that **make print-PLIST** outputs a set of @comments instead of real @dirrm lines. You can also do this for specific directories and files, so that the results of that command are very close to reality. This helps *a lot* during the update of packages.

The PRINT_PLIST_AWK variable takes a set of AWK patterns and actions that are used to filter the output of print-PLIST. You can *append* any chunk of AWK scripting you like to it, but be careful with quoting.

For example, to get all files inside the libdata/foo directory removed from the resulting PLIST:

```
PRINT_PLIST_AWK+=       /^libdata\/foo/ { next; }
```

And to get all the @dirrm lines referring to a specific (shared) directory converted to @comments:

```
PRINT_PLIST_AWK+=       /^@dirrm share\/specific/ { print "@comment " $$0; next; }
```

13.4. Variable substitution in PLIST

A number of variables are substituted automatically in PLISTs when a package is installed on a system. This includes the following variables:

`${MACHINE_ARCH}`, `${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 files. To handle this case, PLIST will be preprocessed before actually used, and the symbol "`${MACHINE_ARCH}`" will be replaced by what **uname -p** gives. The same is done if the string `${MACHINE_GNU_ARCH}` is embedded in PLIST somewhere - use this on packages that have GNU autoconf-created configure scripts.

> **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.

`${OPSYS}`, `${LOWER_OPSYS}`, `${OS_VERSION}`

Some packages want to embed the OS name and version into some paths. To do this, use these variables in the `PLIST`:

- `${OPSYS}` - output of "**uname -s**"
- `${LOWER_OPSYS}` - lowercase common name (eg. "solaris")
- `${OS_VERSION}` - "**uname -r**"

For a list of values which are replaced by default, the output of **make help topic=PLIST_SUBST** as well as searching the `pkgsrc/mk` directory with **grep** for `PLIST_SUBST` should help.

If you want to change other variables not listed above, you can add variables and their expansions to this variable in the following way, similar to `MESSAGE_SUBST` (see Section 11.5):

```
PLIST_SUBST+=   SOMEVAR="somevalue"
```

This replaces all occurrences of "${SOMEVAR}" in the `PLIST` with "somevalue".

The `PLIST_VARS` variable can be used to simplify the common case of conditionally including some `PLIST` entries. It can be done by adding `PLIST_VARS+=foo` and setting the corresponding `PLIST.foo` variable to `yes` if the entry should be included. This will substitute "`${PLIST.foo}`" in the `PLIST` with either "`""`" or "`"@comment "`". For example, in `Makefile`:

```
PLIST_VARS+= foo
.if condition
PLIST.foo= yes
.else
```

And then in `PLIST`:

```
@comment $NetBSD: pkgsrc.ps,v 1.90 2015/03/07 21:32:40 tnn Exp $
bin/bar
man/man1/bar.1
${PLIST.foo}bin/foo
${PLIST.foo}man/man1/foo.1
${PLIST.foo}share/bar/foo.data
${PLIST.foo}@dirrm share/bar
```

13.5. Man page compression

Man pages should be installed in compressed form if `MANZ` is set (in `bsd.own.mk`), and uncompressed otherwise. To handle this in the `PLIST` file, the suffix ".gz" is appended/removed automatically for man pages 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 `PLIST` itself.

13.6. Changing PLIST source with `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 the order of things is important. The default for `PLIST_SRC` is `${PKGDIR}/PLIST`.

13.7. Platform-specific and differing PLISTs

Some packages decide to install a different set of files based on the operating system being used. These differences can be automatically handled by using the following files:

- `PLIST.common`
- `PLIST.${OPSYS}`
- `PLIST.${MACHINE_ARCH}`
- `PLIST.${OPSYS}-${MACHINE_ARCH}`
- `PLIST.common_end`

13.8. Build-specific PLISTs

Some packages decide to generate hard-to-guess file names during installation that are hard to wire down.

In such cases, you can set the `GENERATE_PLIST` variable to shell code terminated (with a semicolon) that will output PLIST entries which will be appended to the PLIST

You can find one example in editors/xemacs:

```
GENERATE_PLIST+=        ${ECHO} bin/${DISTNAME}-`${WRKSRC}/src/xemacs -sd`.dmp ;
```

which will append something like `bin/xemacs-21.4.23-54e8ea71.dmp` to the `PLIST`.

13.9. Sharing directories between packages

A "shared directory" is a directory where multiple (and unrelated) packages install files. These directories were problematic because you had to add special tricks in the PLIST to conditionally remove them, or have some centralized package handle them.

In pkgsrc, it is now easy: Each package should create directories and install files as needed; **pkg_delete** will remove any directories left empty after uninstalling a package.

If a package needs an empty directory to work, create the directory during installation as usual, and also add an entry to the PLIST:

```
@pkgdir path/to/empty/directory
```

or take a look at `MAKE_DIRS` and `OWN_DIRS`.

Chapter 14.
Buildlink methodology

Buildlink is a framework in pkgsrc that controls what headers and libraries are seen by a package's configure and build processes. This is implemented in a two step process:

1. Symlink headers and libraries for dependencies into `BUILDLINK_DIR`, which by default is a subdirectory of `WRKDIR`.
2. Create wrapper scripts that are used in place of the normal compiler tools that translate `-I${LOCALBASE}/include` and `-L${LOCALBASE}/lib` into references to `BUILDLINK_DIR`. The wrapper scripts also make native compiler on some operating systems look like GCC, so that packages that expect GCC won't require modifications to build with those native compilers.

This normalizes the environment in which a package is built so that the package may be built consistently despite what other software may be installed. Please note that the normal system header and library paths, e.g. `/usr/include`, `/usr/lib`, etc., are always searched -- buildlink3 is designed to insulate the package build from non-system-supplied software.

14.1. Converting packages to use buildlink3

The process of converting packages to use the buildlink3 framework ("bl3ifying") is fairly straightforward. The things to keep in mind are:

1. Ensure that the build always calls the wrapper scripts instead of the actual toolchain. Some packages are tricky, and the only way to know for sure is the check `${WRKDIR}/.work.log` to see if the wrappers are being invoked.
2. Don't override `PREFIX` from within the package Makefile, e.g. Java VMs, standalone shells, etc., because the code to symlink files into `${BUILDLINK_DIR}` looks for files relative to "pkg_info -qp *`pkgname`*".
3. Remember that *only* the `buildlink3.mk` files that you list in a package's Makefile are added as dependencies for that package.

If a dependency on a particular package is required for its libraries and headers, then we replace:

```
DEPENDS+=       foo>=1.1.0:../../category/foo
```

with

```
.include "../../category/foo/buildlink3.mk"
```

The buildlink3.mk files usually define the required dependencies. If you need a newer version of the dependency when using buildlink3.mk files, then you can define it in your Makefile; for example:

```
BUILDLINK_API_DEPENDS.foo+=   foo>=1.1.0
.include "../../category/foo/buildlink3.mk"
```

There are several `buildlink3.mk` files in `pkgsrc/mk` that handle special package issues:

- `bdb.buildlink3.mk` chooses either the native or a pkgsrc Berkeley DB implementation based on the values of `BDB_ACCEPTED` and `BDB_DEFAULT`.
- `curses.buildlink3.mk`: If the system comes with neither Curses nor NCurses, this will take care to install the `devel/ncurses` package.
- `krb5.buildlink3.mk` uses the value of `KRB5_ACCEPTED` to choose between adding a dependency on Heimdal or MIT-krb5 for packages that require a Kerberos 5 implementation.
- `motif.buildlink3.mk` checks for a system-provided Motif installation or adds a dependency on `x11/lesstif`, `x11/motif` or `x11/openmotif`. The user can set `MOTIF_TYPE` to "dt", "lesstif", "motif" or "openmotif" to choose which Motif version will be used.
- `readline.buildlink3.mk` checks for a system-provided GNU readline or editline (libedit) installation, or adds a dependency on `devel/readline`, `devel/editline`. The user can set `READLINE_DEFAULT` to choose readline implementation. If your package really needs GNU readline library, its Makefile should include `devel/readline/buildlink3.mk` instead of `readline.buildlink3.mk`.
- `oss.buildlink3.mk` defines several variables that may be used by packages that use the Open Sound System (OSS) API.
- `pgsql.buildlink3.mk` will accept any of the Postgres versions in the variable `PGSQL_VERSIONS_ACCEPTED` and default to the version `PGSQL_VERSION_DEFAULT`. See the file for more information.
- `pthread.buildlink3.mk` uses the value of `PTHREAD_OPTS` and checks for native pthreads or adds a dependency on `devel/pth` as needed.
- `xaw.buildlink3.mk` uses the value of `XAW_TYPE` to choose a particular Athena widgets library.

The comments in those `buildlink3.mk` files provide a more complete description of how to use them properly.

14.2. Writing `buildlink3.mk` files

A package's `buildlink3.mk` file is included by Makefiles to indicate the need to compile and link against header files and libraries provided by the package. A `buildlink3.mk` file should always provide enough information to add the correct type of dependency relationship and include any other `buildlink3.mk` files that it needs to find headers and libraries that it needs in turn.

To generate an initial `buildlink3.mk` file for further editing, Rene Hexel's `pkgtools/createbuildlink` package is highly recommended. For most packages, the following command will generate a good starting point for `buildlink3.mk` files:

```
% cd pkgsrc/category/pkgdir
% createbuildlink >buildlink3.mk
```

14.2.1. Anatomy of a buildlink3.mk file

The following real-life example `buildlink3.mk` is taken from `pkgsrc/graphics/tiff`:

```
# $NetBSD: buildlink3.mk,v 1.16 2009/03/20 19:24:45 joerg Exp $

BUILDLINK_TREE+= tiff

.if !defined(TIFF_BUILDLINK3_MK)
TIFF_BUILDLINK3_MK:=

BUILDLINK_API_DEPENDS.tiff+= tiff>=3.6.1
BUILDLINK_ABI_DEPENDS.tiff+= tiff>=3.7.2nb1
BUILDLINK_PKGSRCDIR.tiff?= ../../graphics/tiff

.include "../../devel/zlib/buildlink3.mk"
.include "../../graphics/jpeg/buildlink3.mk"
.endif # TIFF_BUILDLINK3_MK

BUILDLINK_TREE+= -tiff
```

The header and footer manipulate `BUILDLINK_TREE`, which is common across all `buildlink3.mk` files and is used to track the dependency tree.

The main section is protected from multiple inclusion and controls how the dependency on *`pkg`* is added. Several important variables are set in the section:

- `BUILDLINK_API_DEPENDS.`*`pkg`* is the actual dependency recorded in the installed package; this should always be set using **+=** to ensure that we're appending to any pre-existing list of values. This variable should be set to the first version of the package that had an backwards-incompatible API change.
- `BUILDLINK_PKGSRCDIR.`*`pkg`* is the location of the *`pkg`* pkgsrc directory.
- `BUILDLINK_DEPMETHOD.`*`pkg`* (not shown above) controls whether we use `BUILD_DEPENDS` or `DEPENDS` to add the dependency on *`pkg`*. The build dependency is selected by setting `BUILDLINK_DEPMETHOD.`*`pkg`* to "build". By default, the full dependency is used.
- `BUILDLINK_INCDIRS.`*`pkg`* and `BUILDLINK_LIBDIRS.`*`pkg`* (not shown above) are lists of subdirectories of `${BUILDLINK_PREFIX.`*`pkg`*`}` to add to the header and library search paths. These default to "include" and "lib" respectively.
- `BUILDLINK_CPPFLAGS.`*`pkg`* (not shown above) is the list of preprocessor flags to add to `CPPFLAGS`, which are passed on to the configure and build phases. The "-I" option should be avoided and instead be handled using `BUILDLINK_INCDIRS.`*`pkg`* as above.

The following variables are all optionally defined within this second section (protected against multiple inclusion) and control which package files are symlinked into `${BUILDLINK_DIR}` and how their names are transformed during the symlinking:

- `BUILDLINK_FILES.`*`pkg`* (not shown above) is a shell glob pattern relative to `${BUILDLINK_PREFIX.`*`pkg`*`}` to be symlinked into `${BUILDLINK_DIR}`, e.g. `include/*.h`.

- BUILDLINK_FILES_CMD.*pkg* (not shown above) is a shell pipeline that outputs to stdout a list of files relative to ${BUILDLINK_PREFIX.*pkg*}. The resulting files are to be symlinked into ${BUILDLINK_DIR}. By default, this takes the +CONTENTS of a *pkg* and filters it through ${BUILDLINK_CONTENTS_FILTER.*pkg*}.
- BUILDLINK_CONTENTS_FILTER.*pkg* (not shown above) is a filter command that filters +CONTENTS input into a list of files relative to ${BUILDLINK_PREFIX.*pkg*} on stdout. By default, BUILDLINK_CONTENTS_FILTER.*pkg* outputs the contents of the include and lib directories in the package +CONTENTS.
- BUILDLINK_FNAME_TRANSFORM.*pkg* (not shown above) is a list of sed arguments used to transform the name of the source filename into a destination filename, e.g. **-e "s|/curses.h|/ncurses.h|g"**.

This section can additionally include any buildlink3.mk needed for *pkg*'s library dependencies. Including these buildlink3.mk files means that the headers and libraries for these dependencies are also symlinked into ${BUILDLINK_DIR} whenever the *pkg* buildlink3.mk file is included. Dependencies are only added for directly include buildlink3.mk files.

When providing a buildlink3.mk and including other buildlink3.mk files in it, please only add necessary ones, i.e., those whose libraries or header files are automatically exposed when the package is use.

In particular, if only an executable (bin/foo) is linked against a library, that library does not need to be propagated in the buildlink3.mk file.

The following steps should help you decide if a buildlink3.mk file needs to be included:

- Look at the installed header files: What headers do they include? The packages providing these files must be buildlinked.
- Run ldd on all installed libraries and look against what other libraries they link. Some of the packages providing these probably need to be buildlinked; however, it's not automatic, since e.g. GTK on some systems pulls in the X libraries, so they will show up in the ldd output, while on others (like OS X) it won't. ldd output can thus only be used as a hint.

14.2.2. Updating BUILDLINK_API_DEPENDS.*pkg* and BUILDLINK_ABI_DEPENDS.*pkg* in buildlink3.mk files

These two variables differ in that one describes source compatibility (API) and the other binary compatibility (ABI). The difference is that a change in the API breaks compilation of programs while changes in the ABI stop compiled programs from running.

Changes to the BUILDLINK_API_DEPENDS.*pkg* variable in a buildlink3.mk file happen very rarely. One possible reason is that all packages depending on this already need a newer version. In case it is bumped see the description below.

The most common example of an ABI change is that the major version of a shared library is increased. In this case, BUILDLINK_ABI_DEPENDS.*pkg* should be adjusted to require at least the new package version. Then the packages that depend on this package need their PKGREVISIONs increased and, if they have buildlink3.mk files, their BUILDLINK_ABI_DEPENDS.*pkg* adjusted, too. This is needed so

pkgsrc will require the correct package dependency and not settle for an older one when building the source.

See Section 19.1.6 for more information about dependencies on other packages, including the `BUILDLINK_ABI_DEPENDS` and `ABI_DEPENDS` definitions.

Please take careful consideration before adjusting `BUILDLINK_API_DEPENDS.pkg` or `BUILDLINK_ABI_DEPENDS.pkg` as we don't want to cause unneeded package deletions and rebuilds. In many cases, new versions of packages work just fine with older dependencies.

Also it is not needed to set `BUILDLINK_ABI_DEPENDS.pkg` when it is identical to `BUILDLINK_API_DEPENDS.pkg`.

14.3. Writing `builtin.mk` files

Some packages in pkgsrc install headers and libraries that coincide with headers and libraries present in the base system. Aside from a `buildlink3.mk` file, these packages should also include a `builtin.mk` file that includes the necessary checks to decide whether using the built-in software or the pkgsrc software is appropriate.

The only requirements of a builtin.mk file for `pkg` are:

1. It should set `USE_BUILTIN.pkg` to either "yes" or "no" after it is included.
2. It should *not* override any `USE_BUILTIN.pkg` which is already set before the `builtin.mk` file is included.
3. It should be written to allow multiple inclusion. This is *very* important and takes careful attention to `Makefile` coding.

14.3.1. Anatomy of a `builtin.mk` file

The following is the recommended template for builtin.mk files:

```
.if !defined(IS_BUILTIN.foo)
#
# IS_BUILTIN.foo is set to "yes" or "no" depending on whether "foo"
# genuinely exists in the system or not.
#
IS_BUILTIN.foo?=        no

# BUILTIN_PKG.foo should be set here if "foo" is built-in and its package
# version can be determined.
#
.  if !empty(IS_BUILTIN.foo:M[yY][eE][sS])
BUILTIN_PKG.foo?=       foo-1.0
.  endif
.endif  # IS_BUILTIN.foo

.if !defined(USE_BUILTIN.foo)
USE_BUILTIN.foo?=       ${IS_BUILTIN.foo}
```

```
.  if defined(BUILTIN_PKG.foo)
.    for _depend_ in ${BUILDLINK_API_DEPENDS.foo}
.      if !empty(USE_BUILTIN.foo:M[yY][eE][sS])
USE_BUILTIN.foo!=                                                    \
        ${PKG_ADMIN} pmatch '${_depend_}' ${BUILTIN_PKG.foo}         \
        && ${ECHO} "yes" || ${ECHO} "no"
.      endif
.    endfor
.  endif
.endif  # USE_BUILTIN.foo

CHECK_BUILTIN.foo?=     no
.if !empty(CHECK_BUILTIN.foo:M[nN][oO])
#
# Here we place code that depends on whether USE_BUILTIN.foo is set to
# "yes" or "no".
#
.endif  # CHECK_BUILTIN.foo
```

The first section sets `IS_BUILTIN.pkg` depending on if *pkg* really exists in the base system. This should not be a base system software with similar functionality to *pkg*; it should only be "yes" if the actual package is included as part of the base system. This variable is only used internally within the `builtin.mk` file.

The second section sets `BUILTIN_PKG.pkg` to the version of *pkg* in the base system if it exists (if `IS_BUILTIN.pkg` is "yes"). This variable is only used internally within the `builtin.mk` file.

The third section sets `USE_BUILTIN.pkg` and is *required* in all `builtin.mk` files. The code in this section must make the determination whether the built-in software is adequate to satisfy the dependencies listed in `BUILDLINK_API_DEPENDS.pkg`. This is typically done by comparing `BUILTIN_PKG.pkg` against each of the dependencies in `BUILDLINK_API_DEPENDS.pkg`. `USE_BUILTIN.pkg` *must* be set to the correct value by the end of the `builtin.mk` file. Note that `USE_BUILTIN.pkg` may be "yes" even if `IS_BUILTIN.pkg` is "no" because we may make the determination that the built-in version of the software is similar enough to be used as a replacement.

The last section is guarded by `CHECK_BUILTIN.pkg`, and includes code that uses the value of `USE_BUILTIN.pkg` set in the previous section. This typically includes, e.g., adding additional dependency restrictions and listing additional files to symlink into `${BUILDLINK_DIR}` (via `BUILDLINK_FILES.pkg`).

14.3.2. Global preferences for native or pkgsrc software

When building packages, it's possible to choose whether to set a global preference for using either the built-in (native) version or the pkgsrc version of software to satisfy a dependency. This is controlled by setting `PREFER_PKGSRC` and `PREFER_NATIVE`. These variables take values of either "yes", "no", or a list of packages. `PREFER_PKGSRC` tells pkgsrc to use the pkgsrc versions of software, while `PREFER_NATIVE` tells pkgsrc to use the built-in versions. Preferences are determined by the most specific instance of the package in either `PREFER_PKGSRC` or `PREFER_NATIVE`. If a package is specified in neither or in both variables, then `PREFER_PKGSRC` has precedence over `PREFER_NATIVE`. For example, to require using pkgsrc versions of software for all but the most basic bits on a NetBSD system, you can set:

```
PREFER_PKGSRC=  yes
PREFER_NATIVE=  getopt skey tcp_wrappers
```

A package *must* have a `builtin.mk` file to be listed in `PREFER_NATIVE`, otherwise it is simply ignored in that list.

