The pkgsrc guide

Documentation on the NetBSD packages system

(2006/02/18)

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 2006/02/18 01:46:43

Information about using the NetBSD package system (pkgsrc) from both a user view for installing packages as well as from a pkgsrc developers' view for creating new packages.

Table of Contents

Chapter 1.
What is pkgsrc?

1.1. Introduction

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

Once the software has been built, it is manipulated with the **pkg_*** tools so that installation and de-installation, printing of an inventory of all installed packages and retrieval of one-line comments or more verbose descriptions are all simple.

pkgsrc currently contains several thousand packages, including:

- `www/apache` - The Apache web server
- `www/mozilla` - The Mozilla 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.

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:

- Darwin (http://developer.apple.com/darwin/) (Mac OS X (http://www.apple.com/macosx/))
- DragonFly BSD (http://www.DragonFlyBSD.org/)
- FreeBSD (http://www.FreeBSD.org/)
- Microsoft Windows, via Interix (http://www.microsoft.com/windows/sfu/)
- IRIX (http://www.sgi.com/software/irix6.5/)
- Linux (http://www.linux.org/)
- NetBSD (http://www.NetBSD.org/) (of course)
- Tru64 (http://h30097.www3.hp.com/) (Digital UNIX, OSF1)
- OpenBSD (http://www.openbsd.org/)
- Solaris (http://www.sun.com/solaris/)

1.2. Overview

This document is divided into two 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.

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/NetBSD/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.

1.4. 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

There are three ways to get pkgsrc. Either as a tar file, via SUP, or via CVS. All three ways are described here.

2.1. As tar file

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

2.2. Via SUP

As an alternative to the tar file, you can get pkgsrc via the Software Update Protocol, SUP. To do so, make sure your supfile has a line

```
release=pkgsrc
```

in it, see the examples in `/usr/share/examples/supfiles`, and that the `/usr/pkgsrc` directory exists. Then, simply run **sup -v *`/path/to/your/supfile`***.

2.3. Via CVS

To get pkgsrc via CVS, make sure you have “cvs” installed. To do an initial (full) checkout of pkgsrc, do the following steps:

```
% setenv CVSROOT anoncvs@anoncvs.NetBSD.org:/cvsroot
% setenv CVS_RSH ssh
% cd /usr
% cvs checkout -P pkgsrc
```

This will create the `pkgsrc` directory in your `/usr`, and all the package source will be stored under `/usr/pkgsrc`. To update pkgsrc after the initial checkout, make sure you have `CVS_RSH` set as above, then do:

```
% cd /usr/pkgsrc
% cvs -q update -dP
```

Please also note that it is possible to have multiple copies of the pkgsrc hierarchy in use at any one time - all work is done relatively within the pkgsrc tree.

2.4. Keeping pkgsrc up-to-date via CVS

If your copy of pkgsrc contains a lot of `CVS` directories, you can update it using the cvs(1) program. First, **cd** to the top level directory of pkgsrc. Then run **cvs -q update -dP**, and you're done.

If that doesn't work and the file `CVS/Root` contains the string ":pserver:", you have to run **cvs login** once to get known to the NetBSD CVS server. The **cvs** utility will then ask you for a password. Just enter "anoncvs". Then try again to update.

Chapter 3.
Using pkgsrc on systems other than NetBSD

3.1. Bootstrapping pkgsrc

For operating systems other than NetBSD, we provide a bootstrap kit to build the required tools to use pkgsrc on your platform. Besides support for native NetBSD, pkgsrc and the bootstrap kit have support for the following operating systems:

- Darwin (Mac OS X)
- DragonFly BSD
- FreeBSD
- Interix (Windows 2000, XP, 2003)
- IRIX
- Linux
- OpenBSD
- Solaris
- Tru64 (Digital UNIX/OSF1)

Support for other platforms is under development.

Installing the bootstrap kit should be as simple as:

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

See Chapter 2 for other ways to get pkgsrc before bootstrapping. The given **bootstrap** command will use the defaults of `/usr/pkg` for the *prefix* where programs will be installed in, and `/var/db/pkg` for the package database directory where pkgsrc will do its internal bookkeeping. However, these can also be set using command-line arguments.

Binary packages for the pkgsrc tools and an initial set of packages is available for supported platforms. An up-to-date list of these can be found on www.pkgsrc.org (http://www.pkgsrc.org/). Note that this only works for privileged builds that install into `/usr/pkg`.

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

3.2. Platform-specific notes

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

3.2.1. Darwin (Mac OS X)

Darwin 5.x and 6.x are supported. There are two methods of using pkgsrc on Mac OS X, by using a disk image, or a UFS partition.

Before you start, you will need to download and install the Mac OS X Developer Tools from Apple's Developer Connection. See http://developer.apple.com/macosx/ for details. Also, make sure you install X11 for Mac OS X and the X11 SDK from http://www.apple.com/macosx/x11/download/ if you intend to build packages that use the X11 Window System.

If you already have a UFS partition, or have a spare partition that you can format as UFS, it is recommended to use that instead of the disk image. It'll be somewhat faster and will mount automatically at boot time, where you must manually mount a disk image.

> **Note:** You cannot use a HFS+ file system for pkgsrc, because pkgsrc currently requires the file system to be case-sensitive, and HFS+ is not.

3.2.1.1. Using a disk image

Create the disk image:

```
# cd pkgsrc/bootstrap
# ./ufsdiskimage create ~/Documents/NetBSD 512 # megabytes - season to taste
# ./ufsdiskimage mount ~/Documents/NetBSD
# sudo chown ‘id -u‘:‘id -g‘ /Volumes/NetBSD
```

That's it!

3.2.1.2. Using a UFS partition

By default, `/usr` will be on your root file system, normally HFS+. It is possible to use the default *prefix* of `/usr/pkg` by symlinking `/usr/pkg` to a directory on a UFS file system. Obviously, another symlink is required if you want to place the package database directory outside the *prefix*. e.g.

```
# ./bootstrap --pkgdbdir /usr/pkg/pkgdb
```

If you created your partitions at the time of installing Mac OS X and formatted the target partition as UFS, it should automatically mount on `/Volumes/<volume name>` when the machine boots. If you are (re)formatting a partition as UFS, you need to ensure that the partition map correctly reflects "Apple_UFS" and not "Apple_HFS".

The problem is that none of the disk tools will let you touch a disk that is booted from. You can unmount the partition, but even if you newfs it, the partition type will be incorrect and the automounter won't mount it. It can be mounted manually, but it won't appear in Finder.

You'll need to boot off of the OS X Installation (User) CD. When the Installation program starts, go up to the menu and select Disk Utility. Now, you will be able to select the partition you want to be UFS, and Format it Apple UFS. Quit the Disk Utility, quit the installer which will reboot your machine. The new UFS file system will appear in Finder.

Be aware that the permissions on the new file system will be writable by root only.

This note is as of 10.2 (Jaguar) and applies to earlier versions. Hopefully Apple will fix Disk Utility in 10.3 (Panther).

3.2.2. FreeBSD

FreeBSD 4.7 and 5.0 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 `/etc/mk.conf` file will be placed in `/etc/mk.conf.example` file when you use the bootstrap script.

3.2.3. 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, current as of this writing, 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.)

3.2.3.1. 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, a NetBSD developer has made most Interix hotfixes available for personal use from http://www.duh.org/interix/hotfixes.php.

3.2.3.2. 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.

3.2.3.3. 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:
```

3.2.3.4. 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/).

  Also, StarNet Communications has graciously provided a free version of their X-Win32 product that accepts connections only from localhost: X-Win32 LX (http://www.starnet.com/xwin32LX/get_xwin32LX.htm), recommended by the maintainer of Interix pkgsrc support.

- **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).

3.2.3.5. 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
```

3.2.4. 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 can not 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 `/etc/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 `/etc/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 MIPRPro is used, please set your `PATH` to *not* include the location of gcc (often `/usr/freeware/bin`), and (important) pass the '--preserve-path' flag.

3.2.5. 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 CC=/opt/intel_cc_80/bin/icc LDFLAGS=-static-libcxa \
        ac_cv___attribute__=yes ./bootstrap
```

Note: icc 8.1 needs the '-i-static' argument instead of -static-libcxa.

icc supports __attribute__, but the GNU configure test uses a nested function, which icc does not support. #undef'ing __attribute__ has the unfortunate side-effect of breaking many of the Linux header files, which cannot be compiled properly without __attribute__. The test must be overridden so that __attribute__ is assumed supported by the compiler.

After bootstrapping, you should set `PKGSRC_COMPILER` in `/etc/mk.conf`:

```
PKGSRC_COMPILER=        icc
```

The default installation directory for icc is `/opt/intel_cc_80`, which is also the pkgsrc default. If you have installed it into a different directory, set `ICCBASE` in `/etc/mk.conf`:

```
ICCBASE=                /opt/icc
```

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.

3.2.6. OpenBSD

OpenBSD 3.0 and 3.2 are tested and supported.

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 `/etc/mk.conf` file will be placed in `/etc/mk.conf.example` file when you use the bootstrap script. OpenBSD's make program uses `/etc/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
```

3.2.7. Solaris

Solaris 2.6 through 9 are supported on both x86 and sparc. You will need a working C compiler. Both gcc 2.95.3 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 the use of GNU binutils on Solaris is *not* supported.

