

Package Manager Specification

Stephen P. Bennett
spb@exherbo.org

Ciaran McCreesh
ciaran.mccreesh@gmail.com

September 27, 2008

Contents

1	Introduction	9
1.1	Aims and Motivation	9
1.2	Rationale	9
1.3	Conventions	9
1.4	E APIs	9
1.4.1	Reserved E APIs	10
2	Names and Versions	11
2.1	Restrictions upon Names	11
2.1.1	Category Names	11
2.1.2	Package Names	11
2.1.3	Slot Names	11
2.1.4	USE Flag Names	11
2.1.5	Repository Names	12
2.1.6	Keyword Names	12
2.2	Version Specifications	12
2.3	Version Comparison	12
2.4	Uniqueness of versions	13
3	Tree Layout	14
3.1	Top Level	14
3.2	Category Directories	14
3.3	Package Directories	15
3.4	The Profiles Directory	15
3.4.1	The profiles.desc file	16
3.4.2	The thirdpartymirrors file	16
3.4.3	use.desc and related files	16
3.4.4	The updates directory	17
3.5	The Licenses Directory	17
3.6	The Eclass Directory	17
3.7	The Metadata Directory	17
3.7.1	The metadata cache	17
4	Profiles	19
4.1	General principles	19
4.2	Files that make up a profile	19
4.2.1	The parent file	19
4.2.2	deprecated	19
4.2.3	make.defaults	20
4.2.4	virtuals	20
4.2.5	use.defaults	20
4.2.6	Simple line-based files	20
4.2.7	packages	20
4.2.8	packages.build	20

4.2.9	package.mask	21
4.2.10	package.provided	21
4.2.11	package.use	21
4.2.12	USE masking and forcing	21
4.3	Profile variables	21
4.3.1	Incremental Variables	22
4.3.2	Specific variables and their meanings	22
5	Old-Style Virtual Packages	24
5.1	Dependencies on virtual packages	24
6	Ebuild File Format	25
7	Eclasses	26
7.1	The inherit command	26
7.2	Eclass-defined Metadata Keys	26
7.3	EXPORT_FUNCTIONS	26
8	Ebuild-defined Variables	28
8.1	Metadata invariance	28
8.2	Mandatory Ebuild-defined Variables	28
8.3	Optional Ebuild-defined Variables	29
8.3.1	EAPI	29
8.4	RDEPEND value	29
8.5	Magic Ebuild-defined Variables	30
9	Dependencies	31
9.1	Dependency Classes	31
9.2	Dependency Specification Format	31
9.2.1	All-of Dependency Specifications	32
9.2.2	Use-conditional Dependency Specifications	32
9.2.3	Any-of Dependency Specifications	32
9.2.4	Package Dependency Specifications	32
9.2.5	Restrict	34
9.2.6	SRC_URI	34
10	Ebuild-defined Functions	35
10.1	List of Functions	35
10.1.1	pkg_setup	35
10.1.2	src_unpack	35
10.1.3	src_prepare	36
10.1.4	src_configure	36
10.1.5	src_compile	36
10.1.6	src_test	37
10.1.7	src_install	37
10.1.8	pkg_preinst	37
10.1.9	pkg_postinst	38
10.1.10	pkg_prerm	38
10.1.11	pkg_postrm	38
10.1.12	pkg_config	38
10.1.13	pkg_info	38
10.1.14	pkg_nofetch	38
10.1.15	default_Phase Functions	38
10.2	Call Order	39
11	The Ebuild Environment	41
11.1	Defined Variables	41

11.2	The state of variables between functions	43
11.3	Available commands	43
11.3.1	System commands	43
11.3.2	Commands provided by package dependencies	45
11.3.3	Ebuild-specific Commands	45
11.4	The state of the system between functions	51
12	Merging and Unmerging	53
12.1	Overview	53
12.2	Directories	53
12.2.1	Permissions	53
12.2.2	Empty Directories	53
12.3	Regular Files	54
12.3.1	Permissions	54
12.3.2	Configuration File Protection	54
12.4	Symlinks	54
12.4.1	Rewriting	55
12.5	Hard links	55
12.6	Other Files	55
13	Glossary	56
A	metadata.xml	57
B	Unspecified Items	58
C	Historical Curiosities	59
C.1	If-else use blocks	59
C.2	cvs Versions	59
D	Feature Availability by EAPI	60
E	Differences Between EAPIs	61