Chapter 15.
The pkginstall framework

This chapter describes the framework known as `pkginstall`, whose key features are:

- Generic installation and manipulation of directories and files outside the pkgsrc-handled tree, `LOCALBASE`.
- Automatic handling of configuration files during installation, provided that packages are correctly designed.
- Generation and installation of system startup scripts.
- Registration of system users and groups.
- Registration of system shells.
- Automatic updating of fonts databases.

The following sections inspect each of the above points in detail.

You may be thinking that many of the things described here could be easily done with simple code in the package's post-installation target (`post-install`). *This is incorrect*, as the code in them is only executed when building from source. Machines using binary packages could not benefit from it at all (as the code itself could be unavailable). Therefore, the only way to achieve any of the items described above is by means of the installation scripts, which are automatically generated by pkginstall.

15.1. Files and directories outside the installation prefix

As you already know, the `PLIST` file holds a list of files and directories that belong to a package. The names used in it are relative to the installation prefix (`${PREFIX}`), which means that it cannot register files outside this directory (absolute path names are not allowed). Despite this restriction, some packages need to install files outside this location; e.g., under `${VARBASE}` or `${PKG_SYSCONFDIR}`. The only way to achieve this is to create such files during installation time by using installation scripts.

The generic installation scripts are shell scripts that can contain arbitrary code. The list of scripts to execute is taken from the `INSTALL_FILE` variable, which defaults to `INSTALL`. A similar variable exists for package removal (`DEINSTALL_FILE`, whose default is `DEINSTALL`). These scripts can run arbitrary commands, so they have the potential to create and manage files anywhere in the file system.

Using these general installation files is not recommended, but may be needed in some special cases. One reason for avoiding them is that the user has to trust the packager that there is no unwanted or simply erroneous code included in the installation script. Also, previously there were many similar scripts for the same functionality, and fixing a common error involved finding and changing all of them.

The pkginstall framework offers another, standardized way. It provides generic scripts to abstract the manipulation of such files and directories based on variables set in the package's `Makefile`. The rest of this section describes these variables.

15.1.1. Directory manipulation

The following variables can be set to request the creation of directories anywhere in the file system:

- `MAKE_DIRS` and `OWN_DIRS` contain a list of directories that should be created and should attempt to be destroyed by the installation scripts. The difference between the two is that the latter prompts the administrator to remove any directories that may be left after deinstallation (because they were not empty), while the former does not.
- `MAKE_DIRS_PERMS` and `OWN_DIRS_PERMS` contain a list of tuples describing which directories should be created and should attempt to be destroyed by the installation scripts. Each tuple holds the following values, separated by spaces: the directory name, its owner, its group and its numerical mode. For example:

  ```
  MAKE_DIRS_PERMS+=         ${VARBASE}/foo/private ${ROOT_USER} ${ROOT_GROUP} 0700
  ```

  The difference between the two is exactly the same as their non-`PERMS` counterparts.

15.1.2. File manipulation

Creating non-empty files outside the installation prefix is tricky because the `PLIST` forces all files to be inside it. To overcome this problem, the only solution is to extract the file in the known place (i.e., inside the installation prefix) and copy it to the appropriate location during installation (done by the installation scripts generated by pkginstall). We will call the former the *master file* in the following paragraphs, which describe the variables that can be used to automatically and consistently handle files outside the installation prefix:

- `CONF_FILES` and `REQD_FILES` are pairs of master and target files. During installation time, the master file is copied to the target one if and only if the latter does not exist. Upon deinstallation, the target file is removed provided that it was not modified by the installation.

  The difference between the two is that the latter prompts the administrator to remove any files that may be left after deinstallation (because they were not empty), while the former does not.
- `CONF_FILES_PERMS` and `REQD_FILES_PERMS` contain tuples describing master files as well as their target locations. For each of them, it also specifies their owner, their group and their numeric permissions, in this order. For example:

  ```
  REQD_FILES_PERMS+= ${PREFIX}/share/somefile ${VARBASE}/somefile ${ROOT_USER} ${ROOT_GROUP}
  ```

  The difference between the two is exactly the same as their non-`PERMS` counterparts.

15.2. Configuration files

Configuration files are special in the sense that they are installed in their own specific directory, `PKG_SYSCONFDIR`, and need special treatment during installation (most of which is automated by pkginstall). The main concept you must bear in mind is that files marked as configuration files are automatically copied to the right place (somewhere inside `PKG_SYSCONFDIR`) during installation *if and only if* they didn't exist before. Similarly, they will not be removed if they have local modifications. This ensures that administrators never lose any custom changes they may have made.

15.2.1. How PKG_SYSCONFDIR is set

As said before, the PKG_SYSCONFDIR variable specifies where configuration files shall be installed. Its contents are set based upon the following variables:

- PKG_SYSCONFBASE: The configuration's root directory. Defaults to ${PREFIX}/etc although it may be overridden by the user to point to his preferred location (e.g., /etc, /etc/pkg, etc.). Packages must not use it directly.
- PKG_SYSCONFSUBDIR: A subdirectory of PKG_SYSCONFBASE under which the configuration files for the package being built shall be installed. The definition of this variable only makes sense in the package's Makefile (i.e., it is not user-customizable).

  As an example, consider the Apache package, www/apache24, which places its configuration files under the httpd/ subdirectory of PKG_SYSCONFBASE. This should be set in the package Makefile.
- PKG_SYSCONFVAR: Specifies the name of the variable that holds this package's configuration directory (if different from PKG_SYSCONFBASE). It defaults to PKGBASE's value, and is always prefixed with PKG_SYSCONFDIR.
- PKG_SYSCONFDIR.${PKG_SYSCONFVAR}: Holds the directory where the configuration files for the package identified by PKG_SYSCONFVAR's shall be placed.

Based on the above variables, pkginstall determines the value of PKG_SYSCONFDIR, which is the *only* variable that can be used within a package to refer to its configuration directory. The algorithm used to set its value is basically the following:

1. If PKG_SYSCONFDIR.${PKG_SYSCONFVAR} is set, its value is used.
2. If the previous variable is not defined but PKG_SYSCONFSUBDIR is set in the package's Makefile, the resulting value is ${PKG_SYSCONFBASE}/${PKG_SYSCONFSUBDIR}.
3. Otherwise, it is set to ${PKG_SYSCONFBASE}.

It is worth mentioning that ${PKG_SYSCONFDIR} is automatically added to OWN_DIRS. See Section 15.1.1 what this means. This does not apply to subdirectories of ${PKG_SYSCONFDIR}, they still have to be created with OWN_DIRS or MAKE_DIRS.

15.2.2. Telling the software where configuration files are

Given that pkgsrc (and users!) expect configuration files to be in a known place, you need to teach each package where it shall install its files. In some cases you will have to patch the package Makefiles to achieve it. If you are lucky, though, it may be as easy as passing an extra flag to the configuration script; this is the case of GNU Autoconf- generated files:

```
CONFIGURE_ARGS+= --sysconfdir=${PKG_SYSCONFDIR}
```

Note that this specifies where the package has to *look for* its configuration files, not where they will be originally installed (although the difference is never explicit, unfortunately).

15.2.3. Patching installations

As said before, pkginstall automatically handles configuration files. This means that **the packages themselves must not touch the contents of `${PKG_SYSCONFDIR}` directly**. Bad news is that many software installation scripts will, out of the box, mess with the contents of that directory. So what is the correct procedure to fix this issue?

You must teach the package (usually by manually patching it) to install any configuration files under the examples hierarchy, `share/examples/${PKGBASE}/`. This way, the `PLIST` registers them and the administrator always has the original copies available.

Once the required configuration files are in place (i.e., under the examples hierarchy), the pkginstall framework can use them as master copies during the package installation to update what is in `${PKG_SYSCONFDIR}`. To achieve this, the variables `CONF_FILES` and `CONF_FILES_PERMS` are used. Check out Section 15.1.2 for information about their syntax and their purpose. Here is an example, taken from the `mail/mutt` package:

```
EGDIR=        ${PREFIX}/share/doc/mutt/samples
CONF_FILES=   ${EGDIR}/Muttrc ${PKG_SYSCONFDIR}/Muttrc
```

Note that the `EGDIR` variable is specific to that package and has no meaning outside it.

15.2.4. Disabling handling of configuration files

The automatic copying of config files can be toggled by setting the environment variable `PKG_CONFIG` prior to package installation.

15.3. System startup scripts

System startup scripts are special files because they must be installed in a place known by the underlying OS, usually outside the installation prefix. Therefore, the same rules described in Section 15.1 apply, and the same solutions can be used. However, pkginstall provides a special mechanism to handle these files.

In order to provide system startup scripts, the package has to:

1. Store the script inside `${FILESDIR}`, with the `.sh` suffix appended. Considering the `print/cups` package as an example, it has a `cupsd.sh` in its files directory.
2. Tell pkginstall to handle it, appending the name of the script, without its extension, to the `RCD_SCRIPTS` variable. Continuing the previous example:

   ```
   RCD_SCRIPTS+=   cupsd
   ```

Once this is done, pkginstall will do the following steps for each script in an automated fashion:

1. Process the file found in the files directory applying all the substitutions described in the `FILES_SUBST` variable.
2. Copy the script from the files directory to the examples hierarchy, `${PREFIX}/share/examples/rc.d/`. Note that this master file must be explicitly registered in the `PLIST`.

3. Add code to the installation scripts to copy the startup script from the examples hierarchy into the system-wide startup scripts directory.

15.3.1. Disabling handling of system startup scripts

The automatic copying of config files can be toggled by setting the environment variable `PKG_RCD_SCRIPTS` prior to package installation. Note that the scripts will be always copied inside the examples hierarchy, `${PREFIX}/share/examples/rc.d/`, no matter what the value of this variable is.

15.4. System users and groups

If a package needs to create special users and/or groups during installation, it can do so by using the pkginstall framework.

Users can be created by adding entries to the `PKG_USERS` variable. Each entry has the following syntax:

```
user:group
```

Further specification of user details may be done by setting per-user variables. `PKG_UID.`*`user`* is the numeric UID for the user. `PKG_GECOS.`*`user`* is the user's description or comment. `PKG_HOME.`*`user`* is the user's home directory, and defaults to `/nonexistent` if not specified. `PKG_SHELL.`*`user`* is the user's shell, and defaults to `/sbin/nologin` if not specified.

Similarly, groups can be created by adding entries to the `PKG_GROUPS` variable, whose syntax is:

```
group
```

The numeric GID of the group may be set by defining `PKG_GID.`*`group`*.

If a package needs to create the users and groups at an earlier stage, then it can set `USERGROUP_PHASE` to either `configure` or `build` to indicate the phase before which the users and groups are created. In this case, the numeric UIDs and GIDs of the created users and groups are automatically hardcoded into the final installation scripts.

15.5. System shells

Packages that install system shells should register them in the shell database, `/etc/shells`, to make things easier to the administrator. This must be done from the installation scripts to keep binary packages working on any system. pkginstall provides an easy way to accomplish this task.

When a package provides a shell interpreter, it has to set the `PKG_SHELL` variable to its absolute file name. This will add some hooks to the installation scripts to handle it. Consider the following example, taken from `shells/zsh`:

```
PKG_SHELL=      ${PREFIX}/bin/zsh
```

15.5.1. Disabling shell registration

The automatic registration of shell interpreters can be disabled by the administrator by setting the `PKG_REGISTER_SHELLS` environment variable to `NO`.

15.6. Fonts

Packages that install X11 fonts should update the database files that index the fonts within each fonts directory. This can easily be accomplished within the pkginstall framework.

When a package installs X11 fonts, it must list the directories in which fonts are installed in the `FONTS_DIRS.`*`type`* variables, where *`type`* can be one of “ttf”, “type1” or “x11”. This will add hooks to the installation scripts to run the appropriate commands to update the fonts database files within each of those directories. For convenience, if the directory path is relative, it is taken to be relative to the package’s installation prefix. Consider the following example, taken from `fonts/dbz-ttf`:

```
FONTS_DIRS.ttf= ${PREFIX}/share/fonts/X11/TTF
```

15.6.1. Disabling automatic update of the fonts databases

The automatic update of fonts databases can be disabled by the administrator by setting the `PKG_UPDATE_FONTS_DB` environment variable to `NO`.

Chapter 16.
Options handling

Many packages have the ability to be built to support different sets of features. `bsd.options.mk` is a framework in pkgsrc that provides generic handling of those options that determine different ways in which the packages can be built. It's possible for the user to specify exactly which sets of options will be built into a package or to allow a set of global default options apply.

There are two broad classes of behaviors that one might want to control via options. One is whether some particular feature is enabled in a program that will be built anyway, often by including or not including a dependency on some other package. The other is whether or not an additional program will be built as part of the package. Generally, it is better to make a split package for such additional programs instead of using options, because it enables binary packages to be built which can then be added separately. For example, the foo package might have minimal dependencies (those packages without which foo doesn't make sense), and then the foo-gfoo package might include the GTK frontend program gfoo. This is better than including a gtk option to foo that adds gfoo, because either that option is default, in which case binary users can't get foo without gfoo, or not default, in which case they can't get gfoo. With split packages, they can install foo without having GTK, and later decide to install gfoo (pulling in GTK at that time). This is an advantage to source users too, avoiding the need for rebuilds.

Plugins with widely varying dependencies should usually be split instead of options.

It is often more work to maintain split packages, especially if the upstream package does not support this. The decision of split vs. option should be made based on the likelihood that users will want or object to the various pieces, the size of the dependencies that are included, and the amount of work.

A further consideration is licensing. Non-free parts, or parts that depend on non-free dependencies (especially plugins) should almost always be split if feasible.

16.1. Global default options

Global default options are listed in `PKG_DEFAULT_OPTIONS`, which is a list of the options that should be built into every package if that option is supported. This variable should be set in `mk.conf`.

16.2. Converting packages to use `bsd.options.mk`

The following example shows how `bsd.options.mk` should be used by the hypothetical "wibble" package, either in the package `Makefile`, or in a file, e.g. `options.mk`, that is included by the main package `Makefile`.

```
PKG_OPTIONS_VAR=                PKG_OPTIONS.wibble
PKG_SUPPORTED_OPTIONS=          wibble-foo ldap
PKG_OPTIONS_OPTIONAL_GROUPS=    database
PKG_OPTIONS_GROUP.database=     mysql pgsql
```

```
PKG_SUGGESTED_OPTIONS=          wibble-foo
PKG_OPTIONS_LEGACY_VARS+=       WIBBLE_USE_OPENLDAP:ldap
PKG_OPTIONS_LEGACY_OPTS+=       foo:wibble-foo

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

# this package was previously named wibble2
.if defined(PKG_OPTIONS.wibble2)
PKG_LEGACY_OPTIONS+=            ${PKG_OPTIONS.wibble2}
PKG_OPTIONS_DEPRECATED_WARNINGS+= \
        "Deprecated variable PKG_OPTIONS.wibble2 used, use ${PKG_OPTIONS_VAR} instead."
.endif

.include "../../mk/bsd.options.mk"

# Package-specific option-handling

###
### FOO support
###
.if !empty(PKG_OPTIONS:Mwibble-foo)
CONFIGURE_ARGS+=    --enable-foo
.endif

###
### LDAP support
###
.if !empty(PKG_OPTIONS:Mldap)
.  include "../../databases/openldap-client/buildlink3.mk"
CONFIGURE_ARGS+=    --enable-ldap=${BUILDLINK_PREFIX.openldap-client}
.endif

###
### database support
###
.if !empty(PKG_OPTIONS:Mmysql)
.  include "../../mk/mysql.buildlink3.mk"
.endif
.if !empty(PKG_OPTIONS:Mpgsql)
.  include "../../mk/pgsql.buildlink3.mk"
.endif
```

The first section contains the information about which build options are supported by the package, and any default options settings if needed.

1. `PKG_OPTIONS_VAR` is the name of the make(1) variable that the user can set to override the default options. It should be set to PKG_OPTIONS.*`pkgbase`*. Do not set it to PKG_OPTIONS.${PKGBASE}, since `PKGBASE` is not defined at the point where the options are processed.
2. `PKG_SUPPORTED_OPTIONS` is a list of build options supported by the package.

3. PKG_OPTIONS_OPTIONAL_GROUPS is a list of names of groups of mutually exclusive options. The options in each group are listed in PKG_OPTIONS_GROUP.*groupname*. The most specific setting of any option from the group takes precedence over all other options in the group. Options from the groups will be automatically added to PKG_SUPPORTED_OPTIONS.
4. PKG_OPTIONS_REQUIRED_GROUPS is like PKG_OPTIONS_OPTIONAL_GROUPS, but building the packages will fail if no option from the group is selected.
5. PKG_OPTIONS_NONEMPTY_SETS is a list of names of sets of options. At least one option from each set must be selected. The options in each set are listed in PKG_OPTIONS_SET.*setname*. Options from the sets will be automatically added to PKG_SUPPORTED_OPTIONS. Building the package will fail if no option from the set is selected.
6. PKG_SUGGESTED_OPTIONS is a list of build options which are enabled by default.
7. PKG_OPTIONS_LEGACY_VARS is a list of "*USE_VARIABLE*:*option*" pairs that map legacy mk.conf variables to their option counterparts. Pairs should be added with "+=" to keep the listing of global legacy variables. A warning will be issued if the user uses a legacy variable.
8. PKG_OPTIONS_LEGACY_OPTS is a list of "*old-option*:*new-option*" pairs that map options that have been renamed to their new counterparts. Pairs should be added with "+=" to keep the listing of global legacy options. A warning will be issued if the user uses a legacy option.
9. PKG_LEGACY_OPTIONS is a list of options implied by deprecated variables used. This can be used for cases that neither PKG_OPTIONS_LEGACY_VARS nor PKG_OPTIONS_LEGACY_OPTS can handle, e. g. when PKG_OPTIONS_VAR is renamed.
10. PKG_OPTIONS_DEPRECATED_WARNINGS is a list of warnings about deprecated variables or options used, and what to use instead.

A package should never modify PKG_DEFAULT_OPTIONS or the variable named in PKG_OPTIONS_VAR. These are strictly user-settable. To suggest a default set of options, use PKG_SUGGESTED_OPTIONS.

PKG_OPTIONS_VAR must be defined before including bsd.options.mk. If none of PKG_SUPPORTED_OPTIONS, PKG_OPTIONS_OPTIONAL_GROUPS, and PKG_OPTIONS_REQUIRED_GROUPS are defined (as can happen with platform-specific options if none of them is supported on the current platform), PKG_OPTIONS is set to the empty list and the package is otherwise treated as not using the options framework.

After the inclusion of bsd.options.mk, the variable PKG_OPTIONS contains the list of selected build options, properly filtered to remove unsupported and duplicate options.

The remaining sections contain the logic that is specific to each option. The correct way to check for an option is to check whether it is listed in PKG_OPTIONS:

```
.if !empty(PKG_OPTIONS:Moption)
```

16.3. Option Names

Options that enable similar features in different packages (like optional support for a library) should use a common name in all packages that support it (like the name of the library). If another package already has an option with the same meaning, use the same name.

Options that enable features specific to one package, where it's unlikely that another (unrelated) package has the same (or a similar) optional feature, should use a name prefixed with *pkgname-*.

If a group of related packages share an optional feature specific to that group, prefix it with the name of the "main" package (e. g. `djbware-errno-hack`).

For new options, add a line to `mk/defaults/options.description`. Lines have two fields, separated by tab. The first field is the option name, the second its description. The description should be a whole sentence (starting with an uppercase letter and ending with a period) that describes what enabling the option does. E. g. "Enable ispell support." The file is sorted by option names.

16.4. Determining the options of dependencies

When writing `buildlink3.mk` files, it is often necessary to list different dependencies based on the options with which the package was built. For querying these options, the file `pkgsrc/mk/pkg-build-options.mk` should be used. A typical example looks like this:

```
pkgbase := libpurple
.include "../../mk/pkg-build-options.mk"

.if !empty(PKG_BUILD_OPTIONS.libpurple:Mdbus)
...
.endif
```

Including `pkg-build-options.mk` here will set the variable `PKG_BUILD_OPTIONS.libpurple` to the build options of the libpurple package, which can then be queried like `PKG_OPTIONS` in the `options.mk` file. See the file `pkg-build-options.mk` for more details.