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}`.

3.2.7.1. 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/gcc` or install a binary gcc package, then remove gcc used during bootstrapping.

Binary packages of gcc can be found through
http://www.sun.com/bigadmin/common/freewareSearch.html.

3.2.7.2. 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 `CC`, `CXX` and optionally, `CPP` in `/etc/mk.conf`, e.g.:

```
CC=     cc
CXX=    CC
CPP=    /usr/ccs/lib/cpp
```

3.2.7.3. Buildling 64-bit binaries with SunPro

Building 64-bit binaries is a little trickier. First, you need to bootstrap pkgsrc in 64-bit mode. One problem here is that while building one of the programs in the bootstrap kit (`bmake`), the `CFLAGS` variable is not honored, even if it is set in the environment. To work around this bug, you can create a simple shell script called `cc64` and put it somewhere in the `PATH`:

```
#! /bin/sh
exec /opt/SUNWspro/bin/cc -xtarget=ultra -xarch=v9 ${1+"$@"}
```

Then, pass the definition for `CC` in the environment of the **bootstrap** command:

```
$ cd bootstrap
$ CC=cc64 ./bootstrap
```

After bootstrapping, there are two alternative ways, depending on whether you want to find bugs in packages or get your system ready quickly. If you just want a running system, add the following lines to your `mk.conf` file:

```
CC=                     cc64
CXX=                    CC64
PKGSRC_COMPILER=        sunpro
```

This way, all calls to the compiler will be intercepted by the above wrapper and therefore get the necessary ABI options automatically. (Don't forget to create the shell script `CC64`, too.)

To find packages that ignore the user-specified `CFLAGS` and `CXXFLAGS`, add the following lines to your `mk.conf` file:

```
CC=                     cc
CXX=                    CC
PKGSRC_COMPILER=        sunpro
CFLAGS=                 -xtarget=ultra -xarch=v9
CXXFLAGS=               -xtarget=ultra -xarch=v9
LDFLAGS=                -xtarget=ultra -xarch=v9
```

Packages that don't use the flags provided in the configuration file will try to build 32-bit binaries and fail during linking. Detecting this is useful to prevent bugs on other platforms where the error would not show up but pass silently.

3.2.7.4. 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
```

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

To use binary packages, you need some tools to manage them. On NetBSD, these tools are already installed. On all other operating systems, you need to install them first. For the following platforms, prebuilt versions of the package tools are available and can simply be downloaded and unpacked in the `/` directory:

Platform	URL
Solaris 5.10	`http://public.enst.fr/pkgsrc/packages/bootstrap-p`

These prebuilt package tools use `/usr/pkg` for the base directory, and `/var/db/pkg` for the database of installed packages. If you cannot use these directories for whatever reasons (maybe because you’re not root), you have to build the package tools yourself, which is explained in Section 3.1.

4.1.1. Finding binary packages

To install binary packages, you first need to know from where to get them. You can get them on CD-ROMs, DVDs, or via FTP or HTTP.

For NetBSD, the binary packages are made available on `ftp.NetBSD.org` and its mirrors, in the directory `/pub/NetBSD/packages/`*`OSVERSION`*`/`*`ARCH`*`/`. For *`OSVERSION`*, you should insert the output of **uname -r**, and for *`ARCH`* the output of **uname -p**.

For some other platforms, binary packages can be found at the following locations:

Platform	URL
Solaris 5.10	`http://public.enst.fr/pkgsrc/packages/`

In each of these directories, there is a subdirectory `All` that contains all the binary packages. Further, there are subdirectories for categories that contain symbolic links that point to the actual binary package in `../All`. This directory layout is used for all package repositories, no matter if they are accessed via HTTP, FTP, NFS, CD-ROM, or the local filesystem.

4.1.2. Installing binary packages

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

```
# pkg_add /path/to/package.tgz
```

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

```
# pkg_add ftp://ftp.NetBSD.org/pub/NetBSD/packages/<OSVERSION>/<ARCH>/All/package.tgz
```

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

To save some typing, you can set the `PKG_PATH` environment variable to a semicolon-separated list of paths (including remote URLs); trailing slashes are not allowed.

Additionally to the `All` directory there exists a `vulnerable` directory to which binary packages with known vulnerabilities are moved, since removing them could cause missing dependencies. To use these packages, add the `vulnerable` directory to your `PKG_PATH`. However, you should run `security/audit-packages` regularly, especially after installing new packages, and verify that the vulnerabilities are acceptable for your configuration. An example `PKG_PATH` would be:
`ftp://ftp.NetBSD.org/pub/NetBSD/packages/<OSVERSION>/<ARCH>/All;ftp://ftp.NetBSD.org/pub/`
Please note that semicolon (';') is a shell meta-character, so you'll probably have to quote it.

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. 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

This 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 on a NetBSD system the "comp" and the "text" distribution sets must be installed. 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.

You can overwrite some of the major distribution sites to fit to sites that are close to your own. Have a look at `pkgsrc/mk/defaults/mk.conf` to find some examples — in particular, look for the `MASTER_SORT`, `MASTER_SORT_REGEX` and `INET_COUNTRY` definitions. 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 `/etc/mk.conf` file, and adding the definitions there.

If you don't have a permanent Internet connection and you want to know which files to download, **make fetch-list** will tell you what you'll need. Put these distfiles into `/usr/pkgsrc/distfiles`.

4.2.3. How to build and install

Assuming that the distfile has been fetched (see previous section), become root and change into the relevant directory and run **make**.

> **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, and

```
# make install
```

to install the various components into the correct places on your system. Installing the package on your system requires 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

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 `/etc/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 /etc/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 can not 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.

4.2.4. 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:

- distcc: distributed C/C++ (chainable)
- ccache: compiler cache (chainable)
- gcc: GNU C/C++ Compiler
- mipspro: Silicon Graphics, Inc. MIPSpro (n32/n64)
- mipspro: Silicon Graphics, Inc. MIPSpro (o32)

- `sunpro`: Sun Microsystems, Inc. WorkShip/Forte/Sun ONE Studio

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.

`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.

Chapter 5.
Configuring pkgsrc

5.1. General configuration

In this section, you can find some variables that apply to all pkgsrc packages. The preferred method of setting these variables is by setting them in `/etc/mk.conf`.

- `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 `${LOCALBASE}/cross`.
- `X11BASE`: Where X11 is installed on the system. The default is `/usr/X11R6`.
- `DISTDIR`: Where to store the downloaded copies of the original source distributions used for building pkgsrc packages. The default is `${PKGSRCDIR}/distfiles`.
- `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/NetBSD/packages/distfiles/${DIST_SUBDIR}/` and `ftp://ftp.freebsd.org/pub/FreeBSD/distfiles/${DIST_SUBDIR}/`.
- `BINPKG_SITES`: List of sites carrying binary pkgs.

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 8.3 for more information. *`rel`* and *`arch`* are replaced with OS release ("2.0", etc.) and architecture ("mipsel", etc.).

- 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.

5.3. Developer/advanced settings

XXX

- 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.
- ALLOW_VULNERABILITIES.*pkgbase*: A space separated list of vulnerability IDs that may be ignored when performing the automated security checks. These IDs are listed in the pkg-vulnerabilities file and are displayed by **audit-packages** when it finds a vulnerable package.
- SKIP_AUDIT_PACKAGES: If this is set to "yes", the automated security checks (which use the security/audit-packages package) will be **entirely** skipped for **all** packages built. Normally you'll want to use ALLOW_VULNERABILITIES instead of this.

5.4. 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 /etc/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" /etc/mk.conf
PKG_DEFAULT_OPTIONS=    -arts -dvdread -esound
PKG_OPTIONS.kdebase=    debug -sasl
PKG_OPTIONS.apache=     suexec
```

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 /etc/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 /etc/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 18 for information on how to submit such a binary package.

6.2. Settings for creation of binary packages

See Section 14.16.

6.3. Doing a bulk build of all packages

If you want to get a full set of precompiled binary packages, this section describes how to get them. Beware that the bulk build will remove all currently installed packages from your system!

Having an FTP server configured either on the machine doing the bulk builds or on a nearby NFS server can help to make the packages available to other machines that can then save time by installing only the binary packages. See ftpd(8) for more information. If you use a remote NFS server's storage, be sure to not actually compile on NFS storage, as this slows things down a lot.