List of Algorithms

1	USE masking logic	22
2	USE_EXPAND logic	23
3	econf --libdir logic	47
4	Determining the library directory	48

Listings

7.1	EXPORT_FUNCTIONS example: foo.eclass	27
11.1	Environment state between functions	44
11.2	einstall command	47
C.1	If-else use blocks	59

List of Tables

8.1	E APIs supporting IUSE defaults	29
9.1	E APIs supporting SRC_URI arrows	32
9.2	E APIs supporting SLOT dependencies	33
9.3	E APIs supporting USE dependencies	33
9.4	Exclamation mark strengths for E APIs	33
10.1	E APIs supporting src_prepare	36
10.2	E APIs supporting src_configure	36
10.3	src_compile behaviour for E APIs	37
10.4	E APIs supporting default_phase functions	39
11.1	Defined variables	41
11.2	E APIs supporting domain languages	49
11.3	E APIs supporting the default function	51
D.1	Features in E APIs	60

List of Corrections

Acknowledgements

Thanks to Mike Kelly (package manager provided utilities, section 11.3.3), Danny van Dyk (ebuild functions, section 10), David Leverton (various sections) and Petteri Rätty (environment state, section 11.2) for contributions. Thanks to Christian Faulhammer for fixing some of the more horrible formatting screwups. Thanks also to Mike Frysinger and Brian Haring for proof-reading and suggestions for fixes and/or clarification.

Copyright and Licence

The bulk of this document is © 2007, 2008 Stephen Bennett and Ciaran McCreesh. Contributions are owned by their respective authors, and may have been changed substantially before inclusion.

This document is released under the Creative Commons Attribution-Share Alike 3.0 Licence. The full text of this licence can be found at <http://creativecommons.org/licenses/by-sa/3.0/>.

Reporting Issues

Issues (inaccuracies, wording problems, omissions etc.) in this document should be reported via Gentoo Bugzilla using product *Gentoo Hosted Projects*, component *PMS/EAPI* and the default assignee. There should be one bug per issue, and one issue per bug.

Patches (in `git format-patch` form if possible) may be submitted either via Bugzilla or to the `pms-bugs@gentoo.org` alias. Patches will be reviewed by the PMS team, who will do one of the following:

- Accept and apply the patch.
- Explain why the patch cannot be applied as-is. The patch may then be updated and resubmitted if appropriate.
- Reject the patch outright.
- Take special action merited by the individual circumstances.

When reporting issues, remember that this document is not the appropriate place for pushing through changes to the tree or the package manager, except where those changes are bugs.

If any issue cannot be resolved by the PMS team, it may be escalated to the Gentoo Council.

Chapter 1

Introduction

1.1 Aims and Motivation

This document aims to fully describe the format of an ebuild repository and the ebuilds therein, as well as certain aspects of package manager behaviour required to support such a repository.

This document is *not* designed to be an introduction to ebuild development. Prior knowledge of ebuild creation and an understanding of how the package management system works is assumed; certain less familiar terms are explained in the Glossary in chapter 13.

This document does not specify any user or package manager configuration information.

1.2 Rationale

At present the only definition of what an ebuild can assume about its environment, and the only definition of what is valid in an ebuild, is the source code of the latest Portage release and a general consensus about which features are too new to assume availability. This has several drawbacks: not only is it impossible to change any aspect of Portage behaviour without verifying that nothing in the tree relies upon it, but if a new package manager should appear it becomes impossible to fully support such an ill-defined standard.

This document aims to address both of these concerns by defining almost all aspects of what an ebuild repository looks like, and how an ebuild is allowed to behave. Thus, both Portage and other package managers can change aspects of their behaviour not defined here without worry of incompatibilities with any particular repository.

1.3 Conventions

Text in `teletype` is used for filenames or variable names. *Italic* text is used for terms with a particular technical meaning in places where there may otherwise be ambiguity.

The term *package manager* is used throughout this document in a broad sense. Although some parts of this document are only relevant to fully featured package managers, many items are equally applicable to tools or other applications that interact with ebuilds or ebuild repositories.

1.4 EAPIs

An EAPI can be thought of as a ‘version’ of this specification to which a package conforms. An EAPI value is a string. The following EAPIs are defined by this specification:

0 The ‘original’ base EAPI.

1 EAPI ‘1’ contains a number of extensions to EAPI ‘0’. Except where explicitly noted, it is in all other ways identical to EAPI ‘0’.

2 EAPI ‘2’ contains a number of extensions to EAPI ‘1’. Except where explicitly noted, it is in all other ways identical to EAPI ‘1’.