Chapter 17.
The build process

17.1. Introduction

This chapter gives a detailed description on how a package is built. Building a package is separated into different *phases* (for example `fetch`, `build`, `install`), all of which are described in the following sections. Each phase is split into so-called *stages*, which take the name of the containing phase, prefixed by one of `pre-`, `do-` or `post-`. (Examples are `pre-configure`, `post-build`.) Most of the actual work is done in the `do-*` stages.

Never override the regular targets (like `fetch`), if you have to, override the `do-*` ones instead.

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 pkgsrc-specific patches to compile properly 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.

To get more details about what is happening at each step, you can set the `PKG_VERBOSE` variable, or the `PATCH_DEBUG` variable if you are just interested in more details about the *patch* step.

17.2. Program location

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. The value of `PREFIX` needs to be put into the various places in the program's source where paths to these files are encoded. See Section 11.3 and Section 19.3.1 for more details.

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

- `PREFIX` always points to the location where the current pkg will be installed. When referring to a pkg'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}". The name `LOCALBASE` stems from FreeBSD, which installed all packages in `/usr/local`. As pkgsrc leaves `/usr/local` for the system administrator, this variable is a misnomer.
- `X11BASE` is where the actual X11 distribution (from xsrc, etc.) is installed. When looking for *standard* X11 includes (not those installed by a package), use "${X11BASE}".
- X11-based packages are special in that they may be installed in either `X11BASE` or `LOCALBASE`.

Usually, X11 packages should be installed under LOCALBASE whenever possible. Note that you will need to include ../../mk/x11.buildlink3.mk in them to request the presence of X11 and to get the right compilation flags.

Even though, there are some packages that cannot be installed under LOCALBASE: those that come with app-defaults files. These packages are special and they must be placed under X11BASE. To accomplish this, set either USE_X11BASE or USE_IMAKE in your package.

Some notes: If you need to find includes or libraries installed by a pkg that has USE_IMAKE or USE_X11BASE in its pkg Makefile, you need to look in *both* ${X11BASE} and ${LOCALBASE}. To force installation of all X11 packages in LOCALBASE, the pkgtools/xpkgwedge package is enabled by default.

- X11PREFIX should be used to refer to the installed location of an X11 package. X11PREFIX will be set to X11BASE if xpkgwedge is not installed, and to LOCALBASE if xpkgwedge is installed.
- If xpkgwedge is installed, it is possible to have some packages installed in X11BASE and some in LOCALBASE. To determine the prefix of an installed package, the EVAL_PREFIX definition can be used. It takes pairs in the format "DIRNAME=<package>", and the make(1) variable DIRNAME will be set to the prefix of the installed package <package>, or "${X11PREFIX}" if the package is not installed.

  This is best illustrated by example.

  The following lines are taken from pkgsrc/wm/scwm/Makefile:

  ```
  EVAL_PREFIX+=           GTKDIR=gtk+
  CONFIGURE_ARGS+=        --with-guile-prefix=${LOCALBASE:Q}
  CONFIGURE_ARGS+=        --with-gtk-prefix=${GTKDIR:Q}
  CONFIGURE_ARGS+=        --enable-multibyte
  ```

  Specific defaults can be defined for the packages evaluated using EVAL_PREFIX, by using a definition of the form:

  ```
  GTKDIR_DEFAULT= ${LOCALBASE}
  ```

  where GTKDIR corresponds to the first definition in the EVAL_PREFIX pair.
- Within ${PREFIX}, packages should install files according to hier(7), with the exception that manual pages go into ${PREFIX}/man, not ${PREFIX}/share/man.

17.3. Directories used during the build process

When building a package, various directories are used to store source files, temporary files, pkgsrc-internal files, and so on. These directories are explained here.

Some of the directory variables contain relative pathnames. There are two common base directories for these relative directories: PKGSRCDIR/PKGPATH is used for directories that are pkgsrc-specific. WRKSRC is used for directories inside the package itself.

PKGSRCDIR

This is an absolute pathname that points to the pkgsrc root directory. Generally, you don't need it.

PKGDIR

This is an absolute pathname that points to the current package.

PKGPATH

This is a pathname relative to PKGSRCDIR that points to the current package.

WRKDIR

This is an absolute pathname pointing to the directory where all work takes place. The distfiles are extracted to this directory. It also contains temporary directories and log files used by the various pkgsrc frameworks, like *buildlink* or the *wrappers*.

WRKSRC

This is an absolute pathname pointing to the directory where the distfiles are extracted. It is usually a direct subdirectory of WRKDIR, and often it's the only directory entry that isn't hidden. This variable may be changed by a package Makefile.

The CREATE_WRKDIR_SYMLINK definition takes either the value *yes* or *no* and defaults to *no*. It indicates whether a symbolic link to the WRKDIR is to be created in the pkgsrc entry's directory. If users would like to have their pkgsrc trees behave in a read-only manner, then the value of CREATE_WRKDIR_SYMLINK should be set to *no*.

17.4. Running a phase

You can run a particular phase by typing **make phase**, where *phase* is the name of the phase. This will automatically run all phases that are required for this phase. The default phase is build, that is, when you run **make** without parameters in a package directory, the package will be built, but not installed.

17.5. The *fetch* phase

The first step in building a package is to fetch the distribution files (distfiles) from the sites that are providing them. This is the task of the *fetch* phase.

17.5.1. What to fetch and where to get it from

In simple cases, MASTER_SITES defines all URLs from where the distfile, whose name is derived from the DISTNAME variable, is fetched. The more complicated cases are described below.

The variable DISTFILES specifies the list of distfiles that have to be fetched. Its value defaults to ${DEFAULT_DISTFILES} and its value is ${DISTNAME}${EXTRACT_SUFX}, so that most packages don't need to define it at all. EXTRACT_SUFX is .tar.gz by default, but can be changed freely. Note that if your package requires additional distfiles to the default one, you cannot just append the additional filenames using the += operator, but you have write for example:

```
DISTFILES=      ${DEFAULT_DISTFILES} additional-files.tar.gz
```

Each distfile is fetched from a list of sites, usually MASTER_SITES. If the package has multiple DISTFILES or multiple PATCHFILES from different sites, you can set SITES.*distfile* to the list of URLs where the file *distfile* (including the suffix) can be found.

```
DISTFILES=      ${DISTNAME}${EXTRACT_SUFX}
```

```
DISTFILES+=     foo-file.tar.gz
SITES.foo-file.tar.gz= \
http://www.somewhere.com/somehow/ \
http://www.somewhereelse.com/mirror/somehow/
```

When actually fetching the distfiles, each item from `MASTER_SITES` or `SITES.*` gets the name of each distfile appended to it, without an intermediate slash. Therefore, all site values have to end with a slash or other separator character. This allows for example to set `MASTER_SITES` to a URL of a CGI script that gets the name of the distfile as a parameter. In this case, the definition would look like:

```
MASTER_SITES=   http://www.example.com/download.cgi?file=
```

The exception to this rule are URLs starting with a dash. In that case the URL is taken as is, fetched and the result stored under the name of the distfile. You can use this style for the case when the download URL style does not match the above common case. For example, if permanent download URL is a redirector to the real download URL, or the download file name is offered by an HTTP Content-Disposition header. In the following example, `foo-1.0.0.tar.gz` will be created instead of the default `v1.0.0.tar.gz`.

```
DISTNAME=       foo-1.0.0
MASTER_SITES=   -http://www.example.com/archive/v1.0.0.tar.gz
```

There are some predefined values for `MASTER_SITES`, which can be used in packages. The names of the variables should speak for themselves.

```
${MASTER_SITE_APACHE}
${MASTER_SITE_BACKUP}
${MASTER_SITE_CYGWIN}
${MASTER_SITE_DEBIAN}
${MASTER_SITE_FREEBSD}
${MASTER_SITE_FREEBSD_LOCAL}
${MASTER_SITE_GENTOO}
${MASTER_SITE_GNOME}
${MASTER_SITE_GNU}
${MASTER_SITE_GNUSTEP}
${MASTER_SITE_HASKELL_HACKAGE}
${MASTER_SITE_IFARCHIVE}
${MASTER_SITE_KDE}
${MASTER_SITE_MOZILLA}
${MASTER_SITE_MOZILLA_ALL}
${MASTER_SITE_MOZILLA_ESR}
${MASTER_SITE_MYSQL}
${MASTER_SITE_NETLIB}
${MASTER_SITE_OPENOFFICE}
${MASTER_SITE_PERL_CPAN}
${MASTER_SITE_PGSQL}
${MASTER_SITE_RUBYGEMS}
${MASTER_SITE_R_CRAN}
${MASTER_SITE_SOURCEFORGE}
${MASTER_SITE_SOURCEFORGE_JP}
${MASTER_SITE_SUNSITE}
${MASTER_SITE_SUSE}
${MASTER_SITE_TEX_CTAN}
```

```
${MASTER_SITE_XCONTRIB}
${MASTER_SITE_XEMACS}
${MASTER_SITE_XORG}
```

Some explanations for the less self-explaining ones: `MASTER_SITE_BACKUP` contains backup sites for packages that are maintained in ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/${DIST_SUBDIR}. `MASTER_SITE_LOCAL` contains local package source distributions that are maintained in ftp://ftp.NetBSD.org/pub/pkgsrc/distfiles/LOCAL_PORTS/.

If you choose one of these predefined sites, you may want 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_SITES=   ${MASTER_SITE_GNU:=subdirectory/name/}
MASTER_SITES=   ${MASTER_SITE_SOURCEFORGE:=project_name/}
```

Note the trailing slash after the subdirectory name.

17.5.2. How are the files fetched?

The *fetch* phase makes sure that all the distfiles exist in a local directory (`DISTDIR`, which can be set by the pkgsrc user). If the files do not exist, they are fetched using commands of the form

```
${FETCH_CMD} ${FETCH_BEFORE_ARGS} ${site}${file} ${FETCH_AFTER_ARGS}
```

where `${site}` varies through several possibilities in turn: first, `MASTER_SITE_OVERRIDE` is tried, then the sites specified in either `SITES.file` if defined, else `MASTER_SITES` or `PATCH_SITES`, as applies, then finally the value of `MASTER_SITE_BACKUP`. The order of all except the first and the last can be optionally sorted by the user, via setting either `MASTER_SORT_RANDOM`, and `MASTER_SORT_AWK` or `MASTER_SORT_REGEX`.

The specific command and arguments used depend on the `FETCH_USING` parameter. The example above is for `FETCH_USING=custom`.

The distfiles mirror run by the NetBSD Foundation uses the *mirror-distfiles* target to mirror the distfiles, if they are freely distributable. Packages setting `NO_SRC_ON_FTP` (usually to "${RESTRICTED}") will not have their distfiles mirrored.

17.6. The *checksum* phase

After the distfile(s) are fetched, their checksum is generated and compared with the checksums stored in the distinfo 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.

17.7. The *extract* phase

When the distfiles are present on the local system, they need to be extracted, as they usually come in the form of some compressed archive format.

By default, all `DISTFILES` are extracted. If you only need some of them, you can set the `EXTRACT_ONLY` variable to the list of those files.

Extracting the files is usually done by a little program, `mk/extract/extract`, which already knows how to extract various archive formats, so most likely you will not need to change anything here. But if you need, the following variables may help you:

`EXTRACT_OPTS_{BIN,LHA,PAX,RAR,TAR,ZIP,ZOO}`

Use these variables to override the default options for an extract command, which are defined in `mk/extract/extract`.

`EXTRACT_USING`

This variable can be set to `bsdtar`, `gtar`, `nbtar` (which is the default value), `pax`, or an absolute pathname pointing to the command with which tar archives should be extracted. It is preferred to choose bsdtar over gtar if NetBSD's pax-as-tar is not good enough.

If the `extract` program doesn't serve your needs, you can also override the `EXTRACT_CMD` variable, which holds the command used for extracting the files. This command is executed in the `${WRKSRC}` directory. During execution of this command, the shell variable `extract_file` holds the absolute pathname of the file that is going to be extracted.

And if that still does not suffice, you can override the `do-extract` target in the package Makefile.

17.8. The *patch* phase

After extraction, all the patches named by the `PATCHFILES`, those present in the patches subdirectory of the package as well as in $LOCALPATCHES/$PKGPATH (e.g. `/usr/local/patches/graphics/png`) 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 Section 11.3 for more details.

By default patch(1) is given special args to make it fail if the patches apply with some lines of fuzz. Please fix (regen) the patches so that they apply cleanly. The rationale behind this is that patches that don't apply cleanly may end up being applied in the wrong place, and cause severe harm there.

17.9. The *tools* phase

This is covered in Chapter 18.

17.10. The *wrapper* phase

This phase creates wrapper programs for the compilers and linkers. The following variables can be used to tweak the wrappers.

ECHO_WRAPPER_MSG

The command used to print progress messages. Does nothing by default. Set to ${ECHO} to see the progress messages.

WRAPPER_DEBUG

This variable can be set to yes (default) or no, depending on whether you want additional information in the wrapper log file.

WRAPPER_UPDATE_CACHE

This variable can be set to yes or no, depending on whether the wrapper should use its cache, which will improve the speed. The default value is yes, but is forced to no if the platform does not support it.

WRAPPER_REORDER_CMDS

A list of reordering commands. A reordering command has the form reorder:l:*lib1*:*lib2*. It ensures that that -l*lib1* occurs before -l*lib2*.

WRAPPER_TRANSFORM_CMDS

A list of transformation commands. [TODO: investigate further]

17.11. The *configure* phase

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

If the package contains a configure script, this can be invoked by setting HAS_CONFIGURE to "yes". If the configure script is a GNU autoconf script, you should set GNU_CONFIGURE to "yes" instead. What happens in the *configure* phase is roughly:

```
.for d in ${CONFIGURE_DIRS}
        cd ${WRKSRC} \
        && cd ${d} \
        && env ${CONFIGURE_ENV} ${CONFIGURE_SCRIPT} ${CONFIGURE_ARGS}
.endfor
```

CONFIGURE_DIRS (default: ".") is a list of pathnames relative to WRKSRC. In each of these directories, the configure script is run with the environment CONFIGURE_ENV and arguments CONFIGURE_ARGS. The variables CONFIGURE_ENV, CONFIGURE_SCRIPT (default: "./configure") and CONFIGURE_ARGS may all be changed by the package.

If the program uses the Perl way of configuration (mainly Perl modules, but not only), i.e. a file called Makefile.PL, it should include ../../lang/perl5/module.mk. To set any parameter for Makefile.PL use the MAKE_PARAMS variable (e.g., MAKE_PARAMS+=foo=bar

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.) You can add variables to xmkmf's environment by adding them to the SCRIPTS_ENV variable.

If the program uses cmake for configuration, the appropriate steps can be invoked by setting USE_CMAKE to "yes". You can add variables to cmake's environment by adding them to the CONFIGURE_ENV variable and arguments to cmake by adding them to the CMAKE_ARGS variable. The top directory argument is given by the CMAKE_ARG_PATH variable, that defaults to "." (relative to CONFIGURE_DIRS)

If there is no configure step at all, set NO_CONFIGURE to "yes".

17.12. The *build* phase

For building a package, a rough equivalent of the following code is executed.

```
.for d in ${BUILD_DIRS}
        cd ${WRKSRC} \
        && cd ${d} \
        && env ${MAKE_ENV} \
            ${MAKE_PROGRAM} ${BUILD_MAKE_FLAGS} \
                -f ${MAKE_FILE} \
                ${BUILD_TARGET}
.endfor
```

BUILD_DIRS (default: ".") is a list of pathnames relative to WRKSRC. In each of these directories, MAKE_PROGRAM is run with the environment MAKE_ENV and arguments BUILD_MAKE_FLAGS. The variables MAKE_ENV, BUILD_MAKE_FLAGS, MAKE_FILE and BUILD_TARGET may all be changed by the package.

The default value of MAKE_PROGRAM is "gmake" if USE_TOOLS contains "gmake", "make" otherwise. The default value of MAKE_FILE is "Makefile", and BUILD_TARGET defaults to "all".

If there is no build step at all, set NO_BUILD to "yes".

17.13. The *test* phase

[TODO]

17.14. The *install* phase

Once the build stage has completed, the final step is to install the software in public directories, so users can access the programs and files.

In the *install* phase, a rough equivalent of the following code is executed. Additionally, before and after this code, much magic is performed to do consistency checks, registering the package, and so on.

```
.for d in ${INSTALL_DIRS}
        cd ${WRKSRC} \
        && cd ${d} \
        && env ${MAKE_ENV} \
            ${MAKE_PROGRAM} ${INSTALL_MAKE_FLAGS} \
```

```
                -f ${MAKE_FILE} \
                ${INSTALL_TARGET}
.endfor
```

The variable's meanings are analogous to the ones in the *build* phase. `INSTALL_DIRS` defaults to `BUILD_DIRS`. `INSTALL_TARGET` is "install" by default, plus "install.man" if `USE_IMAKE` is defined and `NO_INSTALL_MANPAGES` is not defined.

In the *install* phase, the following variables are useful. They are all variations of the install(1) command that have the owner, group and permissions preset. `INSTALL` is the plain install command. The specialized variants, together with their intended use, are:

`INSTALL_PROGRAM_DIR`

directories that contain binaries

`INSTALL_SCRIPT_DIR`

directories that contain scripts

`INSTALL_LIB_DIR`

directories that contain shared and static libraries

`INSTALL_DATA_DIR`

directories that contain data files

`INSTALL_MAN_DIR`

directories that contain man pages

`INSTALL_PROGRAM`

binaries that can be stripped from debugging symbols

`INSTALL_SCRIPT`

binaries that cannot be stripped

`INSTALL_GAME`

game binaries

`INSTALL_LIB`

shared and static libraries

`INSTALL_DATA`

data files

`INSTALL_GAME_DATA`

data files for games

`INSTALL_MAN`

man pages

Some other variables are:

`INSTALLATION_DIRS`

A list of directories relative to `PREFIX` that are created by pkgsrc at the beginning of the *install* phase. The package is supposed to create all needed directories itself before installing files to it and list all other directories here.

In the rare cases that a package shouldn't install anything, set `NO_INSTALL` to "yes". This is mostly relevant for packages in the `regress` category.

17.15. The *package* phase

Once the install stage has completed, a binary package of the installed files can be built. These binary packages can be used for quick installation without previous compilation, e.g. by the **make bin-install** or by using **pkg_add**.

By default, the binary packages are created in `${PACKAGES}/All` and symlinks are created in `${PACKAGES}/`*`category`*, one for each category in the `CATEGORIES` variable. `PACKAGES` defaults to `pkgsrc/packages`.

17.16. Cleaning up

Once you're finished with a package, you can clean the work directory by running **make clean**. If you want to clean the work directories of all dependencies too, use **make clean-depends**.

17.17. 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 be performed from a package's Makefile, for example, which a program's configure script or install target omitted.

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.

This is the default value of `DEPENDS_TARGET` except in the case of **make update** and **make package**, where the defaults are “package” and “update”, respectively.

deinstall

This target does a pkg_delete(1) in the current directory, effectively de-installing the package. The following variables can be used to tune the behaviour:

`PKG_VERBOSE`

Add a "-v" 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 `pkgsrc/x11/kde`, this is likely to remove whole KDE. Works by adding “-R” to the pkg_delete(1) command line.

bin-install

Install a binary package from local disk and via FTP from a list of sites (see the `BINPKG_SITES` variable), and do a **make package** if no binary package is available anywhere. The arguments given to **pkg_add** can be set via `BIN_INSTALL_FLAGS` e.g., to do verbose operation, etc.

update

This target causes the current package to be updated to the latest version. The package and all depending packages first get de-installed, 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** (or whatever `UPDATE_TARGET` is set to) 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 `mk.conf` to alter the behaviour of **make update**:

`UPDATE_TARGET`

Install target to recursively use for the updated package and the dependent packages. Defaults to `DEPENDS_TARGET` if set, "install" otherwise for **make update**. Other good targets are "package" or "bin-install". Do not set this to "update" or you will get stuck in an endless loop!

`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`

Deinstall each package before installing (making `DEPENDS_TARGET`). This may be necessary if the "clean-update" target (see below) was called after interrupting a running **make update**.

`DEPENDS_TARGET`

Allows you to disable recursion and hardcode the target for packages. The default is "update" for the update target, facilitating a recursive update of prerequisite packages. Only set `DEPENDS_TARGET` if you want to disable recursive updates. Use `UPDATE_TARGET` instead to just set a specific target for each package to be installed during **make update** (see above).

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 de-installed (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 run **make update** and only if you have a dirty package tree (e.g., if you used `NOCLEAN`).

If you are unsure about whether your tree is clean, you can either perform a **make clean** at the top of the tree, or use the following sequence of commands from the directory of the package you want to update (*before* running **make update** for the first time, otherwise you lose all the packages you wanted to update!):