6.3.1. Configuration

6.3.1.1. `build.conf`

The `build.conf` file is the main configuration file for bulk builds. You can configure how your copy of pkgsrc is kept up to date, how the distfiles are downloaded, how the packages are built and how the report

is generated. You can find an annotated example file in pkgsrc/mk/bulk/build.conf-example. To use it, copy build.conf-example to build.conf and edit it, following the comments in that file.

6.3.1.2. /etc/mk.conf

You may want to set variables in /etc/mk.conf. Look at pkgsrc/mk/defaults/mk.conf for details of the default settings. You will want to ensure that ACCEPTABLE_LICENSES meet your local policy. As used in this example, _ACCEPTABLE=yes accepts *all* licenses.

```
PACKAGES?=      ${_PKGSRCDIR}/packages/${MACHINE_ARCH}
WRKOBJDIR?=     /usr/tmp/pkgsrc   # build here instead of in pkgsrc
BSDSRCDIR=      /usr/src
BSDXSRCDIR=     /usr/xsrc         # for x11/xservers
OBJHOSTNAME?=   yes               # use work.‘hostname‘
FAILOVER_FETCH= yes               # insist on the correct checksum
PKG_DEVELOPER?= yes
_ACCEPTABLE=    yes
```

Some options that are especially useful for bulk builds can be found at the top lines of the file mk/bulk/bsd.bulk-pkg.mk. The most useful options of these are briefly described here.

- If you are on a slow machine, you may want to set USE_BULK_BROKEN_CHECK to “no”.
- If you are doing bulk builds from a read-only copy of pkgsrc, you have to set BULKFILESDIR to the directory where all log files are created. Otherwise the log files are created in the pkgsrc directory.
- Another important variable is BULK_PREREQ, which is a list of packages that should be always available while building other packages.

Some other options are scattered in the pkgsrc infrastructure:

- ALLOW_VULNERABLE_PACKAGES should be set to yes. The purpose of the bulk builds is creating binary packages, no matter if they are vulnerable or not. When uploading the packages to a public server, the vulnerable packages will be put into a directory of their own. Leaving this variable unset would prevent the bulk build system from even trying to build them, so possible building errors would not show up.
- CHECK_FILES (pkgsrc/mk/bsd.pkg.check.mk) can be set to “yes” to check that the installed set of files matches the PLIST.
- CHECK_INTERPRETER (pkgsrc/mk/bsd.pkg.check.mk) can be set to “yes” to check that the installed “#!”-scripts will find their interpreter.

6.3.1.3. pre-build.local

It is possible to configure the bulk build to perform certain site-specific tasks at the end of the pre-build stage. If the file pre-build.local exists in /usr/pkgsrc/mk/bulk, it will be executed (as a sh(1) script) at the end of the usual pre-build stage. An example use of pre-build.local is to have the line:

```
# echo "I do not have enough disk space to build this pig." \
    > pkgsrc/misc/openoffice/$BROKENF
```

to prevent the system from trying to build a particular package which requires nearly 3 GB of disk space.

6.3.2. Other environmental considerations

As `/usr/pkg` will be completely deleted at the start of bulk builds, make sure your login shell is placed somewhere else. Either drop it into `/usr/local/bin` (and adjust your login shell in the passwd file), or (re-)install it via pkg_add(1) from `/etc/rc.local`, so you can login after a reboot (remember that your current process won't die if the package is removed, you just can't start any new instances of the shell any more). Also, if you use NetBSD earlier than 1.5, or you still want to use the pkgsrc version of ssh for some reason, be sure to install ssh before starting it from `rc.local`:

```
( cd /usr/pkgsrc/security/ssh ; make bulk-install )
if [ -f /usr/pkg/etc/rc.d/sshd ]; then
    /usr/pkg/etc/rc.d/sshd
fi
```

Not doing so will result in you being not able to log in via ssh after the bulk build is finished or if the machine gets rebooted or crashes. You have been warned! :)

6.3.3. Operation

Make sure you don't need any of the packages still installed.

> **Warning**
>
> During the bulk build, *all packages will be removed!*

Be sure to remove all other things that might interfere with builds, like some libs installed in `/usr/local`, etc. then become root and type:

```
# cd /usr/pkgsrc
# sh mk/bulk/build
```

If for some reason your last build didn't complete (power failure, system panic, ...), you can continue it by running:

```
# sh mk/bulk/build restart
```

At the end of the bulk build, you will get a summary via mail, and find build logs in the directory specified by `FTP` in the `build.conf` file.

6.3.4. What it does

The bulk builds consist of three steps:

1. pre-build

   The script updates your pkgsrc tree via (anon)cvs, then cleans out any broken distfiles, and removes all packages installed.

2. the bulk build

   This is basically “make bulk-package” with an optimised order in which packages will be built. Packages that don’t require other packages will be built first, and packages with many dependencies will be built later.

3. post-build

   Generates a report that’s placed in the directory specified in the `build.conf` file named `broken.html`, a short version of that report will also be mailed to the build’s admin.

During the build, a list of broken packages will be compiled in `/usr/pkgsrc/.broken` (or `.../.broken.${MACHINE}` if `OBJMACHINE` is set), individual build logs of broken builds can be found in the package’s directory. These files are used by the bulk-targets to mark broken builds to not waste time trying to rebuild them, and they can be used to debug these broken package builds later.

6.3.5. Disk space requirements

Currently, roughly the following requirements are valid for NetBSD 2.0/i386:

- 10 GB - distfiles (NFS ok)
- 8 GB - full set of all binaries (NFS ok)
- 5 GB - temp space for compiling (local disk recommended)

Note that all pkgs will be de-installed as soon as they are turned into a binary package, and that sources are removed, so there is no excessively huge demand to disk space. Afterwards, if the package is needed again, it will be installed via pkg_add(1) instead of building again, so there are no cycles wasted by recompiling.

6.3.6. Setting up a sandbox for chrooted builds

If you don’t want all the packages nuked from a machine (rendering it useless for anything but pkg compiling), there is the possibility of doing the package bulk build inside a chroot environment.

The first step is to set up a chroot sandbox, e.g. `/usr/sandbox`. This can be done by using null mounts, or manually.

There is a shell script called `pkgsrc/mk/bulk/mksandbox` which will set up the sandbox environment using null mounts. It will also create a script called `sandbox` in the root of the sandbox environment, which will allow the null mounts to be activated using the **sandbox mount** command and deactivated using the **sandbox umount** command.

To set up a sandbox environment by hand, after extracting all the sets from a NetBSD installation or doing a **make distribution DESTDIR=/usr/sandbox** in `/usr/src/etc`, be sure the following items are present and properly configured:

1. Kernel

   ```
   # cp /netbsd /usr/sandbox
   ```

2. `/dev/*`

   ```
   # cd /usr/sandbox/dev ; sh MAKEDEV all
   ```

3. `/etc/resolv.conf` (for `security/smtpd` and mail):

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

4. Working(!) mail config (hostname, sendmail.cf):

   ```
   # cp /etc/mail/sendmail.cf /usr/sandbox/etc/mail
   ```

5. `/etc/localtime` (for `security/smtpd`):

   ```
   # ln -sf /usr/share/zoneinfo/UTC /usr/sandbox/etc/localtime
   ```

6. `/usr/src` (system sources, for `sysutils/aperture`, `net/ppp-mppe`):

   ```
   # ln -s ../disk1/cvs .
   # ln -s cvs/src-2.0 src
   ```

7. Create `/var/db/pkg` (not part of default install):

   ```
   # mkdir /usr/sandbox/var/db/pkg
   ```

8. Create `/usr/pkg` (not part of default install):

   ```
   # mkdir /usr/sandbox/usr/pkg
   ```

9. Checkout pkgsrc via cvs into `/usr/sandbox/usr/pkgsrc`:

   ```
   # cd /usr/sandbox/usr
   # cvs -d anoncvs@anoncvs.NetBSD.org:/cvsroot checkout -d -P pkgsrc
   ```

   Do not mount/link this to the copy of your pkgsrc tree you do development in, as this will likely cause problems!

10. Make `/usr/sandbox/usr/pkgsrc/packages` and `.../distfiles` point somewhere appropriate. NFS- and/or nullfs-mounts may come in handy!

11. Edit `/etc/mk.conf`, see Section 6.3.1.2.

12. Adjust `mk/bulk/build.conf` to suit your needs.

When the chroot sandbox is set up, you can start the build with the following steps:

```
# cd /usr/sandbox/usr/pkgsrc
# sh mk/bulk/do-sandbox-build
```

This will just jump inside the sandbox and start building. At the end of the build, mail will be sent with the results of the build. Created binary pkgs will be in `/usr/sandbox/usr/pkgsrc/packages` (wherever that points/mounts to/from).