Except where explicitly noted, everything in this specification applies to all EAPIs.

If a package manager encounters a package version with an unrecognised EAPI, it must not attempt to perform any operations upon it. It could, for example, ignore the package version entirely (although this can lead to user confusion), or it could mark the package version as masked. A package manager must not use any metadata generated from a package with an unrecognised EAPI.

The package manager must not attempt to perform any kind of comparison test other than equality upon EAPIs.

1.4.1 Reserved EAPIs

- EAPIs whose value consists purely of an integer are reserved for future versions of this specification.
- EAPIs whose value starts with the string `paludis-` are reserved for experimental use by the Paludis package manager.
- EAPIs whose value starts with the string `kdebuild-` are reserved for the Gentoo KDE project.

Chapter 2

Names and Versions

2.1 Restrictions upon Names

No name may be empty. Package managers must not impose fixed upper boundaries upon the length of any name. A package manager should indicate or reject any name that is invalid according to these rules.

2.1.1 Category Names

A category name may contain any of the characters [A-Za-z0-9+_.-]. It must not begin with a hyphen or a dot.

Note: A hyphen is *not* required because of the `virtual` category. Usually, however, category names will contain a hyphen.

2.1.2 Package Names

A package name may contain any of the characters [A-Za-z0-9+_-]. It must not begin with a hyphen, and must not end in a hyphen followed by one or more digits.

Note: A package name does not include the category. The term *qualified package name* is used where a `category/package` pair is meant.

2.1.3 Slot Names

A slot name may contain any of the characters [A-Za-z0-9+_.-]. It must not begin with a hyphen or a dot.

2.1.4 USE Flag Names

A USE flag name may contain any of the characters [A-Za-z0-9+@_-]. It must begin with an alphanumeric character. Underscores should be considered reserved for `USE_EXPAND`, as described in section 4.3.2.

Note: The at-sign is required for `LINGUAS`.

2.1.5 Repository Names

A repository name may contain any of the characters `[A-Za-z0-9_-]`. It must not begin with a hyphen.

2.1.6 Keyword Names

A keyword name may contain any of the characters `[A-Za-z0-9_-]`. It must not begin with a hyphen. In contexts where it makes sense to do so, a keyword name may be prefixed by a tilde or a hyphen. In `KEYWORDS`, `-*` is also acceptable as a keyword, to indicate that a package will only work on listed targets.

A tilde prefixed keyword is, by convention, used to indicate a less stable package. It is generally assumed that any user accepting keyword `~foo` will also accept `foo`.

The exact meaning of any keywords value is beyond the scope of this specification.

2.2 Version Specifications

The package manager must not impose fixed limits upon the number of version components. Package managers should indicate or reject any version that is invalid according to these rules.

A version starts with the number part, which is in the form `[0-9]+(\.[0-9]+)*` (a positive integer, followed by zero or more dot-prefixed positive integers).

This may optionally be followed by one of `[a-z]` (a lowercase letter).

This may be followed by zero or more of the suffixes `_alpha`, `_beta`, `_pre`, `_rc` or `_p`, which themselves may be suffixed by an optional integer.

This may optionally be followed by the suffix `-r` followed immediately by an integer (the “revision number”). If this suffix is not present, it is assumed to be `-r0`.

2.3 Version Comparison

Version specifications are compared component by component, moving from left to right.

The first component of the number part is compared using strict integer comparison.

Any subsequent components of the number part are compared as follows:

- If neither component has a leading zero, components are compared using strict integer comparison.
- Otherwise, if a component has a leading zero, any trailing zeroes in that component are stripped (if this makes the component empty, proceed as if it were `0` instead), and the components are compared using a stringwise comparison.

If one number part is a prefix of the other, then the version with the longer number part is greater. Note in particular that `1.0` is less than `1.0.0`.

Letter suffixes are compared alphabetically, with any letter being newer than no letter.

If the letters are equal, suffixes are compared. The ordering is `_alpha` is less than `_beta` is less than `_pre` is less than `_rc` is less than no suffix is less than `_p`. If a suffix string is equal, the associated integer parts are compared using strict integer comparison. A missing integer part is treated as zero.

If at this point the two versions are still equal, the revision number is compared using strict integer comparison as per the previous part. If the revision numbers are equal, so are the two versions.