```
# make clean-update
# make clean CLEANDEPENDS=YES
# make update
```

The following variables can be used either on the command line or in `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).

replace

Update the installation of the current package. This differs from update in that it does not replace dependent packages. You will need to install `pkgtools/pkg_tarup` for this target to work.

Be careful when using this target! There are no guarantees that dependent packages will still work, in particular they will most certainly break if you **make replace** a library package whose shared library major version changed between your installed version and the new one. For this reason, this target is not officially supported and only recommended for advanced users.

info

This target invokes pkg_info(1) for the current package. You can use this to check which version of a package is installed.

index

This is a top-level command, i.e. it should be used in the `pkgsrc` directory. It creates a database of all packages in the local pkgsrc tree, including dependencies, comment, maintainer, and some other useful information. Individual entries are created by running **make describe** in the packages' directories. This index file is saved as `pkgsrc/INDEX`. It can be displayed in verbose format by running **make print-index**. You can search in it with **make search key=*something***. You can extract a list of all packages that depend on a particular one by running **make show-deps PKG=*somepackage***.

Running this command takes a very long time, some hours even on fast machines!

readme

This target generates a `README.html` file, which can be viewed using a browser such as `www/firefox` or `www/links`. 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.

The target can be run at the toplevel or in category directories, in which case it descends recursively.

readme-all

This is a top-level command, run it in `pkgsrc`. 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 (`ALLFILES`, which contains all `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 "show-host-specific-pkgs" target.

show-installed-depends

This target shows which installed packages match the current package's `DEPENDS`. Useful if out of date dependencies are causing build problems.

check-shlibs

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

print-PLIST

After a "make install" from a new or upgraded pkg, this prints out an attempt to generate a new `PLIST` from a **find -newer work/.extract_done**. An attempt is made to care for shared libs etc., but it is *strongly* recommended to review the result before putting it into `PLIST`. On upgrades, it's useful to diff the output of this command against an already existing `PLIST` file.

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 `PLIST`, as the "find -newer" command used by this target won't catch them!

See Section 13.3 for more information on this target.

bulk-package

Used to do bulk builds. If an appropriate binary package already exists, no action is taken. If not, this target will compile, install and package it (and its depends, if `PKG_DEPENDS` is set properly. See Chapter 7). After creating the binary package, the sources, the just-installed package and its required packages are removed, preserving free disk space.

Beware that this target may deinstall all packages installed on a system!

bulk-install

Used during bulk-installs to install required packages. If an up-to-date binary package is available, it will be installed via pkg_add(1). If not, **make bulk-package** will be executed, but the installed

binary won't be removed.

A binary package is considered "up-to-date" to be installed via pkg_add(1) if:

- None of the package's files (`Makefile`, ...) were modified since it was built.
- None of the package's required (binary) packages were modified since it was built.

Beware that this target may deinstall all packages installed on a system!

Chapter 18.
Tools needed for building or running

The USE_TOOLS definition is used both internally by pkgsrc and also for individual packages to define what commands are needed for building a package (like BUILD_DEPENDS) or for later run-time of an installed packaged (such as DEPENDS). If the native system provides an adequate tool, then in many cases, a pkgsrc package will not be used.

When building a package, the replacement tools are made available in a directory (as symlinks or wrapper scripts) that is early in the executable search path. Just like the buildlink system, this helps with consistent builds.

A tool may be needed to help build a specific package. For example, perl, GNU make (gmake) or yacc may be needed.

Also a tool may be needed, for example, because the native system's supplied tool may be inefficient for building a package with pkgsrc. For example, a package may need GNU awk, bison (instead of yacc) or a better sed.

The tools used by a package can be listed by running **make show-tools**.

18.1. Tools for pkgsrc builds

The default set of tools used by pkgsrc is defined in bsd.pkg.mk. This includes standard Unix tools, such as: **cat**, **awk**, **chmod**, **test**, and so on. These can be seen by running: **make show-var VARNAME=USE_TOOLS**.

If a package needs a specific program to build then the USE_TOOLS variable can be used to define the tools needed.

18.2. Tools needed by packages

In the following examples, the :run means that it is needed at run-time (and becomes a DEPENDS). The default is a build dependency which can be set with :build. (So in this example, it is the same as gmake:build and pkg-config:build.)

```
USE_TOOLS+=     gmake perl:run pkg-config
```

When using the tools framework, a TOOLS_PATH.foo variable is defined which contains the full path to the appropriate tool. For example, TOOLS_PATH.bash could be "/bin/bash" on Linux systems.

If you always need a pkgsrc version of the tool at run-time, then just use DEPENDS instead.

18.3. Tools provided by platforms

When improving or porting pkgsrc to a new platform, have a look at (or create) the corresponding platform specific make file fragment under `pkgsrc/mk/tools/tools.${OPSYS}.mk` which defines the name of the common tools. For example:

```
.if exists(/usr/bin/bzcat)
TOOLS_PLATFORM.bzcat?=          /usr/bin/bzcat
.elif exists(/usr/bin/bzip2)
TOOLS_PLATFORM.bzcat?=          /usr/bin/bzip2 -cd
.endif

TOOLS_PLATFORM.true?=           true                    # shell builtin
```

18.4. Questions regarding the tools

1. How do I add a new tool?

TODO

2. How do I get a list of all available tools?

TODO

3. How can I get a list of all the tools that a package is using while being built? I want to know whether it uses sed or not.

Currently, you can't. (TODO: But I want to be able to do it.)

Chapter 19.
Making your package work

19.1. General operation

19.1.1. Portability of packages

One appealing feature of pkgsrc is that it runs on many different platforms. As a result, it is important to ensure, where possible, that packages in pkgsrc are portable. This chapter mentions some particular details you should pay attention to while working on pkgsrc.

19.1.2. How to pull in user-settable variables from `mk.conf`

The pkgsrc user can configure pkgsrc by overriding several variables in the file pointed to by `MAKECONF`, which is `mk.conf` by default. When you want to use those variables in the preprocessor directives of make(1) (for example `.if` or `.for`), you need to include the file `../../mk/bsd.prefs.mk` before, which in turn loads the user preferences.

But note that some variables may not be completely defined after `../../mk/bsd.prefs.mk` has been included, as they may contain references to variables that are not yet defined. In shell commands this is no problem, since variables are actually macros, which are only expanded when they are used. But in the preprocessor directives mentioned above and in dependency lines (of the form `target: dependencies`) the variables are expanded at load time.

> **Note:** Currently there is no exhaustive list of all variables that tells you whether they can be used at load time or only at run time, but it is in preparation.

19.1.3. User interaction

Occasionally, packages require interaction from the user, and this can be in a number of ways:

- When fetching the distfiles, some packages require user interaction such as entering username/password or accepting a license on a web page.
- When extracting the distfiles, some packages may ask for passwords.
- help to configure the package before it is built
- help during the build process
- help during the installation of a package

The INTERACTIVE_STAGE definition is provided to notify the pkgsrc mechanism of an interactive stage which will be needed, and this should be set in the package's Makefile, e.g.:

```
INTERACTIVE_STAGE=      build
```

Multiple interactive stages can be specified:

```
INTERACTIVE_STAGE=      configure install
```

The user can then decide to skip this package by setting the BATCH variable.

19.1.4. Handling licenses

Authors of software can choose the licence under which software can be copied. This is due to copyright law, and reasons for license choices are outside the scope of pkgsrc. The pkgsrc system recognizes that there are a number of licenses which some users may find objectionable or difficult or impossible to comply with. The Free Software Foundation has declared some licenses "Free", and the Open Source Initiative has a definition of "Open Source". The pkgsrc system, as a policy choice, does not label packages which have licenses that are Free or Open Source. However, packages without a license meeting either of those tests are labeled with a license tag denoting the license. Note that a package with no license to copy trivially does not meet either the Free or Open Source test.

For packages which are not Free or Open Source, pkgsrc will not build the package unless the user has indicated to pkgsrc that packages with that particular license may be built. Note that this documentation avoids the term "accepted the license". The pkgsrc system is merely providing a mechanism to avoid accidentally building a package with a non-free license; judgement and responsibility remain with the user. (Installation of binary packages are not currently subject to this mechanism; this is a bug.)

One might want to only install packages with a BSD license, or the GPL, and not the other. The free licenses are added to the default ACCEPTABLE_LICENSES variable. The user can override the default by setting the ACCEPTABLE_LICENSES variable with "=" instead of "+=". The licenses accepted by default are:

```
apache-1.1 apache-2.0
arphic-public
artistic artistic-2.0
boost-license
cc-by-sa-v3.0
cc0-1.0-universal
cddl-1.0
cpl-1.0
epl-v1.0
gnu-fdl-v1.1 gnu-fdl-v1.2 gnu-fdl-v1.3
gnu-gpl-v1
gnu-gpl-v2 gnu-lgpl-v2 gnu-lgpl-v2.1
gnu-gpl-v3 gnu-lgpl-v3
ibm-public-license-1.0
ipafont
isc
lppl-1.3c
```

```
lucent
miros
mit
mpl-1.0 mpl-1.1 mpl-2.0
mplusfont
ofl-v1.0 ofl-v1.1
original-bsd modified-bsd 2-clause-bsd
php
png-license
postgresql-license
public-domain
python-software-foundation
qpl-v1.0
sgi-free-software-b-v2.0
sleepycat-public
unlicense
x11
zlib
zpl
```

The license tag mechanism is intended to address copyright-related issues surrounding building, installing and using a package, and not to address redistribution issues (see `RESTRICTED` and `NO_SRC_ON_FTP`, etc.). Packages with redistribution restrictions should set these tags.

Denoting that a package may be copied according to a particular license is done by placing the license in `pkgsrc/licenses` and setting the `LICENSE` variable to a string identifying the license, e.g. in `graphics/xv`:

```
LICENSE=        xv-license
```

When trying to build, the user will get a notice that the package is covered by a license which has not been placed in the `ACCEPTABLE_LICENSES` variable:

```
% make
===> xv-3.10anb9 has an unacceptable license: xv-license.
===>     To view the license, enter "/usr/bin/make show-license".
===>     To indicate acceptance, add this line to your /etc/mk.conf:
===>     ACCEPTABLE_LICENSES+=xv-license
*** Error code 1
```

The license can be viewed with **make show-license**, and if the user so chooses, the line printed above can be added to `mk.conf` to convey to pkgsrc that it should not in the future fail because of that license:

```
ACCEPTABLE_LICENSES+=xv-license
```

When adding a package with a new license, the following steps are required:

1. Check if the file can avoid the -license filename tag as described above by referencing Various Licenses and Comments about Them (http://www.gnu.org/licenses/license-list.html) and Licenses by Name | Open Source Initiative (http://opensource.org/licenses/alphabetical). If this is the case, additionally add the license filename to:
   - DEFAULT_ACCEPTABLE_LICENSES in `pkgsrc/mk/license.mk`
   - default_acceptable_licenses in `pkgsrc/pkgtools/pkg_install/files/lib/license.c`
   - the ACCEPTABLE_LICENSES list in `pkgsrc/doc/guide/files/fixes.xml`

   with the proper syntax as demonstrated in those files, respectively.
2. The license text should be added to `pkgsrc/licenses` for displaying. A list of known licenses can be seen in this directory.

When the license changes (in a way other than formatting), please make sure that the new license has a different name (e.g., append the version number if it exists, or the date). Just because a user told pkgsrc to build programs under a previous version of a license does not mean that pkgsrc should build programs under the new licenses. The higher-level point is that pkgsrc does not evaluate licenses for reasonableness; the only test is a mechanistic test of whether a particular text has been approved by either of two bodies.

The use of `LICENSE=shareware`, `LICENSE=no-commercial-use`, and similar language is deprecated because it does not crisply refer to a particular license text. Another problem with such usage is that it does not enable a user to tell pkgsrc to proceed for a single package without also telling pkgsrc to proceed for all packages with that tag.

19.1.5. Restricted packages

Some licenses restrict how software may be re-distributed. Because a license tag is required unless the package is Free or Open Source, all packages with restrictions should have license tags. By declaring the restrictions, package tools can automatically refrain from e.g. placing binary packages on FTP sites.

There are four restrictions that may be encoded, which are the cross product of sources (distfiles) and binaries not being placed on FTP sites and CD-ROMs. Because this is rarely the exact language in any license, and because non-Free licenses tend to be different from each other, pkgsrc adopts a definition of FTP and CD-ROM. Pkgsrc uses "FTP" to mean that the source or binary file should not be made available over the Internet at no charge. Pkgsrc uses "CD-ROM" to mean that the source or binary may not be made available on some kind of media, together with other source and binary packages, and which is sold for a distribution charge.

In order to encode these restrictions, the package system defines five make variables that can be set to note these restrictions:

- `RESTRICTED`

  This variable should be set whenever a restriction exists (regardless of its kind). Set this variable to a string containing the reason for the restriction. It should be understood that those wanting to understand the restriction will have to read the license, and perhaps seek advice of counsel.
- `NO_BIN_ON_CDROM`

Binaries may not be placed on CD-ROM containing other binary packages, for which a distribution charge may be made. In this case, set this variable to `${RESTRICTED}`.

- `NO_BIN_ON_FTP`

  Binaries may not made available on the Internet without charge. In this case, set this variable to `${RESTRICTED}`. If this variable is set, binary packages will not be included on ftp.NetBSD.org.

- `NO_SRC_ON_CDROM`

  Distfiles may not be placed on CD-ROM, together with other distfiles, for which a fee may be charged. In this case, set this variable to `${RESTRICTED}`.

- `NO_SRC_ON_FTP`

  Distfiles may not made available via FTP at no charge. In this case, set this variable to `${RESTRICTED}`. If this variable is set, the distfile(s) will not be mirrored on ftp.NetBSD.org.

Please note that packages will to be removed from pkgsrc when the distfiles are not distributable and cannot be obtained for a period of one full quarter branch. Packages with manual / interactive fetch must have a maintainer and it is his/her responsibility to ensure this.

19.1.6. Handling dependencies

Your package may depend on some other package being present - and there are various ways of expressing this dependency. pkgsrc supports the `BUILD_DEPENDS` and `DEPENDS` definitions, the `USE_TOOLS` definition, as well as dependencies via `buildlink3.mk`, which is the preferred way to handle dependencies, and which uses the variables named above. See Chapter 14 for more information.

The basic difference between the two variables is as follows: The `DEPENDS` definition registers that pre-requisite in the binary package so it will be pulled in when the binary package is later installed, whilst the `BUILD_DEPENDS` definition does not, marking a dependency that is only needed for building the package.

This means that if you only need a package present whilst you are building, it should be noted as a `BUILD_DEPENDS`.

The format for a `BUILD_DEPENDS` and a `DEPENDS` definition is:

```
<pre-req-package-name>:../../<category>/<pre-req-package>
```

Please note that the “pre-req-package-name” may include any of the wildcard version numbers recognized by pkg_info(1).

1. If your package needs another package’s binaries or libraries to build and run, and if that package has a `buildlink3.mk` file available, use it:

   ```
   .include "../../graphics/jpeg/buildlink3.mk"
   ```

2. If your package needs another package’s binaries or libraries only for building, and if that package has a `buildlink3.mk` file available, use it:

   ```
   .include "../../graphics/jpeg/buildlink3.mk"
   ```

but set `BUILDLINK_DEPMETHOD.jpeg?=build` to make it a build dependency only. This case is rather rare.

3. If your package needs binaries from another package to build, use the `BUILD_DEPENDS` definition:

   ```
   BUILD_DEPENDS+= scons-[0-9]*:../../devel/scons
   ```

4. If your package needs a library with which to link and there is no `buildlink3.mk` file available, create one. Using `DEPENDS` won't be sufficient because the include files and libraries will be hidden from the compiler.

5. If your package needs some executable to be able to run correctly and if there's no `buildlink3.mk` file, this is specified using the `DEPENDS` variable. 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-[0-9]*:../../print/teTeX
   ```

6. You can use wildcards in package dependencies. Note that such wildcard dependencies are retained when creating binary packages. The dependency is checked when installing the binary package and any package which matches the pattern will be used. Wildcard dependencies should be used with care.

   The "-[0-9]*" should be used instead of "-*" to avoid potentially ambiguous matches such as "tk-postgresql" matching a "tk-*" `DEPENDS`.

   Wildcards can also be used to specify that a package will only build against a certain minimum version of a pre-requisite:

   ```
   DEPENDS+=       ImageMagick>=6.0:../../graphics/ImageMagick
   ```

   This means that the package will build using version 6.0 of ImageMagick or newer. Such a dependency may be warranted if, for example, the command line options of an executable have changed.

   If you need to depend on minimum versions of libraries, see the buildlink section of the pkgsrc guide.

   For security fixes, please update the package vulnerabilities file. See Section 19.1.10 for more information.

7. If the package depends on either one of two (or more) packages, specify the "pre-req-package-name" as a comma-separated list between curly braces.

   As an example, take a package that depends on the Perl "version" module, which has been part of Perl itself since version 5.10.0. This either/or dependency is expressed as:

   ```
   DEPENDS+= {perl>=5.10.0,p5-version-[0-9]*}:../../devel/p5-version
   ```

If your package needs files from another package to build, add the relevant distribution files to `DISTFILES`, so they will be extracted automatically. See the `print/ghostscript` package for an example. (It relies on the jpeg sources being present in source form during the build.)

19.1.7. Handling 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 as another package in the pkgsrc tree or has the same PKGNAME.

These cases are handled automatically by the packaging tools at package installation time and do not need to be handled manually.

In case the conflicts can not be recognized automatically (e.g., packages using the same config file location but no other shared files), you can set CONFLICTS to a space-separated list of packages (including version string) your package conflicts with.

For example, if both foo/bar and foo/baz use the same config file, you would set in foo/bar/Makefile:

```
CONFLICTS=      baz-[0-9]*
```

and in pkgsrc/foo/baz/Makefile:

```
CONFLICTS=      bar-[0-9]*
```

19.1.8. Packages that cannot or should not be built

There are several reasons why a package might be instructed to not build under certain circumstances. If the package builds and runs on most platforms, the exceptions should be noted with BROKEN_ON_PLATFORM. If the package builds and runs on a small handful of platforms, set BROKEN_EXCEPT_ON_PLATFORM instead. Both BROKEN_ON_PLATFORM and BROKEN_EXCEPT_ON_PLATFORM are OS triples (OS-version-platform) that can use glob-style wildcards.

If a package is not appropriate for some platforms (as oopposed to merely broken), a different set of variables should be used as this affects failure reporting and statistics. If the package is appropriate for most platforms, the exceptions should be noted with NOT_FOR_PLATFORM. If the package is appropriate for only a small handful of platforms (often exactly one), set ONLY_FOR_PLATFORM instead. Both ONLY_FOR_PLATFORM and NOT_FOR_PLATFORM are OS triples (OS-version-platform) that can use glob-style wildcards.