6.3.7. Building a partial set of packages

In addition to building a complete set of all packages in pkgsrc, the `pkgsrc/mk/bulk/build` script may be used to build a subset of the packages contained in pkgsrc. By setting `SPECIFIC_PKGS` in `/etc/mk.conf`, the variables

- SITE_SPECIFIC_PKGS
- HOST_SPECIFIC_PKGS
- GROUP_SPECIFIC_PKGS
- USER_SPECIFIC_PKGS

will define the set of packages which should be built. The bulk build code will also include any packages which are needed as dependencies for the explicitly listed packages.

One use of this is to do a bulk build with `SPECIFIC_PKGS` in a chroot sandbox periodically to have a complete set of the binary packages needed for your site available without the overhead of building extra packages that are not needed.

6.3.8. Uploading results of a bulk build

This section describes how pkgsrc developers can upload binary pkgs built by bulk builds to ftp.NetBSD.org.

If you would like to automatically create checksum files for the binary packages you intend to upload, remember to set `MKSUMS=yes` in your `mk/bulk/build.conf`.

If you would like to PGP sign the checksum files (highly recommended!), remember to set `SIGN_AS=username@NetBSD.org` in your `mk/bulk/build.conf`. This will prompt you for your GPG password to sign the files before uploading everything.

Then, make sure that you have `RSYNC_DST` set properly in your `mk/bulk/build.conf` file, i.e. adjust it to something like one of the following:

```
RSYNC_DST=ftp.NetBSD.org:/pub/NetBSD/packages/pkgsrc-200xQy/NetBSD-a.b.c/arch/upload
```

Please use appropriate values for "pkgsrc-200xQy", "NetBSD-a.b.c" and "arch" here. If your login on ftp.NetBSD.org is different from your local login, write your login directly into the variable, e.g. my local account is "feyrer", but for my login "hubertf", I use:

```
RSYNC_DST=hubertf@ftp.NetBSD.org:/pub/NetBSD/packages/pkgsrc-200xQy/NetBSD-a.b.c/arch/up
```

A separate `upload` directory is used here to allow "closing" the directory during upload. To do so, run the following command on ftp.NetBSD.org next:

```
nbftp% mkdir -p -m 750 /pub/NetBSD/packages/pkgsrc-200xQy/NetBSD-a.b.c/arch/upload
```

Please note that `/pub/NetBSD/packages` is only appropriate for packages for the NetBSD operating system. Binary packages for other operating systems should go into `/pub/pkgsrc`.

Before uploading the binary pkgs, ssh authentication needs to be set up. This example shows how to set up temporary keys for the root account *inside the sandbox* (assuming that no keys should be present there usually):

```
# chroot /usr/sandbox
chroot-# rm $HOME/.ssh/id-dsa*
chroot-# ssh-keygen -t dsa
chroot-# cat $HOME/.ssh/id-dsa.pub
```

Now take the output of `id-dsa.pub` and append it to your `~/.ssh/authorized_keys` file on ftp.NetBSD.org. You can remove the key after the upload is done!

Next, test if your ssh connection really works:

```
chroot-# ssh ftp.NetBSD.org date
```

Use "-l yourNetBSDlogin" here as appropriate!

Now after all this works, you can exit the sandbox and start the upload:

```
chroot-# exit
# cd /usr/sandbox/usr/pkgsrc
# sh mk/bulk/do-sandbox-upload
```

The upload process may take quite some time. Use ls(1) or du(1) on the FTP server to monitor progress of the upload. The upload script will take care of not uploading restricted packages and putting vulnerable packages into the `vulnerable` subdirectory.

After the upload has ended, first thing is to revoke ssh access:

```
nbftp% vi ~/.ssh/authorized_keys
Gdd:x!
```

Use whatever is needed to remove the key you've entered before! Last, move the uploaded packages out of the `upload` directory to have them accessible to everyone:

```
nbftp% cd /pub/NetBSD/packages/pkgsrc-200xQy/NetBSD-a.b.c/arch
nbftp% mv upload/* .
nbftp% rmdir upload
nbftp% chmod 755 .
```

6.4. 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.

6.4.1. 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 7.
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.

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

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

- pkgsrc-bugs (http://www.NetBSD.org/MailingLists/index.html#pkgsrc-bugs): A list where problem reports related to pkgsrc are sent and discussed.
- 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): A list where all commit messages to pkgsrc are sent.
- tech-pkg (http://www.NetBSD.org/MailingLists/index.html#tech-pkg): A general discussion list for all things related to pkgsrc.

To subscribe, do:

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

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

7.2. Where's the pkgviews documentation?

Pkgviews is tightly integrated with buildlink. You can find a pkgviews User's guide in `pkgsrc/mk/buildlink3/PKGVIEWS_UG`.

7.3. Utilities for package management (pkgtools)

The `pkgsrc/pkgtools` directory 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.
- `pkgtools/xpkgwedge`: Put X11 packages someplace else (enabled by default).
- `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/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, lintpkgsrc(1) 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/rpm2pkg`, `pkgtools/url2pkg`: Aids in converting to pkgsrc.
- `pkgtools/gensolpkg`: Convert pkgsrc to a Solaris package.

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.

7.4. 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 overriden by the "--prefix" flag provided by the script, as well as some others that allow finer tuning of the tree layout.

7.5. 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 `/etc/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 default ftp(1) by changing the `FETCH_CMD` variable. Don't forget to set `FETCH_RESUME_ARGS` and `FETCH_OUTPUT_ARGS` if you are not using default values.

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

```
FETCH_CMD=              wget
FETCH_BEFORE_ARGS=      --passive-ftp
FETCH_RESUME_ARGS=      -c
FETCH_OUTPUT_ARGS=      -O
```

7.6. How can I install/use XFree86 from pkgsrc?

If you want to use XFree86 from pkgsrc instead of your system's own X11 (`/usr/X11R6`, `/usr/openwin`, ...), you will have to add the following line into `/etc/mk.conf`:

```
X11_TYPE=XFree86
```

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

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

```
X11_TYPE=xorg
```

Note: The DragonFly operating system defaults to using this X.org X11 implementation from pkgsrc.

7.8. 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/
```

7.9. 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 `/etc/mk.conf` file: `PASSIVE_FETCH=1`.

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

7.10. 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/NetBSD/packages/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 lukemftp) 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
```

7.11. 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 /etc/mk.conf.

7.12. 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**).

7.13. 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 /etc/mk.conf:

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

7.14. 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 /etc/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.

7.15. 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, install the security/audit-packages package. It has two components:

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

   ftp://ftp.NetBSD.org/pub/NetBSD/packages/distfiles/pkg-vulnerabilities

2. **audit-packages**, an easy way to audit the current machine, checking each vulnerability which is known. 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 the security/audit-packages package is strongly recommended! After "audit-packages" is installed, please read the package's message, which you can get by running **pkg_info -D audit-packages**.

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.

II. The pkgsrc developer’s guide

Chapter 8.
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.

8.1. `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.

- `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 only need to provide it if it differs from `DISTNAME`. Usually it is the 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` is a list of URLs where the distribution files can be downloaded. Each URL must end with a slash.

  The `MASTER_SITES` may make use of the following predefined sites:

  ```
  ${MASTER_SITE_APACHE}
  ${MASTER_SITE_BACKUP}
  ${MASTER_SITE_CYGWIN}
  ```

```
${MASTER_SITE_DEBIAN}
${MASTER_SITE_FREEBSD}
${MASTER_SITE_FREEBSD_LOCAL}
${MASTER_SITE_GNOME}
${MASTER_SITE_GNU}
${MASTER_SITE_GNUSTEP}
${MASTER_SITE_IFARCHIVE}
${MASTER_SITE_MOZILLA}
${MASTER_SITE_OPENOFFICE}
${MASTER_SITE_PERL_CPAN}
${MASTER_SITE_R_CRAN}
${MASTER_SITE_SOURCEFORGE}
${MASTER_SITE_SUNSITE}
${MASTER_SITE_SUSE}
${MASTER_SITE_TEX_CTAN}
${MASTER_SITE_XCONTRIB}
${MASTER_SITE_XEMACS}
```

If one of these predefined sites is chosen, 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_SITE_GNU:=subdirectory/name/}
${MASTER_SITE_SOURCEFORGE:=project_name/}
```

Note the trailing slash after the subdirectory name.

If the package has multiple `DISTFILES` or multiple `PATCHFILES` from different sites, set `SITES_foo` to a list of URIs where file “foo” may be found. “foo” includes the suffix, e.g.:

```
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/
```

- `DISTFILES`: Name(s) of archive file(s) containing distribution. The default is `${DISTNAME}${EXTRACT_SUFX}`. Should only be set if you have more than one distfile.

  Note that the normal default setting of `DISTFILES` must be made explicit if you want to add to it (rather than replace it), as you usually would.

- `EXTRACT_SUFX`: Suffix of the distribution file, will be appended to `DISTNAME`. Defaults to `.tar.gz`.

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` below) if not found locally.

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 should contact the MAINTAINER before making major changes to the package. When packaging a new program, set MAINTAINER to yourself. If you really can't maintain the package for future updates, set it to <tech-pkg@NetBSD.org>.
- 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 /etc/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; see comment in bsd.pkg.mk.
- Replace /usr/local with "${PREFIX}" in all files (see patches, below).
- If the package installs any info files, see Section 16.5.7.

8.2. 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 8.3).

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 `www/navigator`). 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.

8.3. patches/*

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

The `patch-*` files should be in **diff -bu** format, and apply without a fuzz to avoid problems. (To force patches to apply with fuzz you can set `PATCH_FUZZ_FACTOR=-F2`). Furthermore, do not put changes for more than one file into a single patch file, as this will make future modifications more difficult.

Similar, a file should be patched at most once, not several times by several different patches. If a file needs several patches, they should be combined into one file.

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** 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**.

When you have finished a package, remember to generate the checksums for the patch files by using the **make makepatchsum** command, see Section 8.2.

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 enables removing the patch in future version.

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*.

8.4. 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 10 for more information.

8.5. Optional files

`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.

`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.

`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`.

8.6. `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.

8.7. `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.

Chapter 9.
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 `Makefile`s, including the pitfalls that come along with them.

9.1. `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 backspace 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`.

9.1.1. Naming conventions

- All variable names starting with an underscore are reserved for use by the pkgsrc infrastructure. They shall not be used by package `Makefile`s.
- 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`.

9.2. 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.

9.2.1. 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.

9.2.2. 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.

9.2.3. Passing variables to a shell command

```
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
        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.

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.

9.2.4. 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}""`.

9.2.5. 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 10.
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!).

10.1. RCS ID

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

```
@comment $NetBSD$
```

10.2. 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 14.16 for more information on this target.

10.3. Tweaking output of make print-PLIST

If you have used any of the *-dirs packages, as explained in Section 10.8, 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; }
```

10.4. 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**"

`${PKGLOCALEDIR}`

Packages that install locale files should list them in the PLIST as "${PKGLOCALEDIR}/locale/de/LC_MESSAGES/..." instead of "share/locale/de/LC_MESSAGES/...". This properly handles the fact that different operating systems expect locale files to be either in `share` or `lib` by default.

For a complete list of values which are replaced by default, please look in `bsd.pkg.mk` (and search for *PLIST_SUBST*).

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 8.5):

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

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

10.5. 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.

10.6. 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 order of things is important.

10.7. 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

10.8. Sharing directories between packages

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

Within pkgsrc, you’ll find both approaches. If a directory is shared by a few unrelated packages, it’s often not worth to add an extra package to remove it. Therefore, one simply does:

```
@unexec ${RMDIR} %D/path/to/shared/directory 2>/dev/null || ${TRUE}
```

in the PLISTs of all affected packages, instead of the regular "@dirrm" line.

However, if the directory is shared across many packages, two different solutions are available:

1. If the packages have a common dependency, the directory can be removed in that. For example, see textproc/scrollkeeper, which removes the shared directory share/omf.
2. If the packages using the directory are not related at all (they have no common dependencies), a *-dirs package is used.

From now on, we’ll discuss the second solution. To get an idea of the *-dirs packages available, issue:

```
% cd .../pkgsrc
% ls -d */*-dirs
```

Their use from other packages is very simple. The USE_DIRS variable takes a list of package names (without the “-dirs” part) together with the required version number (always pick the latest one when writing new packages).

For example, if a package installs files under share/applications, it should have the following line in it:

```
USE_DIRS+=      xdg-1.1
```

After regenerating the PLIST using **make print-PLIST**, you should get the right (commented out) lines.

Note that even if your package is using `$X11BASE`, it must not depend on the *-x11-dirs packages. Just specify the name without that part and pkgsrc (in particular, `mk/dirs.mk`) will take care of it.

Chapter 11.
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.

11.1. 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_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 or x11/openmotif.
- ossaudio.buildlink3.mk defines several variables that may be used by packages that use the Open Sound System (OSS) API.
- pgsql.buildlink3.mk will accept either Postgres 7.3 or 7.4, whichever is found installed. 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.

11.2. 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
```

11.2.1. Anatomy of a buildlink3.mk file

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

```
# $NetBSD: buildlink3.mk,v 1.7 2004/03/18 09:12:12 jlam Exp $

BUILDLINK_DEPTH:=       ${BUILDLINK_DEPTH}+
TIFF_BUILDLINK3_MK:=    ${TIFF_BUILDLINK3_MK}+
```

```
.if !empty(BUILDLINK_DEPTH:M+)
BUILDLINK_DEPENDS+=     tiff
.endif

BUILDLINK_PACKAGES:=    ${BUILDLINK_PACKAGES:Ntiff}
BUILDLINK_PACKAGES+=    tiff

.if !empty(TIFF_BUILDLINK3_MK:M+)
BUILDLINK_DEPENDS.tiff+=        tiff>=3.6.1
BUILDLINK_PKGSRCDIR.tiff?=      ../../graphics/tiff
.endif  # TIFF_BUILDLINK3_MK

.include "../../devel/zlib/buildlink3.mk"
.include "../../graphics/jpeg/buildlink3.mk"

BUILDLINK_DEPTH:=       ${BUILDLINK_DEPTH:S/+$//}
```

The header and footer manipulate BUILDLINK_DEPTH, which is common across all buildlink3.mk files and is used to track at what depth we are including buildlink3.mk files.

The first section controls if the dependency on *pkg* is added. BUILDLINK_DEPENDS is the global list of packages for which dependencies are added by buildlink3.

The second section advises pkgsrc that the buildlink3.mk file for *pkg* has been included at some point. BUILDLINK_PACKAGES is the global list of packages for which buildlink3.mk files have been included. It must *always* be appended to within a buildlink3.mk file.

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

- BUILDLINK_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 the last change in the major number of a shared library or that had a major 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 for overwrite packages, BUILDLINK_CONTENTS_FILTER.*pkg* outputs the contents of the include and lib directories in the package +CONTENTS, and for pkgviews packages, it outputs any libtool archives in lib directories.
- BUILDLINK_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"**.

The last section includes 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.

11.2.2. Updating BUILDLINK_DEPENDS.*pkg* in buildlink3.mk files

There are two situations that require increasing the dependency listed in BUILDLINK_DEPENDS.*pkg* after a package update:

1. if the sonames (major number of the library version) of any installed shared libraries change.
2. if the API or interface to the header files change.

In these cases, BUILDLINK_DEPENDS.*pkg* should be adjusted to require at least the new package version. In some cases, the packages that depend on this new version may need their PKGREVISIONs increased and, if they have buildlink3.mk files, their BUILDLINK_DEPENDS.*pkg* adjusted, too. This is needed so that binary packages made using it will require the correct package dependency and not settle for an older one which will not contain the necessary shared libraries.

Please take careful consideration before adjusting BUILDLINK_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. See Section 16.1.4 for more information about dependencies on other packages, including the BUILDLINK_RECOMMENDED and RECOMMENDED definitions.

11.3. 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.

11.3.1. 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_DEPENDS.foo}
.      if !empty(USE_BUILTIN.foo:M[yY][eE][sS])
USE_BUILTIN.foo!=                                                       \
      if ${PKG_ADMIN} pmatch '${_depend_}' ${BUILTIN_PKG.foo}; then     \
              ${ECHO} "yes";                                            \
      else                                                              \
              ${ECHO} "no";                                             \
      fi
.      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_DEPENDS.*pkg*. This is typically done by comparing BUILTIN_PKG.*pkg* against each of the dependencies in BUILDLINK_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*).

11.3.2. 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 12.
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.

12.1. 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 the installation scripts. These scripts can run arbitrary commands, so they have the potential to create and manage files anywhere in the file system. Here is where pkginstall comes into play: 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.

12.1.1. 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.

12.1.2. 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 SUPPORT_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 SUPPORT_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:

  ```
  SUPPORT_FILES_PERMS+= ${PREFIX}/share/somefile ${VARBASE}/somefile ${ROOT_USER} ${R
  ```

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

12.2. 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.

12.2.1. 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/apache2, 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 12.1.1 what this means.

12.2.2. 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).

12.2.3. 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 12.1.2 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.

12.2.4. 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.

12.3. 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 12.1 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.

12.3.1. 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.

12.4. 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, which mimics /etc/passwd:

```
user:group[:[userid][:[descr][:[home][:shell]]]]
```

Only the user and group are required; everything else is optional, but the colons must be in the right places when specifying optional bits. By default, a new user will have home directory /nonexistent, and login shell /sbin/nologin unless they are specified as part of the user element. Note that if the description contains spaces, then spaces should be backslash-escaped, as in:

```
foo:foogrp::The\ Foomister
```

Similarly, groups can be created using the PKG_GROUPS variable, whose syntax is:

```
group[:groupid]
```

As before, only the group name is required; the numeric identifier is optional.