2.4 Uniqueness of versions

No two packages in a given repository may have the same qualified package name and equal versions. For example, a repository may not contain more than one of `foo-bar/baz-1.0.2`, `foo-bar/baz-1.0.2-r0` and `foo-bar/baz-1.000.2`.

Chapter 3

Tree Layout

This chapter defines the layout on-disk of an ebuild repository. In all cases below where a file or directory is specified, a symlink to a file or directory is also valid. In this case, the package manager must follow the operating system's semantics for symbolic links and must not behave differently from normal.

3.1 Top Level

An ebuild repository shall occupy one directory on disk, with the following subdirectories:

- One directory per category, whose name shall be the name of the category. The layout of these directories shall be as described in section 3.2.
- A `profiles` directory, described in section 3.4.
- A `licenses` directory (optional), described in section 3.5.
- An `eclass` directory (optional), described in section 3.6.
- A `metadata` directory (optional), described in section 3.7.
- Other optional support files and directories (skeleton ebuilds or `ChangeLogs`, for example) may exist but are not covered by this specification. The package manager must ignore any of these files or directories that it does not recognise.

3.2 Category Directories

Each category provided by the repository (see also: the `profiles/categories` file, section 3.4) shall be contained in one directory, whose name shall be that of the category. Each category directory shall contain:

- A `metadata.xml` file, as described in appendix A. Optional.
- Zero or more package directories, one for each package in the category, as described in section 3.3. The name of the package directory shall be the corresponding package name.

Category directories may contain additional files, whose purpose is not covered by this specification. Additional directories that are not for a package may *not* be present, to avoid conflicts with package name directories; an exception is made for filesystem components whose name starts with a dot, which the package manager must ignore, and for any directory named `CVS`.

It is not required that a directory exists for each category provided by the repository. A category directory that does not exist shall be considered equivalent to an empty category (and by extension, a package manager may treat an empty category as a category that does not exist).

3.3 Package Directories

A package directory contains the following:

- Zero or more ebuilds. These are as described in section 6 and others.
- A `metadata.xml` file, as described in appendix A. Optional only for legacy support.
- A `ChangeLog`, in a format determined by the provider of the repository. Optional.
- A `Manifest` file, whose format is described in [1].
- A `files` directory, containing any support files needed by the ebuilds. Optional.

Any ebuild in a package directory must be named `name-ver.suffix`, where:

- `name` is the (unqualified) package name.
- `ver` is the package's version.
- `suffix` is ebuild.

Package managers must ignore any ebuild file that does not match these rules.

A package directory that contains no correctly named ebuilds shall be considered a package with no versions. A package with no versions shall be considered equivalent to a package that does not exist (and by extension, a package manager may treat a package that does not exist as a package with no versions).

A package directory may contain other files or directories, whose purpose is not covered by this specification.

3.4 The Profiles Directory

The profiles directory shall contain zero or more profile directories as described in section 4, as well as the following files and directories. In any line-based file, lines beginning with a `#` character are treated as comments, whilst blank lines are ignored. All contents of this directory, with the exception of `repo_name`, are optional.

If the repository is not intended to be stand-alone, the contents of these files are to be taken from or merged with the master repository as necessary.

Other files not described by this specification may exist, but may not be relied upon. The package manager must ignore any files in this directory that it does not recognise.

arch.list Contains a list, one entry per line, of permissible values for the `ARCH` variable, and hence permissible keywords for packages in this repository.

categories Contains a list, one entry per line, of categories provided by this repository.

info_pkgs Contains a list, one entry per line, of qualified package names. Any package matching one of these is to be listed when a package manager displays a 'system information' listing.

info_vars Contains a list, one entry per line, of profile, configuration, and environment variables which are considered to be of interest. The value of each of these variables may be shown when the package manager displays a 'system information' listing.

package.mask Contains a list, one entry per line, of (EAPI-0) package dependency specifications. Any package version matching one of these is considered to be masked, and will not be installed regardless of profile unless it is unmasked by the user configuration.

profiles.desc Described below in section 3.4.1.

repo_name Contains, on a single line, the name of this repository. The repository name must conform to section 2.1.5.

thirdpartymirrors Described below in section 3.4.2.

use.desc Contains descriptions of valid global USE flags for this repository. The format is described in section 3.4.3.

use.local.desc Contains descriptions of valid local USE flags for this repository, along with the packages to which they apply. The format is as described in section 3.4.3.

desc/ This directory contains files analogous to `use.desc` for the various `USE_EXPAND` variables. Each file in it is named `<varname>.desc`, where `<varname>` is the variable name, in lowercase, whose possible values the file describes. The format of each file is as for `use.desc`, described in section 3.4.3. The `USE_EXPAND` name is *not* included as a prefix here.

updates/ This directory is described in section 3.4.4.