Some packages are tightly bound to a specific version of an operating system, e.g. LKMs or sysutils/lsof. Such binary packages are not backwards compatible with other versions of the OS, and should be uploaded to a version specific directory on the FTP server. Mark these packages by setting OSVERSION_SPECIFIC to “yes”. This variable is not currently used by any of the package system internals, but may be used in the future.

If the package should be skipped (for example, because it provides functionality already provided by the system), set PKG_SKIP_REASON to a descriptive message. If the package should fail because some preconditions are not met, set PKG_FAIL_REASON to a descriptive message.

19.1.9. Packages which should not be deleted, once installed

To ensure that a package may not be deleted, once it has been installed, the PKG_PRESERVE definition should be set in the package Makefile. This will be carried into any binary package that is made from this

pkgsrc entry. A “preserved” package will not be deleted using pkg_delete(1) unless the “-f” option is used.

19.1.10. Handling packages with security problems

When a vulnerability is found, this should be noted in `localsrc/security/advisories/pkg-vulnerabilities`, and after committing that file, ask pkgsrc-security@NetBSD.org to update the file on ftp.NetBSD.org.

After fixing the vulnerability by a patch, its `PKGREVISION` should be increased (this is of course not necessary if the problem is fixed by using a newer release of the software), and the pattern in the pkg-vulnerabilities file must be updated.

Also, if the fix should be applied to the stable pkgsrc branch, be sure to submit a pullup request!

Binary packages already on ftp.NetBSD.org will be handled semi-automatically by a weekly cron job.

19.1.11. 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, a “nb1”, “nb2”, ... suffix can be used on package versions by setting `PKGREVISION=1` (2, ...). The “nb” is treated like a “.” by the package tools. e.g.

```
DISTNAME=       foo-17.42
PKGREVISION=    9
```

will result in a `PKGNAME` of “foo-17.42nb9”. If you want to use the original value of `PKGNAME` without the “nbX” suffix, e.g. for setting `DIST_SUBDIR`, use `PKGNAME_NOREV`.

When a new release of the package is released, the `PKGREVISION` should be removed, e.g. on a new minor release of the above package, things should be like:

```
DISTNAME=       foo-17.43
```

`PKGREVISION` should be incremented for any non-trivial change in the resulting binary package. Without a `PKGREVISION` bump, someone with the previous version installed has no way of knowing that their package is out of date. Thus, changes without increasing `PKGREVISION` are essentially labeled "this is so trivial that no reasonable person would want to upgrade", and this is the rough test for when increasing `PKGREVISION` is appropriate. Examples of changes that do not merit increasing `PKGREVISION` are:

- Changing `HOMEPAGE`, `MAINTAINER`, `OWNER`, or comments in Makefile.
- Changing build variables if the resulting binary package is the same.
- Changing `DESCR`.
- Adding `PKG_OPTIONS` if the default options don’t change.

Examples of changes that do merit an increase to `PKGREVISION` include:

- Security fixes
- Changes or additions to a patch file
- Changes to the PLIST
- A dependency is changed or renamed.

PKGREVISION must also be incremented when dependencies have ABI changes.

19.1.12. Substituting variable text in the package files (the SUBST framework)

When you want to replace the same text in multiple files or when the replacement text varies, patches alone cannot help. This is where the SUBST framework comes in. It provides an easy-to-use interface for replacing text in files. Example:

```
SUBST_CLASSES+=                 fix-paths
SUBST_STAGE.fix-paths=          pre-configure
SUBST_MESSAGE.fix-paths=        Fixing absolute paths.
SUBST_FILES.fix-paths=          src/*.c
SUBST_FILES.fix-paths+=         scripts/*.sh
SUBST_SED.fix-paths=            -e 's,"/usr/local,"${PREFIX},g'
SUBST_SED.fix-paths+=           -e 's,"/var/log,"${VARBASE}/log,g'
```

SUBST_CLASSES is a list of identifiers that are used to identify the different SUBST blocks that are defined. The SUBST framework is heavily used by pkgsrc, so it is important to always use the += operator with this variable. Otherwise some substitutions may be skipped.

The remaining variables of each SUBST block are parameterized with the identifier from the first line (fix-paths in this case.) They can be seen as parameters to a function call.

SUBST_STAGE.* specifies the stage at which the replacement will take place. All combinations of pre-, do- and post- together with a phase name are possible, though only few are actually used. Most commonly used are post-patch and pre-configure. Of these two, pre-configure should be preferred because then it is possible to run **bmake patch** and have the state after applying the patches but before making any other changes. This is especially useful when you are debugging a package in order to create new patches for it. Similarly, post-build is preferred over pre-install, because the install phase should generally be kept as simple as possible. When you use post-build, you have the same files in the working directory that will be installed later, so you can check if the substitution has succeeded.

SUBST_MESSAGE.* is an optional text that is printed just before the substitution is done.

SUBST_FILES.* is the list of shell globbing patterns that specifies the files in which the substitution will take place. The patterns are interpreted relatively to the WRKSRC directory.

SUBST_SED.* is a list of arguments to sed(1) that specify the actual substitution. Every sed command should be prefixed with -e, so that all SUBST blocks look uniform.

There are some more variables, but they are so seldomly used that they are only documented in the mk/subst.mk file.

19.2. Fixing problems in the *fetch* phase

19.2.1. Packages whose distfiles aren't available for plain downloading

If you need to download from a dynamic URL you can set `DYNAMIC_MASTER_SITES` and a **make fetch** will call `files/getsite.sh` with the name of each file to download as an argument, expecting it to output the URL of the directory from which to download it. `graphics/ns-cult3d` is an example of this usage.

If the download can't be automated, because the user must submit personal information to apply for a password, or must pay for the source, or whatever, you can set `FETCH_MESSAGE` to a list of lines that are displayed to the user before aborting the build. Example:

```
FETCH_MESSAGE=  "Please download the files"
FETCH_MESSAGE+= "    "${DISTFILES:Q}
FETCH_MESSAGE+= "manually from "${MASTER_SITES:Q}"."
```

19.2.2. 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 checksum will no longer match. The contents of the new distfile should be compared against the old one before changing anything, to make sure the distfile was really updated on purpose, and that no trojan horse or so crept in. Please mention that the distfiles were compared and what was found in your commit message.

Then, the correct way to work around this is to set `DIST_SUBDIR` to a unique directory name, usually based on `PKGNAME_NOREV`. All `DISTFILES` and `PATCHFILES` for this package will be put in that subdirectory of the local distfiles directory. (See Section 19.1.11 for more details.) In case this happens more often, `PKGNAME` can be used (thus including the `nbX` suffix) or a date stamp can be appended, like `${PKGNAME_NOREV}-YYYYMMDD`.

`DIST_SUBDIR` is also used when a distfile's name does not contain a version and the distfile is apt to change. In cases where the likelihood of this is very small, `DIST_SUBDIR` might not be required. Additionally, `DIST_SUBDIR` must not be removed unless the distfile name changes, even if a package is being moved or renamed.

Do not forget regenerating the `distinfo` file after that, since it contains the `DIST_SUBDIR` path in the filenames. Also, increase the PKGREVISION if the installed package is different. Furthermore, a mail to the package's authors seems appropriate telling them that changing distfiles after releases without changing the file names is not good practice.

19.2.3. Packages hosted on github.com

Helper methods exist for packages hosted on github.com which will often have distfile names that clash with other packages, for example `1.0.tar.gz`. Use one of the three recipes from below:

19.2.3.1. Fetch based on a tagged release

If your distfile URL looks similar to `http://github.com/username/exampleproject/archive/v1.0.zip`, then you are packaging a tagged release.

```
DISTNAME=       exampleproject-1.0
MASTER_SITES=   ${MASTER_SITE_GITHUB:=username/}
#GITHUB_PROJECT=        # can be omitted if same as DISTNAME
GITHUB_TAG=     v${PKGVERSION_NOREV}
EXTRACT_SUFX=   .zip
```

19.2.3.2. Fetch based on a specific commit

If your distfile URL looks similar to `http://github.com/example/example/archive/988881adc9fc3655077dc2d4d757d480b5ea0e11.tar.gz`, then you are packaging a specific commit not tied to a release.

```
DISTNAME=       example-1.0
MASTER_SITES=   ${MASTER_SITE_GITHUB:example/}
#GITHUB_PROJECT=        # can be omitted if same as DISTNAME
GITHUB_TAG=     988881adc9fc3655077dc2d4d757d480b5ea0e11
```

19.2.3.3. Fetch based on release

If your distfile URL looks similar to `http://github.com/username/exampleproject/releases/download/rel-1.6/offensive-1.6.zip`, then you are packaging a release.

```
DISTNAME=       offensive-1.6
PKGNAME=        ${DISTNAME:S/offensive/proper/}
MASTER_SITES=   ${MASTER_SITE_GITHUB:=username/}
GITHUB_PROJECT= exampleproject
GITHUB_RELEASE= rel-${PKGVERSION_NOREV} # usually just set this to ${DISTNAME}
EXTRACT_SUFX=   .zip
```

19.3. Fixing problems in the *configure* phase

19.3.1. Shared libraries - libtool

pkgsrc 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 `devel/libtool`

pkg can help here, as it just "knows" how to build both static and dynamic libraries from a set of source files, thus being platform-independent.

Here's how to use libtool in a package in seven 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 instead use:

   ```
   ${LIBTOOL} --mode=link \
       ${CC} -o ${.TARGET:.a=.la} \
           ${OBJS:.o=.lo} \
           -rpath ${PREFIX}/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 `OBJS` as necessary. This automatically creates all of the `.a`, `.so.major.minor`, and ELF symlinks (if necessary) in the build directory. Be sure to include "-version-info", especially when major and minor are zero, as libtool will otherwise strip off the shared library version.

   From the libtool manual:

   ```
   So, libtool library versions are described by three integers:

   CURRENT
   The most recent interface number that this library implements.

   REVISION
   The implementation number of the CURRENT interface.

   AGE
   The difference between the newest and oldest interfaces that
   this library implements.  In other words, the library implements
   all the interface numbers in the range from number 'CURRENT -
   AGE' to 'CURRENT'.

   If two libraries have identical CURRENT and AGE numbers, then the
   dynamic linker chooses the library with the greater REVISION number.
   ```

   The "-release" option will produce different results for a.out and ELF (excluding symlinks) in only one case. An ELF library of the form "libfoo-release.so.*x.y*" will have a symlink of "libfoo.so.*x.y*" on an a.out platform. This is handled automatically.

   The "-rpath argument" is the install directory of the library being built.

   In the `PLIST`, include only the `.la` file, the other files will be added automatically.
4. When linking shared object (`.so`) files, i.e. files that are loaded via dlopen(3), NOT shared libraries, use "-module -avoid-version" to prevent them getting version tacked on.

The `PLIST` file gets the `foo.so` entry.

5. When linking programs that depend on these libraries *before* they are installed, preface the cc(1) or ld(1) 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 expects you to change that argument to be the `.la` file. e.g.

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

   should be changed to:

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

   and it will do the right thing with the libraries.

6. When installing libraries, preface the install(1) or cp(1) command with "${LIBTOOL} --mode=install", and change the library name to `.la`. e.g.

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

   This will install the static `.a`, shared library, any needed symlinks, and run ldconfig(8).

7. In your `PLIST`, include only the `.la` file (this is a change from previous behaviour).

19.3.2. Using libtool on GNU packages that already support libtool

Add `USE_LIBTOOL=yes` to the package Makefile. This will override the package's own libtool in most cases. For older libtool using packages, libtool is made by ltconfig script during the do-configure step; you can check the libtool script location by doing **make configure; find work*/ -name libtool**.

`LIBTOOL_OVERRIDE` specifies which libtool scripts, relative to `WRKSRC`, to override. By default, it is set to "libtool */libtool */*/libtool". If this does not match the location of the package's libtool script(s), set it as appropriate.

If you do not need `*.a` static libraries built and installed, then use `SHLIBTOOL_OVERRIDE` instead.

If your package makes use of the platform-independent library for loading dynamic shared objects, that comes with libtool (libltdl), you should include devel/libltdl/buildlink3.mk.

Some packages use libtool incorrectly so that the package may not work or build in some circumstances. Some of the more common errors are:

- The inclusion of a shared object (-module) as a dependent library in an executable or library. This in itself isn't a problem if one of two things has been done:
  1. The shared object is named correctly, i.e. `libfoo.la`, not `foo.la`
  2. The -dlopen option is used when linking an executable.

- The use of libltdl without the correct calls to initialisation routines. The function lt_dlinit() should be called and the macro `LTDL_SET_PRELOADED_SYMBOLS` included in executables.

19.3.3. GNU Autoconf/Automake

If a package needs GNU autoconf or automake to be executed to regenerate the configure script and Makefile.in makefile templates, then they should be executed in a pre-configure target.

For packages that need only autoconf:

```
AUTOCONF_REQD=  2.50            # if default version is not good enough
USE_TOOLS+=     autoconf        # use "autoconf213" for autoconf-2.13
...

pre-configure:
        cd ${WRKSRC} && autoconf

...
```

and for packages that need automake and autoconf:

```
AUTOMAKE_REQD=  1.7.1           # if default version is not good enough
USE_TOOLS+=     automake        # use "automake14" for automake-1.4
...

pre-configure:
        set -e; cd ${WRKSRC}; \
        aclocal; autoheader; automake -a --foreign -i; autoconf

...
```

Packages which use GNU Automake will almost certainly require GNU Make.

There are times when the configure process makes additional changes to the generated files, which then causes the build process to try to re-execute the automake sequence. This is prevented by touching various files in the configure stage. If this causes problems with your package you can set `AUTOMAKE_OVERRIDE=NO` in the package Makefile.

19.4. Programming languages

19.4.1. C, C++, and Fortran

Compilers for the C, C++, and Fortran languages comes with the NetBSD base system. By default, pkgsrc assumes that a package is written in C and will hide all other compilers (via the wrapper framework, see Chapter 14).

To declare which language's compiler a package needs, set the `USE_LANGUAGES` variable. Allowed values currently are "c", "c++", and "fortran" (and any combination). The default is "c". Packages using GNU configure scripts, even if written in C++, usually need a C compiler for the configure phase.

19.4.2. Java

If a program is written in Java, use the Java framework in pkgsrc. The package must include `../../mk/java-vm.mk`. This Makefile fragment provides the following variables:

- `USE_JAVA` defines if a build dependency on the JDK is added. If `USE_JAVA` is set to "run", then there is only a runtime dependency on the JDK. The default is "yes", which also adds a build dependency on the JDK.
- Set `USE_JAVA2` to declare that a package needs a Java2 implementation. The supported values are "yes", "1.4", and "1.5". "yes" accepts any Java2 implementation, "1.4" insists on versions 1.4 or above, and "1.5" only accepts versions 1.5 or above. This variable is not set by default.
- `PKG_JAVA_HOME` is automatically set to the runtime location of the used Java implementation dependency. It may be used to set `JAVA_HOME` to a good value if the program needs this variable to be defined.

19.4.3. Packages containing perl scripts

If your package contains interpreted perl scripts, add "perl" to the `USE_TOOLS` variable and set `REPLACE_PERL` to ensure that the proper interpreter path is set. `REPLACE_PERL` should contain a list of scripts, relative to `WRKSRC`, that you want adjusted. Every occurrence of `*/bin/perl` in a she-bang line will be replaced with the full path to the perl executable.

If a particular version of perl is needed, set the `PERL5_REQD` variable to the version number. The default is "5.0".

See Section 19.6.6 for information about handling perl modules.

19.4.4. Packages containing shell scripts

`REPLACE_SH`, `REPLACE_BASH`, `REPLACE_CSH`, and `REPLACE_KSH` can be used to replace shell hash bangs in files. Please use the appropriate one, prefering `REPLACE_SH` in case this shell is sufficient. Each should contain a list of scripts, relative to `WRKSRC`, that you want adjusted. Every occurrence of the matching shell in a she-bang line will be replaced with the full path to the shell executable. When using `REPLACE_BASH`, don't forget to add `bash` to `USE_TOOLS`.

19.4.5. Other programming languages

Currently, there is no special handling for other languages in pkgsrc. If a compiler package provides a `buildlink3.mk` file, include that, otherwise just add a (build) dependency on the appropriate compiler package.

19.5. Fixing problems in the *build* phase

The most common failures when building a package are that some platforms do not provide certain header files, functions or libraries, or they provide the functions in a library that the original package

author didn't know. To work around this, you can rewrite the source code in most cases so that it does not use the missing functions or provides a replacement function.

19.5.1. Compiling C and C++ code conditionally

If a package already comes with a GNU configure script, the preferred way to fix the build failure is to change the configure script, not the code. In the other cases, you can utilize the C preprocessor, which defines certain macros depending on the operating system and hardware architecture it compiles for. These macros can be queried using for example `#if defined(__i386)`. Almost every operating system, hardware architecture and compiler has its own macro. For example, if the macros `__GNUC__`, `__i386__` and `__NetBSD__` are all defined, you know that you are using NetBSD on an i386 compatible CPU, and your compiler is GCC.

The list of the following macros for hardware and operating system depends on the compiler that is used. For example, if you want to conditionally compile code on Solaris, don't use `__sun__`, as the SunPro compiler does not define it. Use `__sun` instead.

19.5.1.1. C preprocessor macros to identify the operating system

To distinguish between 4.4 BSD-derived systems and the rest of the world, you should use the following code.

```
#include <sys/param.h>
#if (defined(BSD) && BSD >= 199306)
/* BSD-specific code goes here */
#else
/* non-BSD-specific code goes here */
#endif
```

If this distinction is not fine enough, you can also test for the following macros.