12.5. 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
```

12.5.1. 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.

12.6. 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}/lib/X11/fonts/TTF
```

12.6.1. 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 13.
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.

13.1. 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 `/etc/mk.conf`.

13.2. 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:Q}" ins
.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/buildlink3.mk"
CONFIGURE_ARGS+= --enable-ldap=${BUILDLINK_PREFIX.openldap}
.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 set after PKG_OPTIONS_VAR is used.
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 /etc/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:M option)
```

13.3. 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.

Chapter 14.
The build process

14.1. 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 splitted 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.

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.

14.2. 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 8.3 and Section 16.3.1 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}".
- `X11BASE` is where the actual X11 distribution (from xsrc, etc.) is installed. When looking for *standard* X11 includes (not those installed by a pkg), 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.

14.3. Directories used during the build process

When building a package, a number of directories is 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.

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 extraced 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.

14.4. 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.

14.5. The *fetch* phase

This will check if the file(s) given in the variables DISTFILES and PATCHFILES (as defined in the package's Makefile) are present on the local system in /usr/pkgsrc/distfiles. If they are not present, an attempt will be made to fetch them 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 can be optionally sorted by the user, via setting either MASTER_SORT_AWK or MASTER_SORT_REGEX.

14.6. 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.

14.7. 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/scripts/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/scripts/extract`.

`EXTRACT_USING`

This variable can be set to `pax`, `tar` or an absolute pathname pointing to the command with which tar archives should be extracted.

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.

14.8. 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 8.3 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.

14.9. The *tools* phase

This is covered in Chapter 15.

14.10. The *wrapper* phase

[TODO]

14.11. 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 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.)

14.12. 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 ${MAKEFILE} ${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, MAKEFILE 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 MAKEFILE is “Makefile”, and BUILD_TARGET defaults to “all”.

14.13. The *test* phase

[TODO]

14.14. 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 ${MAKEFILE} ${BUILD_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.

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. If this variable is set, `NO_MTREE`="yes" is assumed, which means that the package claims to create all needed directories itself before installing files to it. Therefore this variable should only be set in `Makefile`s that are under control of the package's author.

14.15. The *package* phase

[TODO]

14.16. 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.

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.

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 `/etc/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**. e.g. **make update UPDATE_TARGET=package**

`NOCLEAN`

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

`REINSTALL`

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 /etc/mk.conf to alter the behaviour of **make clean-update**:

CLEAR_DIRLIST

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

info

This target invokes pkg_info(1) for the current package. You can use this to check which version of a package is installed.

readme

This target generates a README.html file, which can be viewed using a browser such as www/mozilla 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.

readme-all

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

cdrom-readme

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

show-distfiles

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

show-downlevel

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

show-pkgsrc-dir

This target shows the directory in the pkgsrc hierarchy from which the package can be built and installed. This may not be the same directory as the one from which the package was installed. This target is intended to be used by people who may wish to upgrade many packages on a single host, and can be invoked from the top-level pkgsrc Makefile by using the "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 `/etc/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 10.3 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 Section 6.3.1). 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 15.
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**.

15.1. 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.

15.2. Tools needed by packages

In the following examples, the :pkgsrc means to use the pkgsrc version and not the native version for a build dependency. And the :run means that it is used for a run-time dependencies also (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+= mktemp:pkgsrc
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.

15.3. 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
```

Chapter 16.
Making your package work

16.1. General operation

16.1.1. How to pull in variables from /etc/mk.conf

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

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

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

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

If you wish to set the CFLAGS variable in /etc/mk.conf, please make sure to use:

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

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

16.1.2. Where to install documentation

Documentation should be installed into ${PREFIX}/share/doc/${PKGBASE} or ${PREFIX}/share/doc/${PKGNAME} (the latter includes the version number of the package).

16.1.3. Restricted packages

Some licenses restrict how software may be re-distributed. In order to satisfy 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.

- `NO_BIN_ON_CDROM`

  Binaries may not be placed on CD-ROM. Set this variable to `${RESTRICTED}` whenever a binary package may not be included on a CD-ROM.

- `NO_BIN_ON_FTP`

  Binaries may not be placed on an FTP server. Set this variable to `${RESTRICTED}` whenever a binary package may not not be made available on the Internet.

- `NO_SRC_ON_CDROM`

  Distfiles may not be placed on CD-ROM. Set this variable to `${RESTRICTED}` if re-distribution of the source code or other distfile(s) is not allowed on CD-ROMs.

- `NO_SRC_ON_FTP`

  Distfiles may not be placed on FTP. Set this variable to `${RESTRICTED}` if re-distribution of the source code or other distfile(s) via the Internet is not allowed.

Please note that the use of `NO_PACKAGE`, `IGNORE`, `NO_CDROM`, or other generic make variables to denote restrictions is deprecated, because they unconditionally prevent users from generating binary packages!

16.1.4. 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 11 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 or run, and if that package has a `buildlink3.mk` file available, use it:

   ```
   .include "../../graphics/jpeg/buildlink3.mk"
   ```

2. If your package needs to use another package to build itself and there is no `buildlink3.mk` file available, use the `BUILD_DEPENDS` definition:

   ```
   BUILD_DEPENDS+= autoconf-2.13:../../devel/autoconf
   ```

3. If your package needs a library with which to link and again there is no `buildlink3.mk` file available, this is specified using the `DEPENDS` definition. An example of this is the `print/lyx` package, which uses the xpm library, version 3.4j to build:

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

   You can also use wildcards in package dependences:

   ```
   DEPENDS+=       xpm-[0-9]*:../../graphics/xpm
   ```

   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+=       tiff>=3.5.4:../../graphics/tiff
   ```

   This means that the package will build against version 3.5.4 of the tiff library or newer. Such a dependency may be warranted if, for example, the API of the library has changed with version 3.5.4 and a package would not compile against an earlier version of tiff.

   Please note that such dependencies should only be updated if a package requires a newer pre-requisite, but not to denote recommendations such as security updates or ABI changes that do not prevent a package from building correctly. Such recommendations can be expressed using `RECOMMENDED`:

   ```
   RECOMMENDED+=   tiff>=3.6.1:../../graphics/tiff
   ```

   In addition to the above `DEPENDS` line, this denotes that while a package will build against tiff>=3.5.4, at least version 3.6.1 is recommended. `RECOMMENDED` entries will be turned into dependencies unless explicitly ignored (in which case a warning will be printed).

   To ignore these dependency recommendations and just use the required `DEPENDS`, set `IGNORE_RECOMMENDED=YES`. This may make it easier and faster to update packages built using pkgsrc, since older compatible dependencies can continue to be used. This is useful for people who watch their rebuilds very carefully; it is not very good as a general-purpose hammer. If you use it, you need to be mindful of possible ABI changes, including those from the underlying OS.

   Packages that are built with recommendations ignored may not be uploaded to ftp.NetBSD.org by developers and should not be used across different systems that may have different versions of binary packages installed.

   For security fixes, please update the package vulnerabilities file as well as setting `RECOMMENDED`, see Section 16.1.8 for more information.

4. 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
   ```

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

If your package needs files from another package to build, see the first part of the “do-configure” target `print/ghostscript5` package (it relies on the jpeg sources being present in source form during the build):

```
if [ ! -e ${_PKGSRCDIR}/graphics/jpeg/${WRKDIR:T}/jpeg-6b ]; then    \
    cd ${_PKGSRCDIR}/../../graphics/jpeg && ${MAKE} extract; \
fi
```

If you build any other packages that way, please make sure the working files are deleted too when this package’s working files are cleaned up. The easiest way to do so is by adding a pre-clean target:

```
pre-clean:
        cd ${_PKGSRCDIR}/../../graphics/jpeg && ${MAKE} clean
```

Please also note the `BUILD_USES_MSGFMT` and `BUILD_USES_GETTEXT_M4` definitions, which are provided as convenience definitions. The former works out whether msgfmt(1) is part of the base system, and, if it isn’t, installs the `devel/gettext` package. The latter adds a build dependency on either an installed version of an older gettext package, or if it isn’t, installs the `devel/gettext-m4` package.

16.1.5. 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 like another package in our pkgsrc tree.

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

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

```
CONFLICTS=      Xaw-Xpm-[0-9]*
```

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

```
CONFLICTS=      Xaw3d-[0-9]*
```

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

16.1.6. 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 `NOT_FOR_PLATFORM`. If the package builds and runs on a small handful of platforms, 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.

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.

16.1.7. 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.

16.1.8. 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, use **make upload** in the same directory 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). In addition, if a `buildlink3.mk` file exists for an affected package, a corresponding BUILDLINK_RECOMMENDED.*pkg* entry should be added or updated in it.

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.

16.1.9. 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 file/MACHINE_ARCH/compiler combination, and documenting it in `pkgsrc/doc/HACKS`. See that file for a number of examples!

16.1.10. 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 pkg tools. e.g.

```
DISTNAME=       foo-17.42
PKGREVISION=    9
```

will result in a PKGNAME of “foo-17.42nb9”.

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
```

16.1.11. 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. There are some particular details you should pay attention to while working on pkgsrc.

16.1.11.1. ${INSTALL}, ${INSTALL_DATA_DIR}, ...

The BSD-compatible **install** supplied with some operating systems will not perform more than one operation at a time. As such, you should call "${INSTALL}", etc. like this:

```
${INSTALL_DATA_DIR} ${PREFIX}/dir1
${INSTALL_DATA_DIR} ${PREFIX}/dir2
```

16.2. Possible downloading issues

16.2.1. 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 macro which displays a message explaining the situation. `_FETCH_MESSAGE` must be executable shell commands, not just a message. (Generally, it executes `${ECHO}`). As of this writing, the following packages use this: `cad/simian`, `devel/ipv6socket`, `emulators/vmware-module`, `fonts/acroread-jpnfont`, `multimedia/realplayer`, `sysutils/storage-manager`, `www/ap-aolserver`, `www/openacs`. Try to be consistent with them.

16.2.2. 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. Then, the correct way to work around this is to set `DIST_SUBDIR` to a unique directory name, usually based on `PKGNAME_NOREV`. 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`. Do not forget regenerating the `distinfo` file after that, since it contains the `DIST_SUBDIR` path in the filenames. 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.

16.3. Configuration gotchas

16.3.1. 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 pkg 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_DATA} ${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).

16.3.2. 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.

16.3.3. 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:
        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.

16.4. Building the package

16.4.1. CPP defines

Sometimes you need to compile different code depending on the target platform. The C preprocessor has a set of predefined macros that can be queried by using `#ifdef FOO` or `#if defined(FOO)`. Among these macros are usually ones that describe the target CPU and operating system. Depending of which of the macros are defined, you can write code that uses features unique to a specific platform. Generally you should rather use the GNU autotools (automake, autoconf, etc.) to check for specific features (like the existence of a header file, a function or a library), but sometimes this is not possible or desired.

In that case you can use the predefined macros below to configure your code to the platform it runs on. 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.

16.4.1.1. CPP defines for operating systems

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 use the following defines.

```
FreeBSD     __FreeBSD__
DragonFly __DragonFly__
Interix __INTERIX
Linux       linux, __linux, __linux__
NetBSD      __NetBSD__
OpenBSD     __OpenBSD__
Solaris     sun, __sun
```

16.4.1.2. CPP defines for CPUs

```
i386        i386, __i386, __i386__
MIPS        __mips
SPARC       sparc, __sparc
```

16.4.1.3. CPP defines for compilers

```
GCC         __GNUC__ (major version), __GNUC_MINOR__
SunPro __SUNPRO_C (0x570 for version 5.7)
```

16.4.2. Examples of CPP defines for some platforms

The list of the CPP identification macros for hardware and operating system may depend on the compiler that is used. The following list contains some examples that may help you to choose the right ones. 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.

GCC 3.3.3 + SuSE Linux 9.1 + i386

> __ELF__, __gnu_linux__, __i386, __i386__, __linux, __linux__, __unix, __unix__, i386, linux, unix.

GCC 2.95 + NetBSD 1.6.2 + i386

> __ELF__, __NetBSD__, __i386, __i386__, i386.

GCC 3.3.3 + NetBSD 2.0 + i386

__ELF__, __NetBSD__, __i386, __i386__, i386.

GCC 4 + Solaris 8 + SPARC

__ELF__, __sparc, __sparc__, __sun, __sun__, __SVR4, __svr4__, __unix, __unix__, sparc, sun, unix.

SunPro 5.7 + Solaris 8 + SPARC

__SVR4, __sparc, __sun, __unix, sparc, sun, unix.

16.4.3. Getting a list of CPP defines

If your system uses the GNU C Compiler, you can get a list of symbols that are defined by default, e.g. to identify the platform, with the following command:

```
gcc -E -dM - < /dev/null
```

On other systems you may get the list by using the system's syscall trace utility (ktrace, truss, strace) to have a look which arguments are passed to the actual compiler.

16.5. Package specific actions

16.5.1. User interaction

Occasionally, packages require interaction from the user, and this can be in a number of ways:

- help in fetching the distfiles
- 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
```

16.5.2. Handling licenses

A package may be covered by a license which the user has or has not agreed to accept. For these cases, pkgsrc contains a mechanism to note that a package is covered by a particular license, and the package

cannot be built unless the user has accepted the license. (Installation of binary packages are not currently subject to this mechanism.) Packages with licenses that are either Open Source according to the Open Source Initiative or Free according to the Free Software Foundation will not be marked with a license tag. Packages with licenses that have not been determined to meet either definition will be marked with a license tag referring to the license. This will prevent building unless pkgsrc is informed that the license is acceptable, and enables displaying the license.

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.). However, the above definition of licenses for which tags are not needed implies that packages with redistribution restrictions should have tags.

Denoting that a package is covered by 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 accepted:

```
% 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 it is considered appropriate, the line printed above can be added to `/etc/mk.conf` to indicate acceptance of the particular license:

```
ACCEPTABLE_LICENSES+=xv-license
```

When adding a package with a new license, the license text should be added to `pkgsrc/licenses` for displaying. A list of known licenses can be seen in this directory as well as by looking at the list of (commented out) `ACCEPTABLE_LICENSES` variable settings in `pkgsrc/mk/defaults/mk.conf`.

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 denote acceptance of the license for a single package without accepting the same license text for another package. In particular, this can be inappropriate when e.g. one accepts a particular license to indicate to pkgsrc that a fee has been paid.

16.5.3. Installing score 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). The following variables, documented in more detail in `mk/defaults/mk.conf`, control this behaviour: `SETGIDGAME`, `GAMEDATAMODE`, `GAMEGRP`, `GAMEMODE`, `GAMEOWN`.

Note that per default, setgid installation of games is disabled; setting SETGIDGAME=YES will set all the other variables accordingly.

A package should therefor never hard code file ownership or access permissions but rely on INSTALL_GAME and INSTALL_GAME_DATA to set these correctly.

16.5.4. Packages containing perl scripts

If your package contains interpreted perl scripts, 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.

16.5.5. 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
```

16.5.6. 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 paths to packlist files, e.g.:

```
PERL5_PACKLIST= ${PERL5_SITEARCH}/auto/Pg/.packlist
```

The variables PERL5_SITELIB, PERL5_SITEARCH, and PERL5_ARCHLIB represent the three locations in which perl5 modules may be installed, and may be used by perl5 packages that don't have a packlist. These three variables are also substituted for in the PLIST.

16.5.7. Packages installing info files

Some packages install info files or use the "makeinfo" or "install-info" commands. Each of the info files:

- is considered to be installed in the directory ${PREFIX}/${INFO_DIR},
- is registered in the Info directory file ${PREFIX}/${INFO_DIR}/dir,
- and must be listed as a filename in the INFO_FILES variable in the package Makefile.

INFO_DIR defaults to “info” and can be overridden in the package Makefile. 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.

A package which needs the “makeinfo” command at build time must define the variable USE_MAKEINFO 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 USE_MAKEINFO and TEXINFO_REQD either run the appropriate **makeinfo** command or exit on error.

16.5.8. Packages installing man pages

Many packages install manual pages. The man pages are installed under ${PREFIX}/${PKGMANDIR} which is /usr/pkg/man by default. PKGMANDIR defaults to “man”. For example, you can set PKGMANDIR to “share/man” to have man pages install under /usr/pkg/share/man/ by default.

> **Note:** The support for a custom PKGMANDIR is not 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.

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 10.5 for information on installation of compressed manual pages.

16.5.9. Packages installing GConf2 data files

If a package installs .schemas or .entries files, used by GConf2, you need to take some extra steps to make sure they get registered in the database:

1. Include ../../devel/GConf2/schemas.mk instead of its buildlink3.mk file. This takes care of rebuilding the GConf2 database at installation and deinstallation time, and tells the package

where to install GConf2 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 7.14 for more information.
4. Define the `GCONF2_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 `GCONF2_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.

16.5.10. Packages installing scrollkeeper data files

If a package installs `.omf` files, used by scrollkeeper, you need to take some extra steps to make sure they get registered in the database:

1. Include `../../textproc/scrollkeeper/omf.mk` instead of its `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 scrollkeeper.

16.5.11. 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.

16.5.12. 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.

16.5.13. 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.

16.5.14. 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.

16.5.15. Packages using intltool

If a package uses intltool during its build, include the `../../textproc/intltool/buildlink3.mk` file, 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.

16.5.16. 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 `/etc/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.