```
Cygwin      __CYGWIN__
DragonFly   __DragonFly__
FreeBSD     __FreeBSD__
Haiku       __HAIKU__
Interix     __INTERIX
IRIX        __sgi (TODO: get a definite source for this)
Linux       linux, __linux, __linux__
Mac OS X    __APPLE__
MirBSD      __MirBSD__ (__OpenBSD__ is also defined)
Minix3      __minix
NetBSD      __NetBSD__
OpenBSD     __OpenBSD__
Solaris     sun, __sun
```

19.5.1.2. C preprocessor macros to identify the hardware architecture

```
i386        i386, __i386, __i386__
MIPS        __mips
SPARC       sparc, __sparc
```

19.5.1.3. C preprocessor macros to identify the compiler

```
GCC         __GNUC__ (major version), __GNUC_MINOR__
MIPSpro     _COMPILER_VERSION (0x741 for MIPSpro 7.41)
SunPro      __SUNPRO_C (0x570 for Sun C 5.7)
SunPro C++  __SUNPRO_CC (0x580 for Sun C++ 5.8)
```

19.5.2. How to handle compiler bugs

Some source files trigger bugs in the compiler, based on combinations of compiler version and architecture and almost always relation to optimisation being enabled. Common symptoms are gcc internal errors or never finishing compiling a file.

Typically, a workaround involves testing the `MACHINE_ARCH` and compiler version, disabling optimisation for that combination of file, `MACHINE_ARCH` and compiler.

This used to be a big problem in the past, but is rarely needed now as compiler technology has matured. If you still need to add a compiler specific workaround, please do so in the file `hacks.mk` and describe the symptom and compiler version as detailed as possible.

19.5.3. Undefined reference to "..."

This error message often means that a package did not link to a shared library it needs. The following functions are known to cause this error message over and over.

Function	Library	Affected platforms
accept, bind, connect	-lsocket	Solaris
crypt	-lcrypt	DragonFly, NetBSD
dlopen, dlsym	-ldl	Linux
gethost*	-lnsl	Solaris
inet_aton	-lresolv	Solaris
nanosleep, sem_*, timer_*	-lrt	Solaris
openpty	-lutil	Linux

To fix these linker errors, it is often sufficient to say `LIBS.`*`OperatingSystem`*`+= -l`*`foo`* to the package `Makefile` and then say **bmake clean; bmake**.

19.5.3.1. Special issue: The SunPro compiler

When you are using the SunPro compiler, there is another possibility. That compiler cannot handle the following code:

```
extern int extern_func(int);

static inline int
inline_func(int x)
{
```

```
        return extern_func(x);
}

int main(void)
{
        return 0;
}
```

It generates the code for `inline_func` even if that function is never used. This code then refers to `extern_func`, which can usually not be resolved. To solve this problem you can try to tell the package to disable inlining of functions.

19.5.4. Running out of memory

Sometimes packages fail to build because the compiler runs into an operating system specific soft limit. With the `UNLIMIT_RESOURCES` variable pkgsrc can be told to unlimit the resources. Currently, the allowed values are "datasize" and "stacksize" (or both). Setting this variable is similar to running the shell builtin **ulimit** command to raise the maximum data segment size or maximum stack size of a process, respectively, to their hard limits.

19.6. Fixing problems in the *install* phase

19.6.1. Creating needed directories

The BSD-compatible **install** supplied with some operating systems cannot create more than one directory at a time. As such, you should call `${INSTALL_*_DIR}` like this:

```
${INSTALL_DATA_DIR} ${PREFIX}/dir1
${INSTALL_DATA_DIR} ${PREFIX}/dir2
```

You can also just append "`dir1 dir2`" to the `INSTALLATION_DIRS` variable, which will automatically do the right thing.

19.6.2. Where to install documentation

In general, documentation should be installed into `${PREFIX}/share/doc/${PKGBASE}` or `${PREFIX}/share/doc/${PKGNAME}` (the latter includes the version number of the package).

Many modern packages using GNU autoconf allow to set the directory where HTML documentation is installed with the "--with-html-dir" option. Sometimes using this flag is needed because otherwise the documentation ends up in `${PREFIX}/share/doc/html` or other places.

An exception to the above is that library API documentation generated with the `textproc/gtk-doc` tools, for use by special browsers (devhelp) should be left at their default location, which is `${PREFIX}/share/gtk-doc`. Such documentation can be recognized from files ending in `.devhelp`

or `.devhelp2`. (It is also acceptable to install such files in `${PREFIX}/share/doc/${PKGBASE}` or `${PREFIX}/share/doc/${PKGNAME}`; the `.devhelp*` file must be directly in that directory then, no additional subdirectory level is allowed in this case. This is usually achieved by using "--with-html-dir=${PREFIX}/share/doc". `${PREFIX}/share/gtk-doc` is preferred though.)

19.6.3. Installing highscore files

Certain packages, most of them in the games category, install a score file that allows all users on the system to record their highscores. In order for this to work, the binaries need to be installed setgid and the score files owned by the appropriate group and/or owner (traditionally the "games" user/group). Set `USE_GAMESGROUP` to yes to support this. The following variables, documented in more detail in `mk/defaults/mk.conf`, control this behaviour: `GAMEDATAMODE`, `GAMEDIRMODE`, `GAMES_GROUP`, `GAMEMODE`, `GAME_USER`.

A package should therefore never hard code file ownership or access permissions but rely on `INSTALL_GAME` and `INSTALL_GAME_DATA` to set these correctly.

19.6.4. Adding DESTDIR support to packages

`DESTDIR` support means that a package installs into a staging directory, not the final location of the files. Then a binary package is created which can be used for installation as usual. There are two ways: Either the package must install as root ("destdir") or the package can install as non-root user ("user-destdir").

- `PKG_DESTDIR_SUPPORT` has to be set to "none", "destdir", or "user-destdir". By default `PKG_DESTDIR_SUPPORT` is set to "user-destdir" to help catching more potential packaging problems. If bsd.prefs.mk is included in the Makefile, `PKG_DESTDIR_SUPPORT` needs to be set before the inclusion.
- All installation operations have to be prefixed with `${DESTDIR}`.
- automake gets this DESTDIR mostly right automatically. Many manual rules and pre/post-install often are incorrect; fix them.
- If files are installed with special owner/group use `SPECIAL_PERMS`.
- In general, packages should support `UNPRIVILEGED` to be able to use DESTDIR.

19.6.5. Packages with hardcoded paths to other interpreters

Your package may also contain scripts with hardcoded paths to other interpreters besides (or as well as) perl. To correct the full pathname to the script interpreter, you need to set the following definitions in your `Makefile` (we shall use **tclsh** in this example):

```
REPLACE_INTERPRETER+=   tcl
REPLACE.tcl.old=        .*/bin/tclsh
REPLACE.tcl.new=        ${PREFIX}/bin/tclsh
REPLACE_FILES.tcl=      # list of tcl scripts which need to be fixed,
# relative to ${WRKSRC}, just as in REPLACE_PERL
```

Note: Before March 2006, these variables were called `_REPLACE.*` and `_REPLACE_FILES.*`.

19.6.6. Packages installing perl modules

Makefiles of packages providing perl5 modules should include the Makefile fragment `../../lang/perl5/module.mk`. It provides a **do-configure** target for the standard perl configuration for such modules as well as various hooks to tune this configuration. See comments in this file for details.

Perl5 modules will install into different places depending on the version of perl used during the build process. To address this, pkgsrc will append lines to the `PLIST` corresponding to the files listed in the installed `.packlist` file generated by most perl5 modules. This is invoked by defining `PERL5_PACKLIST` to a space-separated list of packlist files relative to `PERL5_PACKLIST_DIR` (`PERL5_INSTALLVENDORARCH` by default), e.g.:

```
PERL5_PACKLIST= auto/Pg/.packlist
```

The perl5 config variables `installarchlib`, `installscript`, `installvendorbin`, `installvendorscript`, `installvendorarch`, `installvendorlib`, `installvendorman1dir`, and `installvendorman3dir` represent those locations in which components of perl5 modules may be installed, provided as variable with uppercase and prefixed with `PERL5_`, e.g. `PERL5_INSTALLARCHLIB` and may be used by perl5 packages that don't have a packlist. These variables are also substituted for in the `PLIST` as uppercase prefixed with `PERL5_SUB_`.

19.6.7. Packages installing info files

Some packages install info files or use the "makeinfo" or "install-info" commands. `INFO_FILES` should be defined in the package Makefile so that `INSTALL` and `DEINSTALL` scripts will be generated to handle registration of the info files in the Info directory file. The "install-info" command used for the info files registration is either provided by the system, or by a special purpose package automatically added as dependency if needed.

`PKGINFODIR` is the directory under `${PREFIX}` where info files are primarily located. `PKGINFODIR` defaults to "info" and can be overridden by the user.

The info files for the package should be listed in the package `PLIST`; however any split info files need not be listed.

A package which needs the "makeinfo" command at build time must add "makeinfo" to `USE_TOOLS` in its Makefile. If a minimum version of the "makeinfo" command is needed it should be noted with the `TEXINFO_REQD` variable in the package `Makefile`. By default, a minimum version of 3.12 is required. If the system does not provide a **makeinfo** command or if it does not match the required minimum, a build dependency on the `devel/gtexinfo` package will be added automatically.

The build and installation process of the software provided by the package should not use the **install-info** command as the registration of info files is the task of the package `INSTALL` script, and it must use the appropriate **makeinfo** command.

To achieve this goal, the pkgsrc infrastructure creates overriding scripts for the **install-info** and **makeinfo** commands in a directory listed early in `PATH`.

The script overriding **install-info** has no effect except the logging of a message. The script overriding **makeinfo** logs a message and according to the value of `TEXINFO_REQD` either runs the appropriate **makeinfo** command or exit on error.

19.6.8. Packages installing man pages

All packages that install manual pages should install them into the same directory, so that there is one common place to look for them. In pkgsrc, this place is `${PREFIX}/${PKGMANDIR}`, and this expression should be used in packages. The default for `PKGMANDIR` is "`man`". Another often-used value is "`share/man`".

> **Note:** The support for a custom `PKGMANDIR` is far from complete.

The `PLIST` files can just use `man/` as the top level directory for the man page file entries, and the pkgsrc framework will convert as needed. In all other places, the correct `PKGMANDIR` must be used.

Packages that are configured with `GNU_CONFIGURE` set as "yes", by default will use the `./configure` --mandir switch to set where the man pages should be installed. The path is `GNU_CONFIGURE_MANDIR` which defaults to `${PREFIX}/${PKGMANDIR}`.

Packages that use `GNU_CONFIGURE` but do not use --mandir, can set `CONFIGURE_HAS_MANDIR` to "no". Or if the `./configure` script uses a non-standard use of --mandir, you can set `GNU_CONFIGURE_MANDIR` as needed.

See Section 13.5 for information on installation of compressed manual pages.

19.6.9. Packages installing GConf data files

If a package installs `.schemas` or `.entries` files, used by GConf, you need to take some extra steps to make sure they get registered in the database:

1. Include `../../devel/GConf/schemas.mk` instead of its `buildlink3.mk` file. This takes care of rebuilding the GConf database at installation and deinstallation time, and tells the package where to install GConf data files using some standard configure arguments. It also disallows any access to the database directly from the package.
2. Ensure that the package installs its `.schemas` files under `${PREFIX}/share/gconf/schemas`. If they get installed under `${PREFIX}/etc`, you will need to manually patch the package.
3. Check the PLIST and remove any entries under the etc/gconf directory, as they will be handled automatically. See Section 9.13 for more information.
4. Define the `GCONF_SCHEMAS` variable in your `Makefile` with a list of all `.schemas` files installed by the package, if any. Names must not contain any directories in them.
5. Define the `GCONF_ENTRIES` variable in your `Makefile` with a list of all `.entries` files installed by the package, if any. Names must not contain any directories in them.

19.6.10. Packages installing scrollkeeper/rarian data files

If a package installs `.omf` files, used by scrollkeeper/rarian, you need to take some extra steps to make sure they get registered in the database:

1. Include `../../mk/omf-scrollkeeper.mk` instead of rarian's `buildlink3.mk` file. This takes care of rebuilding the scrollkeeper database at installation and deinstallation time, and disallows any access to it directly from the package.
2. Check the PLIST and remove any entries under the `libdata/scrollkeeper` directory, as they will be handled automatically.
3. Remove the `share/omf` directory from the PLIST. It will be handled by rarian. (**make print-PLIST** does this automatically.)

19.6.11. Packages installing X11 fonts

If a package installs font files, you will need to rebuild the fonts database in the directory where they get installed at installation and deinstallation time. This can be automatically done by using the pkginstall framework.

You can list the directories where fonts are installed in the `FONTS_DIRS.`*`type`* variables, where *`type`* can be one of "ttf", "type1" or "x11". Also make sure that the database file `fonts.dir` is not listed in the PLIST.

Note that you should not create new directories for fonts; instead use the standard ones to avoid that the user needs to manually configure his X server to find them.

19.6.12. Packages installing GTK2 modules

If a package installs GTK2 immodules or loaders, you need to take some extra steps to get them registered in the GTK2 database properly:

1. Include `../../x11/gtk2/modules.mk` instead of its `buildlink3.mk` file. This takes care of rebuilding the database at installation and deinstallation time.
2. Set `GTK2_IMMODULES=YES` if your package installs GTK2 immodules.
3. Set `GTK2_LOADERS=YES` if your package installs GTK2 loaders.
4. Patch the package to not touch any of the GTK2 databases directly. These are:
   - `libdata/gtk-2.0/gdk-pixbuf.loaders`
   - `libdata/gtk-2.0/gtk.immodules`
5. Check the `PLIST` and remove any entries under the `libdata/gtk-2.0` directory, as they will be handled automatically.

19.6.13. Packages installing SGML or XML data

If a package installs SGML or XML data files that need to be registered in system-wide catalogs (like DTDs, sub-catalogs, etc.), you need to take some extra steps:

1. Include `../../textproc/xmlcatmgr/catalogs.mk` in your `Makefile`, which takes care of registering those files in system-wide catalogs at installation and deinstallation time.
2. Set `SGML_CATALOGS` to the full path of any SGML catalogs installed by the package.
3. Set `XML_CATALOGS` to the full path of any XML catalogs installed by the package.
4. Set `SGML_ENTRIES` to individual entries to be added to the SGML catalog. These come in groups of three strings; see xmlcatmgr(1) for more information (specifically, arguments recognized by the 'add' action). Note that you will normally not use this variable.
5. Set `XML_ENTRIES` to individual entries to be added to the XML catalog. These come in groups of three strings; see xmlcatmgr(1) for more information (specifically, arguments recognized by the 'add' action). Note that you will normally not use this variable.

19.6.14. Packages installing extensions to the MIME database

If a package provides extensions to the MIME database by installing `.xml` files inside `${PREFIX}/share/mime/packages`, you need to take some extra steps to ensure that the database is kept consistent with respect to these new files:

1. Include `../../databases/shared-mime-info/mimedb.mk` (avoid using the `buildlink3.mk` file from this same directory, which is reserved for inclusion from other `buildlink3.mk` files). It takes care of rebuilding the MIME database at installation and deinstallation time, and disallows any access to it directly from the package.
2. Check the PLIST and remove any entries under the `share/mime` directory, *except* for files saved under `share/mime/packages`. The former are handled automatically by the update-mime-database program, but the latter are package-dependent and must be removed by the package that installed them in the first place.
3. Remove any `share/mime/*` directories from the PLIST. They will be handled by the shared-mime-info package.

19.6.15. Packages using intltool

If a package uses intltool during its build, add `intltool` to the `USE_TOOLS`, which forces it to use the intltool package provided by pkgsrc, instead of the one bundled with the distribution file.

This tracks intltool's build-time dependencies and uses the latest available version; this way, the package benefits of any bug fixes that may have appeared since it was released.

19.6.16. Packages installing startup scripts

If a package contains a rc.d script, it won't be copied into the startup directory by default, but you can enable it, by adding the option `PKG_RCD_SCRIPTS=YES` in `mk.conf`. This option will copy the scripts

into /etc/rc.d when a package is installed, and it will automatically remove the scripts when the package is deinstalled.

19.6.17. Packages installing TeX modules

If a package installs TeX packages into the texmf tree, the ls-R database of the tree needs to be updated.

> **Note:** Except the main TeX packages such as kpathsea, packages should install files into ${PREFIX}/share/texmf-dist, not ${PREFIX}/share/texmf.

1. Include ../../print/kpathsea/texmf.mk. This takes care of rebuilding the ls-R database at installation and deinstallation time.
2. If your package installs files into a texmf tree other than the one at ${PREFIX}/share/texmf-dist, set TEX_TEXMF_DIRS to the list of all texmf trees that need database update.

   If your package also installs font map files that need to be registered using **updmap**, include ../../print/tex-tetex/map.mk and set TEX_MAP_FILES and/or TEX_MIXEDMAP_FILES to the list of all such font map files. Then **updmap** will be run automatically at installation/deinstallation to enable/disable font map files for TeX output drivers.
3. Make sure that none of ls-R databases are included in PLIST, as they will be removed only by the kpathsea package.

19.6.18. Packages supporting running binaries in emulation

There are some packages that provide libraries and executables for running binaries from a one operating system on a different one (if the latter supports it). One example is running Linux binaries on NetBSD.

The pkgtools/rpm2pkg helps in extracting and packaging Linux rpm packages.

The CHECK_SHLIBS can be set to no to avoid the **check-shlibs** target, which tests if all libraries for each installed executable can be found by the dynamic linker. Since the standard dynamic linker is run, this fails for emulation packages, because the libraries used by the emulation are not in the standard directories.

19.6.19. Packages installing hicolor theme icons

If a package installs images under the share/icons/hicolor and/or updates the share/icons/hicolor/icon-theme.cache database, you need to take some extra steps to make sure that the shared theme directory is handled appropriately and that the cache database is rebuilt:

1. Include ../../graphics/hicolor-icon-theme/buildlink3.mk.
2. Check the PLIST and remove the entry that refers to the theme cache.

3. Ensure that the PLIST does not remove the shared icon directories from the `share/icons/hicolor` hierarchy because they will be handled automatically.

The best way to verify that the PLIST is correct with respect to the last two points is to regenerate it using **make print-PLIST**.

19.6.20. Packages installing desktop files

If a package installs `.desktop` files under `share/applications` and these include MIME information (MimeType key), you need to take extra steps to ensure that they are registered into the MIME database:

1. Include `../../sysutils/desktop-file-utils/desktopdb.mk`.
2. Check the PLIST and remove the entry that refers to the `share/applications/mimeinfo.cache` file. It will be handled automatically.

The best way to verify that the PLIST is correct with respect to the last point is to regenerate it using **make print-PLIST**.

19.7. Marking packages as having problems

In some cases one does not have the time to solve a problem immediately. In this case, one can plainly mark a package as broken. For this, one just sets the variable `BROKEN` to the reason why the package is broken (similar to the `RESTRICTED` variable). A user trying to build the package will immediately be shown this message, and the build will not be even tried.

`BROKEN` packages are removed from pkgsrc in irregular intervals.

Chapter 20.
Debugging

To check out all the gotchas when building a package, 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.

- Be sure to set `PKG_DEVELOPER=yes` in `mk.conf`.
- Install `pkgtools/url2pkg`, create a directory for a new package, change into it, then run **url2pkg**:

```
% mkdir /usr/pkgsrc/category/examplepkg
% cd /usr/pkgsrc/category/examplepkg
% url2pkg http://www.example.com/path/to/distfile.tar.gz
```

- Edit the `Makefile` as requested.
- Fill in the `DESCR` file
- Run **make configure**
- Add any dependencies glimpsed from documentation and the configure step to the package's `Makefile`.
- Make the package compile, doing multiple rounds of

```
% make
% pkgvi ${WRKSRC}/some/file/that/does/not/compile
% mkpatches
% patchdiff
% mv ${WRKDIR}/.newpatches/* patches
% make mps
% make clean
```

  Doing this step as non-root user will ensure that no files are modified that shouldn't be, especially during the build phase. **mkpatches**, **patchdiff** and **pkgvi** are from the `pkgtools/pkgdiff` package.

- Look at the `Makefile`, fix if necessary; see Section 11.1.
- Generate a `PLIST`:

```
# make install
# make print-PLIST >PLIST
# make deinstall
# make install
# make deinstall
```

  You usually need to be `root` to do this. Look if there are any files left:

```
# make print-PLIST
```

  If this reveals any files that are missing in `PLIST`, add them.

- Now that the `PLIST` is OK, install the package again and make a binary package:

```
# make reinstall
# make package
```

- Delete the installed package:

```
# pkg_delete examplepkg
```

- Repeat the above **make print-PLIST** command, which shouldn't find anything now:

```
# make print-PLIST
```

- Reinstall the binary package:

```
# pkg_add .../examplepkg.tgz
```

- Play with it. Make sure everything works.
- Run **pkglint** from `pkgtools/pkglint`, and fix the problems it reports:

```
# pkglint
```

- Submit (or commit, if you have cvs access); see Chapter 21.

Chapter 21.
Submitting and Committing

21.1. Submitting binary packages

Our policy is that we accept binaries only from pkgsrc 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. NetBSD developers doing bulk builds and wanting to upload them please see Chapter 7.

21.2. Submitting source packages (for non-NetBSD-developers)

First, check that your package is complete, compiles and runs well; see Chapter 20 and the rest of this document. Next, generate an uuencoded gzipped tar(1) archive that contains all files that make up the package. Finally, send this package to the pkgsrc bug tracking system, either with the send-pr(1) command, or if you don't have that, go to the web page http://www.NetBSD.org/support/send-pr.html, which contains some instructions and a link to a form where you can submit packages. The `sysutils/gtk-send-pr` package is also available as a substitute for either of the above two tools.

In the form of the problem report, the category should be "pkg", the synopsis should include the package name and version number, and the description field should contain a short description of your package (contents of the COMMENT variable or DESCR file are OK). The uuencoded package data should go into the "fix" field.

If you want to submit several packages, please send a separate PR for each one, it's easier for us to track things that way.

Alternatively, you can also import new packages into pkgsrc-wip ("pkgsrc work-in-progress"); see the homepage at http://pkgsrc-wip.sourceforge.net/ for details.

21.3. General notes when adding, updating, or removing packages

Please note all package additions, updates, moves, and removals in `pkgsrc/doc/CHANGES-`*`YYYY`*. 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/) and other sites. Additionally, check the `pkgsrc/doc/TODO` file and remove the entry for the package you updated or removed, in case it was mentioned there.

When the `PKGREVISION` of a package is bumped, the change should appear in `pkgsrc/doc/CHANGES-`*`YYYY`* if it is security related or otherwise relevant. Mass bumps that result from a dependency being updated should not be mentioned. In all other cases it's the developer's decision.

There is a make target that helps in creating proper `CHANGES-YYYY` entries: **make changes-entry**. It uses the optional `CTYPE` and `NETBSD_LOGIN_NAME` variables. The general usage is to first make sure that your `CHANGES-YYYY` file is up-to-date (to avoid having to resolve conflicts later-on) and then to **cd** to the package directory. For package updates, **make changes-entry** is enough. For new packages, or package moves or removals, set the `CTYPE` variable on the command line to "Added", "Moved", or "Removed". You can set `NETBSD_LOGIN_NAME` in `mk.conf` if your local login name is not the same as your NetBSD login name. The target also automatically removes possibly existing entries for the package in the `TODO` file. Don't forget to commit the changes, e.g. by using **make commit-changes-entry**! If you are not using a checkout directly from cvs.NetBSD.org, but e.g. a local copy of the repository, you can set USE_NETBSD_REPO=yes. This makes the cvs commands use the main repository.

21.4. Committing: Adding a package to CVS

This section is only of interest for pkgsrc developers with write access to the pkgsrc repository.

When the package is finished, "cvs add" the files. Start by adding the directory and then files in the directory. Don't forget to add the new package to the category's `Makefile`. Make sure you don't forget any files; you can check by running "cvs status". An example:

```
$ cd .../pkgsrc/category
$ cvs add pkgname
$ cd pkgname
$ cvs add DESCR Makefile PLIST distinfo buildlink3.mk patches
$ cvs add patches/p*
$ cvs status | less
$ cvs commit
$ cd ..
$ vi Makefile # add SUBDIRS+=pkgname line
$ cvs commit Makefile
$ cd pkgname
$ make CTYPE=Added commit-changes-entry
```

The commit message of the initial import should include part of the `DESCR` file, so people reading the mailing lists know what the package is/does.

Also mention the new package in `pkgsrc/doc/CHANGES-20xx`.

Previously, "cvs import" was suggested, but it was much easier to get wrong than "cvs add".

21.5. Updating a package to a newer version

Please always put a concise, appropriate and relevant summary of the changes between old and new versions into the commit log when updating a package. There are various reasons for this:

- A URL is volatile, and can change over time. It may go away completely or its information may be overwritten by newer information.
- Having the change information between old and new versions in our CVS repository is very useful for people who use either cvs or anoncvs.

- Having the change information between old and new versions in our CVS repository is very useful for people who read the pkgsrc-changes mailing list, so that they can make tactical decisions about when to upgrade the package.

Please also recognize that, just because a new version of a package has been released, it should not automatically be upgraded in the CVS repository. We prefer to be conservative in the packages that are included in pkgsrc - development or beta packages are not really the best thing for most places in which pkgsrc is used. Please use your judgement about what should go into pkgsrc, and bear in mind that stability is to be preferred above new and possibly untested features.

21.6. Renaming a package in pkgsrc

Renaming packages is not recommended.

When renaming packages, be sure to fix any references to old name in other Makefiles, options, buildlink files, etc.

Also When renaming a package, please define `SUPERSEDES` to the package name and dewey version pattern(s) of the previous package name. This may be repeated for multiple renames. The new package would be an exact replacement.

Note that "successor" in the CHANGES-*YYYY* file doesn't necessarily mean that it *supersedes*, as that successor may not be an exact replacement but is a suggestion for the replaced functionality.

21.7. Moving a package in pkgsrc

It is preferred that packages are not renamed or moved, but if needed please follow these steps.

1. Make a copy of the directory somewhere else.
2. Remove all CVS dirs.

   Alternatively to the first two steps you can also do:

   ```
   % cvs -d user@cvs.NetBSD.org:/cvsroot export -D today pkgsrc/category/package
   ```

   and use that for further work.
3. Fix `CATEGORIES` and any `DEPENDS` paths that just did "../package" instead of "../../category/package".
4. In the modified package's Makefile, consider setting `PREV_PKGPATH` to the previous category/package pathname. The `PREV_PKGPATH` can be used by tools for doing an update using pkgsrc building; for example, it can search the pkg_summary(5) database for `PREV_PKGPATH` (if no `SUPERSEDES`) and then use the corresponding new `PKGPATH` for that moved package. Note that it may have multiple matches, so the tool should also check on the `PKGBASE` too. The `PREV_PKGPATH` probably has no value unless `SUPERSEDES` is not set, i.e. `PKGBASE` stays the same.
5. **cvs import** the modified package in the new place.
6. Check if any package depends on it:

   ```
   % cd /usr/pkgsrc
   % grep /package */*/Makefile* */*/buildlink*
   ```

7. Fix paths in packages from step 5 to point to new location.
8. **cvs rm (-f)** the package at the old location.
9. Remove from `oldcategory/Makefile`.
10. Add to `newcategory/Makefile`.
11. Commit the changed and removed files:

    ```
    % cvs commit oldcategory/package oldcategory/Makefile newcategory/Makefile
    ```

    (and any packages from step 5, of course).

Chapter 22.
Frequently Asked Questions

This section contains the answers to questions that may arise when you are writing a package. If you don't find your question answered here, first have a look in the other chapters, and if you still don't have the answer, ask on the `pkgsrc-users` mailing list.

1. What is the difference between `MAKEFLAGS`, `.MAKEFLAGS` and `MAKE_FLAGS`?

`MAKEFLAGS` are the flags passed to the pkgsrc-internal invocations of make(1), while `MAKE_FLAGS` are the flags that are passed to the `MAKE_PROGRAM` when building the package. [FIXME: What is .MAKEFLAGS for?]

2. What is the difference between `MAKE`, `GMAKE` and `MAKE_PROGRAM`?

`MAKE` is the path to the make(1) program that is used in the pkgsrc infrastructure. `GMAKE` is the path to GNU Make, but you need to say `USE_TOOLS+=gmake` to use that. `MAKE_PROGRAM` is the path to the Make program that is used for building the package.

3. What is the difference between `CC`, `PKG_CC` and `PKGSRC_COMPILER`?

`CC` is the path to the real C compiler, which can be configured by the pkgsrc user. `PKG_CC` is the path to the compiler wrapper. `PKGSRC_COMPILER` is *not* a path to a compiler, but the type of compiler that should be used. See `mk/compiler.mk` for more information about the latter variable.

4. What is the difference between `BUILDLINK_LDFLAGS`, `BUILDLINK_LDADD` and `BUILDLINK_LIBS`?

[FIXME]

5. Why does **make show-var VARNAME=BUILDLINK_PREFIX.*foo*** say it's empty?

For optimization reasons, some variables are only available in the "wrapper" phase and later. To "simulate" the wrapper phase, append **PKG_PHASE=wrapper** to the above command.

6. What does `${MASTER_SITE_SOURCEFORGE:=package/}` mean? I don't understand the `:=` inside it.

The `:=` is not really an assignment operator, like you might expect at first sight. Instead, it is a degenerate form of `${LIST:old_string=new_string}`, which is documented in the make(1) man page and which you may have seen as in `${SRCS:.c=.o}`. In the case of `MASTER_SITE_*`, *`old_string`* is the empty string and *`new_string`* is `package/`. That's where the `:` and the `=` fall together.

7. Which mailing lists are there for package developers?

tech-pkg (http://www.NetBSD.org/mailinglists/index.html#tech-pkg)

This is a list for technical discussions related to pkgsrc development, e.g. soliciting feedback for changes to pkgsrc infrastructure, proposed new features, questions related to porting pkgsrc to a

new platform, advice for maintaining a package, patches that affect many packages, help requests moved from pkgsrc-users when an infrastructure bug is found, etc.

pkgsrc-bugs (http://www.NetBSD.org/mailinglists/index.html#pkgsrc-bugs)

All bug reports in category "pkg" sent with send-pr(1) appear here. Please do not report your bugs here directly; use one of the other mailing lists.

8. Where is the pkgsrc documentation?

There are many places where you can find documentation about pkgsrc:

- The pkgsrc guide (this document) is a collection of chapters that explain large parts of pkgsrc, but some chapters tend to be outdated. Which ones they are is hard to say.
- On the mailing list archives (see http://mail-index.NetBSD.org/), you can find discussions about certain features, announcements of new parts of the pkgsrc infrastructure and sometimes even announcements that a certain feature has been marked as obsolete. The benefit here is that each message has a date appended to it.
- Many of the files in the `mk/` directory start with a comment that describes the purpose of the file and how it can be used by the pkgsrc user and package authors. An easy way to find this documentation is to run **bmake help**.
- The CVS log messages are a rich source of information, but they tend to be highly abbreviated, especially for actions that occur often. Some contain a detailed description of what has changed, but they are geared towards the other pkgsrc developers, not towards an average pkgsrc user. They also only document *changes*, so if you don't know what has been before, these messages may not be worth too much to you.
- Some parts of pkgsrc are only "implicitly documented", that is the documentation exists only in the mind of the developer who wrote the code. To get this information, use the **cvs annotate** command to see who has written it and ask on the `tech-pkg` mailing list, so that others can find your questions later (see above). To be sure that the developer in charge reads the mail, you may CC him or her.

9. I have a little time to kill. What shall I do?

This is not really an FAQ yet, but here's the answer anyway.

- Run **pkg_chk -N** (from the `pkgtools/pkg_chk` package). It will tell you about newer versions of installed packages that are available, but not yet updated in pkgsrc.
- Browse `pkgsrc/doc/TODO` — it contains a list of suggested new packages and a list of cleanups and enhancements for pkgsrc that would be nice to have.
- Review packages for which review was requested on the tech-pkg (http://www.NetBSD.org/mailinglists/index.html#tech-pkg) mailing list.

Chapter 23.
GNOME packaging and porting

Quoting GNOME's web site (http://www.gnome.org/):

> The GNOME project provides two things: The GNOME desktop environment, an intuitive and attractive desktop for users, and the GNOME development platform, an extensive framework for building applications that integrate into the rest of the desktop.

pkgsrc provides a seamless way to automatically build and install a complete GNOME environment *under many different platforms*. We can say with confidence that pkgsrc is one of the most advanced build and packaging systems for GNOME due to its included technologies buildlink3, the wrappers and tools framework and automatic configuration file management. Lots of efforts are put into achieving a completely clean deinstallation of installed software components.

Given that pkgsrc is NetBSD (http://www.NetBSD.org/)'s official packaging system, the above also means that great efforts are put into making GNOME work under this operating system. Recently, DragonFly BSD (http://www.dragonflybsd.org/) also adopted pkgsrc as its preferred packaging system, contributing lots of portability fixes to make GNOME build and install under it.

This chapter is aimed at pkgsrc developers and other people interested in helping our GNOME porting and packaging efforts. It provides instructions on how to manage the existing packages and some important information regarding their internals.

> **We need your help!:** Should you have some spare cycles to devote to NetBSD, pkgsrc and GNOME and are willing to learn new exciting stuff, please jump straight to the pending work (http://www.NetBSD.org/contrib/projects.html#gnome) list! There is still a long way to go to get a fully-functional GNOME desktop under NetBSD and we need your help to achieve it!

23.1. Meta packages

pkgsrc includes three GNOME-related meta packages:

- `meta-pkgs/gnome-base`: Provides the core GNOME desktop environment. It only includes the necessary bits to get it to boot correctly, although it may lack important functionality for daily operation. The idea behind this package is to let end users build their own configurations on top of this one, first installing this meta package to achieve a functional setup and then adding individual applications.
- `meta-pkgs/gnome`: Provides a complete installation of the GNOME platform and desktop as defined by the GNOME project; this is based on the components distributed in the `platform/x.y/x.y.z/sources` and `desktop/x.y/x.y.z/sources` directories of the official FTP server. Developer-only tools found in those directories are not installed unless required by some other component to work properly. Similarly, packages from the bindings set

(`bindings/x.y/x.y.z/sources`) are not pulled in unless required as a dependency for an end-user component. This package "extends" `meta-pkgs/gnome-base`.

- `meta-pkgs/gnome-devel`: Installs all the tools required to build a GNOME component when fetched from the CVS repository. These are required to let the **autogen.sh** scripts work appropriately.

In all these packages, the `DEPENDS` lines are sorted in a way that eases updates: a package may depend on other packages listed before it but not on any listed after it. It is very important to keep this order to ease updates so... *do not change it to alphabetical sorting!*

23.2. Packaging a GNOME application

Almost all GNOME applications are written in C and use a common set of tools as their build system. Things get different with the new bindings to other languages (such as Python), but the following will give you a general idea on the minimum required tools:

- Almost all GNOME applications use the GNU Autotools as their build system. As a general rule you will need to tell this to your package:

  ```
  GNU_CONFIGURE=yes
  USE_LIBTOOL=yes
  USE_TOOLS+=gmake
  ```

- If the package uses pkg-config to detect dependencies, add this tool to the list of required utilities:

  ```
  USE_TOOLS+=pkg-config
  ```

  Also use `pkgtools/verifypc` at the end of the build process to ensure that you did not miss to specify any dependency in your package and that the version requirements are all correct.

- If the package uses intltool, be sure to add `intltool` to the `USE_TOOLS` to handle dependencies and to force the package to use the latest available version.

- If the package uses gtk-doc (a documentation generation utility), do *not* add a dependency on it. The tool is rather big and the distfile should come with pregenerated documentation anyway; if it does not, it is a bug that you ought to report. For such packages you should disable gtk-doc (unless it is the default):

  ```
  CONFIGURE_ARGS+=--disable-gtk-doc
  ```

  The default location of installed HTML files (`share/gtk-doc/<package-name>`) is correct and should not be changed unless the package insists on installing them somewhere else. Otherwise programs as **devhelp** will not be able to open them. You can do that with an entry similar to:

  ```
  CONFIGURE_ARGS+=--with-html-dir=${PREFIX}/share/gtk-doc/...
  ```

GNOME uses multiple *shared* directories and files under the installation prefix to maintain databases. In this context, shared means that those exact same directories and files are used among several different packages, leading to conflicts in the `PLIST`. pkgsrc currently includes functionality to handle the most common cases, so you have to forget about using `@unexec ${RMDIR}` lines in your file lists and omitting shared files from them. If you find yourself doing those, *your package is most likely incorrect*.

The following table lists the common situations that result in using shared directories or files. For each of them, the appropriate solution is given. After applying the solution be sure to *regenerate the package's file list* with **make print-PLIST** and ensure it is correct.

Table 23-1. PLIST handling for GNOME packages

If the package...	Then...
Installs OMF files under `share/omf`.	See Section 19.6.10.
Installs icons under the `share/icons/hicolor` hierarchy or updates `share/icons/hicolor/icon-theme.cache`.	See Section 19.6.19.
Installs files under `share/mime/packages`.	See Section 19.6.14.
Installs `.desktop` files under `share/applications` and these include MIME information.	See Section 19.6.20.

23.3. Updating GNOME to a newer version

When seeing GNOME as a whole, there are two kinds of updates:

Major update

Given that there is still a very long way for GNOME 3 (if it ever appears), we consider a major update one that goes from a `2.X` version to a `2.Y` one, where `Y` is even and greater than `X`. These are hard to achieve because they introduce lots of changes in the components' code and almost all GNOME distfiles are updated to newer versions. Some of them can even break API and ABI compatibility with the previous major version series. As a result, the update needs to be done all at once to minimize breakage.

A major update typically consists of around 80 package updates and the addition of some new ones.

Minor update

We consider a minor update one that goes from a `2.A.X` version to a `2.A.Y` one where `Y` is greater than `X`. These are easy to achieve because they do not update all GNOME components, can be done in an incremental way and do not break API nor ABI compatibility.

A minor update typically consists of around 50 package updates, although the numbers here may vary a lot.

In order to update the GNOME components in pkgsrc to a new stable release (either major or minor), the following steps should be followed:

1. Get a list of all the tarballs that form the new release by using the following commands. These will leave the full list of the components' distfiles into the `list.txt` file:

```
% echo ls "*.tar.bz2" | \
    ftp -V ftp://ftp.gnome.org/pub/gnome/platform/x.y/x.y.z/sources/ | \
    awk '{ print $9 }' >list.txt
% echo ls "*.tar.bz2" | \
    ftp -V ftp://ftp.gnome.org/pub/gnome/desktop/x.y/x.y.z/sources/ | \
    awk '{ print $9 }' >>list.txt
```

2. Open each meta package's `Makefile` and bump their version to the release you are updating them to. The three meta packages should be always consistent with versioning. Obviously remove any `PKGREVISION`s that might be in them.
3. For each meta package, update all its `DEPENDS` lines to match the latest versions as shown by the above commands. Do *not* list any newer version (even if found in the FTP) because the meta packages are supposed to list the exact versions that form a specific GNOME release. Exceptions are permitted here if a newer version solves a serious issue in the overall desktop experience; these typically come in the form of a revision bump in pkgsrc, not in newer versions from the developers.

   Packages not listed in the `list.txt` file should be updated to the latest version available (if found in pkgsrc). This is the case, for example, of the dependencies on the GNU Autotools in the `meta-pkgs/gnome-devel` meta package.
4. Generate a patch from the modified meta packages and extract the list of "new" lines. This will provide you an outline on what packages need to be updated in pkgsrc and in what order:

   ```
   % cvs diff -u gnome-devel gnome-base gnome | grep '^+D' >todo.txt
   ```
5. For major desktop updates it is recommended to zap all your installed packages and start over from scratch at this point.
6. Now comes the longest step by far: iterate over the contents of `todo.txt` and update the packages listed in it in order. For major desktop updates none of these should be committed until the entire set is completed because there are chances of breaking not-yet-updated packages.
7. Once the packages are up to date and working, commit them to the tree one by one with appropriate log messages. At the end, commit the three meta package updates and all the corresponding changes to the `doc/CHANGES-<YEAR>` and `pkgsrc/doc/TODO` files.

23.4. Patching guidelines

GNOME is a very big component in pkgsrc which approaches 100 packages. Please, it is very important that you always, always, **always** feed back any portability fixes you do to a GNOME package to the mainstream developers (see Section 11.3.5). This is the only way to get their attention on portability issues and to ensure that future versions can be built out-of-the box on NetBSD. The less custom patches in pkgsrc, the easier further updates are. Those developers in charge of issuing major GNOME updates will be grateful if you do that.

The most common places to report bugs are the GNOME's Bugzilla (http://bugzilla.gnome.org/) and the freedesktop.org's Bugzilla (http://bugzilla.freedesktop.org/). Not all components use these to track bugs, but most of them do. Do not be short on your reports: always provide detailed explanations of the current failure, how it can be improved to achieve maximum portability and, if at all possible, provide a patch against CVS head. The more verbose you are, the higher chances of your patch being accepted.

Also, please avoid using preprocessor magic to fix portability issues. While the FreeBSD GNOME people are doing a great job in porting GNOME to their operating system, the official GNOME sources are now plagued by conditionals that check for `__FreeBSD__` and similar macros. This hurts portability. Please see our patching guidelines (Section 11.3.4) for more details.

III. The pkgsrc infrastructure internals

This part of the guide deals with everything from the infrastructure that is behind the interfaces described in the developer's guide. A casual package maintainer should not need anything from this part.

Chapter 24.
Design of the pkgsrc infrastructure

The pkgsrc infrastructure consists of many small Makefile fragments. Each such fragment needs a properly specified interface. This chapter explains how such an interface looks like.

24.1. The meaning of variable definitions

Whenever a variable is defined in the pkgsrc infrastructure, the location and the way of definition provide much information about the intended use of that variable. Additionally, more documentation may be found in a header comment or in this pkgsrc guide.

A special file is `mk/defaults/mk.conf`, which lists all variables that are intended to be user-defined. They are either defined using the `?=` operator or they are left undefined because defining them to anything would effectively mean "yes". All these variables may be overridden by the pkgsrc user in the `MAKECONF` file.

Outside this file, the following conventions apply: Variables that are defined using the `?=` operator may be overridden by a package.

Variables that are defined using the `=` operator may be used read-only at run-time.

Variables whose name starts with an underscore must not be accessed outside the pkgsrc infrastructure at all. They may change without further notice.

Note: These conventions are currently not applied consistently to the complete pkgsrc infrastructure.

24.2. Avoiding problems before they arise

All variables that contain lists of things should default to being empty. Two examples that do not follow this rule are `USE_LANGUAGES` and `DISTFILES`. These variables cannot simply be modified using the `+=` operator in package `Makefile`s (or other files included by them), since there is no guarantee whether the variable is already set or not, and what its value is. In the case of `DISTFILES`, the packages "know" the default value and just define it as in the following example.

```
DISTFILES=      ${DISTNAME}${EXTRACT_SUFX} additional-files.tar.gz
```

Because of the selection of this default value, the same value appears in many package Makefiles. Similarly for USE_LANGUAGES, but in this case the default value ("c") is so short that it doesn't stand out. Nevertheless it is mentioned in many files.

24.3. Variable evaluation

24.3.1. At load time

Variable evaluation takes place either at load time or at runtime, depending on the context in which they occur. The contexts where variables are evaluated at load time are:

- The right hand side of the := and != operators,
- Make directives like .if or .for,
- Dependency lines.

A special exception are references to the iteration variables of .for loops, which are expanded inline, no matter in which context they appear.

As the values of variables may change during load time, care must be taken not to evaluate them by accident. Typical examples for variables that should not be evaluated at load time are DEPENDS and CONFIGURE_ARGS. To make the effect more clear, here is an example:

```
CONFIGURE_ARGS=         # none
CFLAGS=                 -O
CONFIGURE_ARGS+=        CFLAGS=${CFLAGS:Q}

CONFIGURE_ARGS:=        ${CONFIGURE_ARGS}

CFLAGS+=                -Wall
```

This code shows how the use of the := operator can quickly lead to unexpected results. The first paragraph is fairly common code. The second paragraph evaluates the CONFIGURE_ARGS variable, which results in CFLAGS=-O. In the third paragraph, the -Wall is appended to the CFLAGS, but this addition will not appear in CONFIGURE_ARGS. In actual code, the three paragraphs from above typically occur in completely unrelated files.

24.3.2. At runtime

After all the files have been loaded, the values of the variables cannot be changed anymore. Variables that are used in the shell commands are expanded at this point.

24.4. How can variables be specified?

There are many ways in which the definition and use of a variable can be restricted in order to detect bugs and violations of the (mostly unwritten) policies. See the `pkglint` developer's documentation for further details.

24.5. Designing interfaces for Makefile fragments

Most of the `.mk` files fall into one of the following classes. Cases where a file falls into more than one class should be avoided as it often leads to subtle bugs.

24.5.1. Procedures with parameters

In a traditional imperative programming language some of the `.mk` files could be described as procedures. They take some input parameters and—after inclusion—provide a result in output parameters. Since all variables in `Makefiles` have global scope care must be taken not to use parameter names that have already another meaning. For example, `PKGNAME` is a bad choice for a parameter name.

Procedures are completely evaluated at preprocessing time. That is, when calling a procedure all input parameters must be completely resolvable. For example, `CONFIGURE_ARGS` should never be an input parameter since it is very likely that further text will be added after calling the procedure, which would effectively apply the procedure to only a part of the variable. Also, references to other variables wit will be modified after calling the procedure.

A procedure can declare its output parameters either as suitable for use in preprocessing directives or as only available at runtime. The latter alternative is for variables that contain references to other runtime variables.

Procedures shall be written such that it is possible to call the procedure more than once. That is, the file must not contain multiple-inclusion guards.

Examples for procedures are `mk/bsd.options.mk` and `mk/buildlink3/bsd.builtin.mk`. To express that the parameters are evaluated at load time, they should be assigned using the `:=` operator, which should be used only for this purpose.

24.5.2. Actions taken on behalf of parameters

Action files take some input parameters and may define runtime variables. They shall not define loadtime variables. There are action files that are included implicitly by the pkgsrc infrastructure, while other must be included explicitly.

An example for action files is `mk/subst.mk`.

24.6. The order in which files are loaded

Package `Makefiles` usually consist of a set of variable definitions, and include the file `../../mk/bsd.pkg.mk` in the very last line. Before that, they may also include various other `*.mk` files if they need to query the availability of certain features like the type of compiler or the X11

implementation. Due to the heavy use of preprocessor directives like `.if` and `.for`, the order in which the files are loaded matters.

This section describes at which point the various files are loaded and gives reasons for that order.

24.6.1. The order in `bsd.prefs.mk`

The very first action in `bsd.prefs.mk` is to define some essential variables like `OPSYS`, `OS_VERSION` and `MACHINE_ARCH`.

Then, the user settings are loaded from the file specified in `MAKECONF`, which is usually `mk.conf`. After that, those variables that have not been overridden by the user are loaded from `mk/defaults/mk.conf`.

After the user settings, the system settings and platform settings are loaded, which may override the user settings.

Then, the tool definitions are loaded. The tool wrappers are not yet in effect. This only happens when building a package, so the proper variables must be used instead of the direct tool names.

As the last steps, some essential variables from the wrapper and the package system flavor are loaded, as well as the variables that have been cached in earlier phases of a package build.

24.6.2. The order in `bsd.pkg.mk`

First, `bsd.prefs.mk` is loaded.

Then, the various `*-vars.mk` files are loaded, which fill default values for those variables that have not been defined by the package. These variables may later be used even in unrelated files.

Then, the file `bsd.pkg.error.mk` provides the target `error-check` that is added as a special dependency to all other targets that use `DELAYED_ERROR_MSG` or `DELAYED_WARNING_MSG`.

Then, the package-specific hacks from `hacks.mk` are included.

Then, various other files follow. Most of them don't have any dependencies on what they need to have included before or after them, though some do.

The code to check `PKG_FAIL_REASON` and `PKG_SKIP_REASON` is then executed, which restricts the use of these variables to all the files that have been included before. Appearances in later files will be silently ignored.

Then, the files for the main targets are included, in the order of later execution, though the actual order should not matter.

At last, some more files are included that don't set any interesting variables but rather just define make targets to be executed.

Chapter 25.
Regression tests

The pkgsrc infrastructure consists of a large codebase, and there are many corners where every little bit of a file is well thought out, making pkgsrc likely to fail as soon as anything is changed near those parts. To prevent most changes from breaking anything, a suite of regression tests should go along with every important part of the pkgsrc infrastructure. This chapter describes how regression tests work in pkgsrc and how you can add new tests.

25.1. The regression tests framework

25.2. Running the regression tests

You first need to install the `pkgtools/pkg_regress` package, which provides the **pkg_regress** command. Then you can simply run that command, which will run all tests in the `regress` category.

25.3. Adding a new regression test

Every directory in the `regress` category that contains a file called `spec` is considered a regression test. This file is a shell program that is included by the **pkg_regress** command. The following functions can be overridden to suit your needs.

25.3.1. Overridable functions

These functions do not take any parameters. They are all called in "set -e" mode, so you should be careful to check the exitcodes of any commands you run in the test.

`do_setup()`

This function prepares the environment for the test. By default it does nothing.

`do_test()`

This function runs the actual test. By default, it calls `TEST_MAKE` with the arguments `MAKEARGS_TEST` and writes its output including error messages into the file `TEST_OUTFILE`.

`check_result()`

This function is run after the test and is typically used to compare the actual output from the one that is expected. It can make use of the various helper functions from the next section.

`do_cleanup()`

This function cleans everything up after the test has been run. By default it does nothing.

25.3.2. Helper functions

`exit_status(expected)`

This function compares the exitcode of the **do_test()** function with its first parameter. If they differ, the test will fail.

`output_require(regex...)`

This function checks for each of its parameters if the output from **do_test()** matches the extended regular expression. If it does not, the test will fail.

`output_prohibit(regex...)`

This function checks for each of its parameters if the output from **do_test()** does *not* match the extended regular expression. If any of the regular expressions matches, the test will fail.

Chapter 26.
Porting pkgsrc

The pkgsrc system has already been ported to many operating systems, hardware architectures and compilers. This chapter explains the necessary steps to make pkgsrc even more portable.

26.1. Porting pkgsrc to a new operating system

To port pkgsrc to a new operating system (called `MyOS` in this example), you need to touch the following files:

`pkgtools/bootstrap-mk-files/files/mods/`*`MyOS`*`.sys.mk`

This file contains some basic definitions, for example the name of the C compiler.

`mk/bsd.prefs.mk`

Insert code that defines the variables `OPSYS`, `OS_VERSION`, `LOWER_OS_VERSION`, `LOWER_VENDOR`, `MACHINE_ARCH`, `OBJECT_FMT`, `APPEND_ELF`, and the other variables that appear in this file.

`mk/platform/`*`MyOS`*`.mk`

This file contains the platform-specific definitions that are used by pkgsrc. Start by copying one of the other files and edit it to your needs.

`mk/tools/tools.`*`MyOS`*`.mk`

This file defines the paths to all the tools that are needed by one or the other package in pkgsrc, as well as by pkgsrc itself. Find out where these tools are on your platform and add them.

Now, you should be able to build some basic packages, like `lang/perl5`, `shells/bash`.

26.2. Adding support for a new compiler

TODO

Appendix A.
A simple example package: bison

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

A.1. files

A.1.1. Makefile

```
# $NetBSD: pkgsrc.ps,v 1.90 2015/03/07 21:32:40 tnn Exp $
#

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

MAINTAINER=     pkgsrc-users@NetBSD.org
HOMEPAGE=       http://www.gnu.org/software/bison/bison.html
COMMENT=        GNU yacc clone

GNU_CONFIGURE=  yes
INFO_FILES=     yes

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

A.1.2. 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.
```

A.1.3. PLIST

```
@comment $NetBSD: pkgsrc.ps,v 1.90 2015/03/07 21:32:40 tnn Exp $
bin/bison
man/man1/bison.1.gz
```

```
share/bison.simple
share/bison.hairy
```

A.1.4. Checking a package with pkglint

The NetBSD package system comes with `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 wish to examine and execute **pkglint**:

```
$ pkglint
looks fine.
```

Depending on the supplied command line arguments (see pkglint(1)), more checks will be performed. Use e.g. **pkglint -Call -Wall** for a very thorough check.

A.2. Steps for building, installing, packaging

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

```
# cd /usr/pkgsrc/lang
# mkdir bison
# cd bison
# mkdir patches
```

Create `Makefile`, `DESCR` and `PLIST` (see Chapter 11) then continue with fetching the distfile:

```
# 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://orpheus.amdahl.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://orpheus.amdah
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://orpheus
Successfully retrieved file.
```

Generate the checksum of the distfile into `distinfo`:

```
# make makedistinfo
```

Now compile:

```
# 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
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DXPFILE=\"/usr/pkg/share/bison.simple\"  -DXPFILE1=\"/usr/pkg/share/bison.hairy\" -DS
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOCA=1
cc  -g -o bison LR0.o allocate.o closure.o conflicts.o derives.o files.o         getargs.o g
./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:

```
# 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 $f /usr/pkg/info/$f
/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**. Should you decide that you want a binary package, do this now:

```
# 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:

```
# make clean
===>  Cleaning for bison-1.25
```

Appendix B.
Build logs

B.1. Building figlet

```
# make
===> Checking for vulnerabilities in figlet-2.2.1nb2
=> figlet221.tar.gz doesn't seem to exist on this system.
=> Attempting to fetch figlet221.tar.gz from ftp://ftp.figlet.org/pub/figlet/program/unix/.
=> [172219 bytes]
Connected to ftp.plig.net.
220 ftp.plig.org NcFTPd Server (licensed copy) ready.
331 Guest login ok, send your complete e-mail address as password.
230-You are user #5 of 500 simultaneous users allowed.
230-
230-  ___ _             _ _
230- |  _| |_ ___   ___| |_|___   ___ ___ ___
230- |  _|  _| . |_| . | | | . |_| . |  _| . |
230- |_| |_| |  _|_|  _|_|_|_  |_|___|_| |_  |
230-         |_|   |_|     |___|         |___|
230-
230-** Welcome to ftp.plig.org **
230-
230-Please note that all transfers from this FTP site are logged. If you
230-do not like this, please disconnect now.
230-
230-This archive is available via
230-
230-HTTP:  http://ftp.plig.org/
230-FTP:   ftp://ftp.plig.org/     (max 500 connections)
230-RSYNC: rsync://ftp.plig.org/   (max  30 connections)
230-
230-Please email comments, bug reports and requests for packages to be
230-mirrored to ftp-admin@plig.org.
230-
230-
230 Logged in anonymously.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Type okay.
250 "/pub" is new cwd.
250-"/pub/figlet" is new cwd.
250-
250-Welcome to the figlet archive at ftp.figlet.org
250-
250- ftp://ftp.figlet.org/pub/figlet/
```

```
250-
250-The official FIGlet web page is:
250- http://www.figlet.org/
250-
250-If you have questions, please mailto:info@figlet.org. If you want to
250-contribute a font or something else, you can email us.
250
250 "/pub/figlet/program" is new cwd.
250 "/pub/figlet/program/unix" is new cwd.
local: figlet221.tar.gz remote: figlet221.tar.gz
502 Unimplemented command.
227 Entering Passive Mode (195,40,6,41,246,104)
150 Data connection accepted from 84.128.86.72:65131; transfer starting for figlet221.tar.gz
38% |**************                      | 65800      64.16 KB/s    00:01 ETA
226 Transfer completed.
172219 bytes received in 00:02 (75.99 KB/s)
221 Goodbye.
=> Checksum OK for figlet221.tar.gz.
===> Extracting for figlet-2.2.1nb2
===> Required installed package ccache-[0-9]*: ccache-2.3nb1 found
===> Patching for figlet-2.2.1nb2
===> Applying pkgsrc patches for figlet-2.2.1nb2
===> Overriding tools for figlet-2.2.1nb2
===> Creating toolchain wrappers for figlet-2.2.1nb2
===> Configuring for figlet-2.2.1nb2
===> Building for figlet-2.2.1nb2
gcc -O2 -DDEFAULTFONTDIR=\"/usr/pkg/share/figlet\"  -DDEFAULTFONTFILE=\"standard.flf\"  figl
chmod a+x figlet
gcc -O2 -o chkfont chkfont.c
=> Unwrapping files-to-be-installed.
#
# make install
===> Checking for vulnerabilities in figlet-2.2.1nb2
===> Installing for figlet-2.2.1nb2
install -d -o root -g wheel -m 755 /usr/pkg/bin
install -d -o root -g wheel -m 755 /usr/pkg/man/man6
mkdir -p /usr/pkg/share/figlet
cp figlet /usr/pkg/bin
cp chkfont /usr/pkg/bin
chmod 555 figlist showfigfonts
cp figlist /usr/pkg/bin
cp showfigfonts /usr/pkg/bin
cp fonts/*.flf /usr/pkg/share/figlet
cp fonts/*.flc /usr/pkg/share/figlet
cp figlet.6 /usr/pkg/man/man6
===> Registering installation for figlet-2.2.1nb2
#
```

B.2. Packaging figlet

```
# make package
===> Checking for vulnerabilities in figlet-2.2.1nb2
===> Packaging figlet-2.2.1nb2
===> Building binary package for figlet-2.2.1nb2
Creating package /home/cvs/pkgsrc/packages/i386/All/figlet-2.2.1nb2.tgz
Using SrcDir value of /usr/pkg
Registering depends:.
#
```

Appendix C.
Directory layout of the pkgsrc FTP server

As in other big projects, the directory layout of pkgsrc is quite complex for newbies. This chapter explains where you find things on the FTP server. The base directory on `ftp.NetBSD.org` is `/pub/pkgsrc/` (ftp://ftp.NetBSD.org/pub/pkgsrc/). On other servers it may be different, but inside this directory, everything should look the same, no matter on which server you are. This directory contains some subdirectories, which are explained below.

C.1. `distfiles`: The distributed source files

The directory `distfiles` contains lots of archive files from all pkgsrc packages, which are mirrored here. The subdirectories are called after their package names and are used when the distributed files have names that don't explicitly contain a version number or are otherwise too generic (for example `release.tar.gz`).

C.2. `misc`: Miscellaneous things

This directory contains things that individual pkgsrc developers find worth publishing.

C.3. `packages`: Binary packages

This directory contains binary packages for the various platforms that are supported by pkgsrc. Each subdirectory is of the form *OPSYS*/*ARCH*/*OSVERSION_TAG*. The meaning of these variables is:

- `OPSYS` is the name of the operating system for which the packages have been built. The name is taken from the output of the **uname** command, so it may differ from the one you are used to hear.
- `ARCH` is the hardware architecture of the platform for which the packages have been built. It also includes the `ABI` (Application Binary Interface) for platforms that have several of them.
- `OSVERSION` is the version of the operating system. For version numbers that change often (for example NetBSD-current), the often-changing part should be replaced with an `x`, for example `4.99.x`.
- `TAG` is either `20`*xx*`Q`*y* for a stable branch, or `head` for packages built from the HEAD branch. The latter should only be used when the packages are updated on a regular basis. Otherwise the date from checking out pkgsrc should be appended, for example `head_20071015`.

The rationale for exactly this scheme is that the pkgsrc users looking for binary packages can quickly click through the directories on the server and find the best binary packages for their machines. Since they usually know the operating system and the hardware architecture, OPSYS and ARCH are placed first. After these choices, they can select the best combination of OSVERSION and TAG together, since it is usually the case that packages stay compatible between different version of the operating system.

In each of these directories, there is a whole binary packages collection for a specific platform. It has a directory called `All` which contains all binary packages. Besides that, there are various category directories that contain symbolic links to the real binary packages.

C.4. `reports`: Bulk build reports

Here are the reports from bulk builds, for those who want to fix packages that didn't build on some of the platforms. The structure of subdirectories should look like the one in Section C.3.

C.5. `current`, `pkgsrc-20`*`xx`*`Q`*`y`*: source packages

These directories contain the "real" pkgsrc, that is the files that define how to create binary packages from source archives.

The directory `pkgsrc` contains a snapshot of the CVS repository, which is updated regularly. The file `pkgsrc.tar.gz` contains the same as the directory, ready to be downloaded as a whole.

In the directories for the quarterly branches, there is an additional file called `pkgsrc-20`*`xx`*`Q`*`y`*`.tar.gz`, which contains the state of pkgsrc when it was branched.

Appendix D.
Editing guidelines for the pkgsrc guide

This section contains information on editing the pkgsrc guide itself.

D.1. Make targets

The pkgsrc guide's source code is stored in `pkgsrc/doc/guide/files`, and several files are created from it:

- `pkgsrc/doc/pkgsrc.txt`
- `pkgsrc/doc/pkgsrc.html`
- http://www.NetBSD.org/docs/pkgsrc/
- http://www.NetBSD.org/docs/pkgsrc/pkgsrc.pdf: The PDF version of the pkgsrc guide.
- http://www.NetBSD.org/docs/pkgsrc/pkgsrc.ps: PostScript version of the pkgsrc guide.

D.2. Procedure

The procedure to edit the pkgsrc guide is:

1. Make sure you have the packages needed to regenerate the pkgsrc guide (and other XML-based NetBSD documentation) installed. These are automatically installed when you install the `meta-pkgs/pkgsrc-guide-tools` package.
2. Run **cd doc/guide** to get to the right directory. All further steps will take place here.
3. Edit the XML file(s) in `files/`.
4. Run **bmake** to check the pkgsrc guide for valid XML and to build the final output files. If you get any errors at this stage, you can just edit the files, as there are only symbolic links in the working directory, pointing to the files in `files/`.
5. **(cd files && cvs commit)**
6. Run **bmake clean && bmake** to regenerate the output files with the proper RCS Ids.
7. Run **bmake regen** to install and commit the files in both `pkgsrc/doc` and `htdocs`.

> **Note:** If you have added, removed or renamed some chapters, you need to synchronize them using **cvs add** or **cvs delete** in the htdocs directory.