16.5.17. 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 teTeX-texmf, packages should install files into `PKG_LOCALTEXMFPREFIX`, not `PKG_TEXMFPREFIX`.

1. Include `../../print/teTeX/module.mk` instead of `../../mk/tex.buildlink3.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 `PKG_LOCALTEXMFPREFIX`, set `TEXMFDIRS` 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**, set `TEX_FONTMAPS` 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 teTeX-bin package.

16.6. Feedback to the author

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

Support the idea of free software!

Chapter 17.
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=1` in `/etc/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 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 8.1.
- 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 blub
  ```

- Repeat the above **make print-PLIST** command, which shouldn't find anything now:

  ```
  # make print-PLIST
  ```

- Reinstall the binary package:

  ```
  # pkgadd .../blub.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 18.

Chapter 18.
Submitting and Committing

18.1. Submitting your packages

You have to separate between binary and “normal” (source) packages here:

- precompiled 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 Section 6.3.8.

- packages

  First, check that your package is complete, compiles and runs well; see Chapter 17 and the rest of this document. Next, generate an uuencoded gzipped tar(1) archive, preferably with all files in a single directory. Finally, **send-pr** with category “pkg”, a synopsis which includes the package name and version number, a short description of your package (contents of the COMMENT variable or DESCR file are OK) and attach the archive to your PR.

  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.

18.2. General notes when adding, updating, or removing packages

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

There is a make target that helps in creating proper `CHANGES` 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` 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 `/etc/mk.conf` if your local login name is not the same as your NetBSD login name. Don’t forget to commit the changes to `pkgsrc/doc/CHANGES`!

18.3. Committing: Importing a package into CVS

This section is only of interest for pkgsrc developers with write access to the pkgsrc repository. Please remember that cvs imports files relative to the current working directory, and that the pathname that you give the **cvs import** command is so that it knows where to place the files in the repository. Newly created packages should be imported with a vendor tag of "TNF" and a release tag of "pkgsrc-base", e.g:

```
$ cd .../pkgsrc/category/pkgname
$ cvs import pkgsrc/category/pkgname TNF pkgsrc-base
```

Remember to move the directory from which you imported out of the way, or cvs will complain the next time you "cvs update" your source tree. Also don't forget to add the new package to the category's `Makefile`.

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.

For new packages, "cvs import" is preferred to "cvs add" because the former gets everything with a single command, and provides a consistent tag.

18.4. 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.

18.5. Moving a package in pkgsrc

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. **cvs import** the modified package in the new place.
5. Check if any package depends on it:

   ```
   % cd /usr/pkgsrc
   % grep /package */*/Makefile* */*/buildlink*
   ```

6. Fix paths in packages from step 5 to point to new location.
7. **cvs rm (-f)** the package at the old location.
8. Remove from `oldcategory/Makefile`.
9. Add to `newcategory/Makefile`.
10. Commit the changed and removed files:

    ```
    % cvs commit oldcategory/package oldcategory/Makefile newcategory/Makefile
    ```

    (and any packages from step 5, of course).

Chapter 19.
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.

19.1. 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:

`bootstrap/mods/mk/`*`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/platform/MyOS.pkg.dist`

This file contains a list of directories, together with their permission bits and ownership. These directories will be created automatically with every package that does not explicitly set `NO_MTREE`. There have been some discussions about whether this file is needed at all, but with no result.

`mk/platform/MyOS.x11.dist`

Just copy one of the pre-existing x11.dist files to your *`MyOS`*`.x11.dist`.

`mk/tools/bootstrap.mk`

On some operating systems, the tools that are provided with the base system are not good enough for pkgsrc. For example, there are many versions of sed(1) that have a narrow limit on the line length they can process. Therefore pkgsrc brings its own tools, which can be enabled here.

`mk/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`.

19.2. 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

```
# $NetBSD$
#

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

MAINTAINER=     thorpej@NetBSD.org
HOMEPAGE=       http://www.gnu.org/software/bison/bison.html
COMMENT=        GNU yacc clone

GNU_CONFIGURE=  yes
INFO_FILES=     bison.info

.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 $NetBSD$
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 8) 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.co
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.a
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://orp
Successfully retrieved file.
```

Generate the checksum of the distfile into `distinfo`:

```
# make makesum
```

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_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DXPFILE=\"/usr/pkg/share/bison.simple\"  -DXPFILE1=\"/usr/pkg/share/bison.hairy\"
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc -c -DSTDC_HEADERS=1 -DHAVE_STRING_H=1 -DHAVE_STDLIB_H=1 -DHAVE_MEMORY_H=1 -DHAVE_ALLOC
cc  -g -o bison LR0.o allocate.o closure.o conflicts.o derives.o files.o         getargs
./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
/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/uni
=> [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 arhive 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
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\"  f
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.
Layout of the FTP server's package archive

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

```
/pub/NetBSD/packages/
    distfiles/

    # Unpacked pkgsrc trees
    pkgsrc-current -> /pub/NetBSD/NetBSD-current/pkgsrc
    pkgsrc-2003Q4 -> N/A
    pkgsrc-2004Q1/pkgsrc

    # pkgsrc archives
    pkgsrc-current.tar.gz -> ../NetBSD-current/tar_files/pkgsrc.tar.gz
    pkgsrc-2003Q4.tar.gz -> N/A
    pkgsrc-2004Q1.tar.gz -> N/A

    # Per pkgsrc-release/OS-release/arch package archives
    pkgsrc-2003Q4/
        NetBSD-1.6.2/
            i386/
                All/
                archivers/
                    foo -> ../All/foo
                ...
    pkgsrc-2004Q1/
        NetBSD-1.6.2/
            i386/
                All/
                ...
        NetBSD-2.0/
            i386/
                All/
                ...
        SunOS-5.9/
            sparc/
                All/
                ...
            x86/
                All/
                ...

    # Per os-release package archive convenience links
```

```
NetBSD-1.6.2 -> 1.6.2
1.6.2/
    i386 -> ../pkgsrc-2004Q1/NetBSD-1.6.2/i386
    m68k/
        All/
        archivers/
            foo -> ../All/foo
        ...
    amiga -> m68k
    atari -> m68k
    ...

2.0 -> NetBSD-2.0       # backward compat, historic
NetBSD-2.0/
    i386 -> ../pkgsrc-2004Q1/NetBSD-2.0/i386
SunOS-5.9/
    sparc -> ../pkgsrc-2004Q1/SunOS-5.9/sparc
    x86 -> ../pkgsrc-2004Q1/SunOS-5.9/x86
```

To create:

1. Run bulk build, see Section 6.3
2. Upload /usr/pkgsrc/packages to

```
ftp://ftp.NetBSD.org/pub/NetBSD/packages/\
    pkgsrc-2004Q4/\             # pkgsrc-branch
    ‘uname -s‘-‘uname -r‘/\     # OS & version
    ‘uname -p‘                  # architecture
```

3. If necessary, create a symlink **ln -s ‘uname -m‘ ‘uname -p‘** (amiga -> m68k, ...)

Appendix D. *Editing guidelines for the pkgsrc guide*

This section contains information on editing the pkgsrc guide itself.

D.1. 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/Documentation/pkgsrc/`: the documentation on the NetBSD website will be built from pkgsrc and kept up to date on the web server itself. This means you *must* make sure that your changes haven't broken the build!
- `http://www.NetBSD.org/Documentation/pkgsrc/pkgsrc.pdf`: PDF version of the pkgsrc guide.
- `http://www.NetBSD.org/Documentation/pkgsrc/pkgsrc.ps`: PostScript version of the pkgsrc guide.

D.2. Procedure

The procedure to edit the pkgsrc guide is:

- Make sure you have the packages needed to re-generate the pkgsrc guide (and other XML-based NetBSD documentation) installed. These are "netbsd-doc" for creating the ASCII and HTML versions, and "netbsd-doc-print" for the PostScript and PDF versions. You will need both packages installed, to make sure documentation is consistent across all formats. The packages can be found in `pkgsrc/meta-pkgs/netbsd-doc` and `pkgsrc/meta-pkgs/netbsd-doc-print`.
- Edit the XML file(s) in `pkgsrc/doc/guide/files`.
- Run **make extract && make do-lint** in `pkgsrc/doc/guide` to check the XML syntax, and fix it if needed.
- Run **make** in `pkgsrc/doc/guide` to build the HTML and ASCII version.
- If all is well, run **make install-doc** to put the generated files into `pkgsrc/doc`.

- **cvs commit pkgsrc/doc/guide/files**
- **cvs commit -m re-generate pkgsrc/doc/pkgsrc.{html,txt}**
- Until the webserver on www.NetBSD.org is really updated automatically to pick up changes to the pkgsrc guide automatically, also run **make install-htdocs HTDOCSDIR=../../../htdocs** (or similar, adjust `HTDOCSDIR`!).
- **cvs commit htdocs/Documentation/pkgsrc**