3.4.1 The profiles.desc file

`profiles.desc` is a line-based file, with the standard commenting rules from section 3.4, containing a list of profiles that are valid for use, along with their associated architecture and status. Each line has the format:

```
<keyword> <profile path> <stability>
```

Where:

- `<keyword>` is the default keyword for the profile and the `ARCH` for which the profile is valid.
- `<profile path>` is the (relative) path from the `profiles` directory to the profile in question.
- `<stability>` indicates the stability of the profile. This may be useful for QA tools, which may wish to display warnings with a reduced severity for some profiles. The values `stable` and `dev` are widely used, but repositories may use other values.

Fields are whitespace-delimited.

3.4.2 The thirdpartymirrors file

`thirdpartymirrors` is another simple line-based file, describing the valid mirrors for use with `mirror://` URIs in this repository, and the associated download locations. The format of each line is:

```
<mirror name> <mirror 1> <mirror 2> ... <mirror n>
```

Fields are whitespace-delimited. When parsing a URI of the form `mirror://name/path/filename`, where the `path/` part is optional, the `thirdpartymirrors` file is searched for a line whose first field is `name`. Then the download URIs in the subsequent fields have `path/filename` appended to them to generate the URIs from which a download is attempted.

Each mirror name may appear at most once in a file. Behaviour when a mirror name appears multiple times is undefined. Behaviour when a mirror is defined in terms of another mirror is undefined. A package manager may choose to fetch from all of or a subset of the listed mirrors, and may use an order other than the one described.

The mirror with the name equal to the repository's name (and if the repository has a master, the master's name) may be consulted for all downloads.

3.4.3 use.desc and related files

`use.desc` contains descriptions of every valid global USE flag for this repository. It is a line-based file with the standard rules for comments and blank lines. The format of each line is:

```
<flagname> - <description>
```

`use.local.desc` contains descriptions of every valid local USE flag—those that apply only to a small number of packages, or that have different meanings for different packages. Its format is:

```
<category/package>:<flagname> - <description>
```

Flags must be listed once for each package to which they apply, or if a flag is listed in both `use.desc` and `use.local.desc`, it must be listed once for each package for which its meaning differs from that described in `use.desc`.

3.4.4 The updates directory

The `updates` directory is used to inform the package manager that a package has moved categories, names, or that a version has changed SLOT. It contains one file per quarter year, named `[1-4]Q-[YYYY]` for the first to fourth quarter of a given year, for example `1Q-2004` or `3Q-2006`. The format of each file is again line-based, with each line having one of the following formats:

```
move <qpn1> <qpn2>
slotmove <spec> <slot1> <slot2>
```

The first form, where `qpn1` and `qpn2` are *qualified package names*, instructs the package manager that the package `qpn1` has changed name, category, or both, and is now called `qpn2`.

The second form instructs the package manager that any currently installed package version matching package dependency specification `spec` whose SLOT is set to `slot1` should have it updated to `slot2`.

Any name that has appeared as the origin of a move must not be reused in the future. Any slot that has appeared as the origin of a slot move may not be used by packages matching the `spec` of that slot move in the future.

3.5 The Licenses Directory

The `licenses` directory shall contain copies of the licenses used by packages in the repository. Each file will be named according to the name used in the `LICENSE` variable as described in section 8.2, and will contain the complete text of the license in human-readable form. Plain text format is strongly preferred but not required.

3.6 The Eclass Directory

The `eclass` directory shall contain copies of the eclasses provided by this repository. The format of these files is described in section 7. It may also contain, in their own directory, support files needed by these eclasses.

3.7 The Metadata Directory

The `metadata` directory contains various repository-level metadata that is not contained in `profiles/`. All contents are optional. In this standard only the `cache` subdirectory is described; other contents are optional but may include security advisories, DTD files for the various XML files used in the repository, and repository timestamps.

3.7.1 The metadata cache

The `metadata/cache` directory contains a cached form of all important ebuild metadata variables. The `cache` directory, if it exists, contains directories whose names are the same as categories in the repository—not all categories and packages must be contained in it. Each subdirectory may optionally contain one file per package version in that category, named `<package>-<version>`, in the following format:

Each cache file contains the textual values of various metadata keys, one per line, in the following order. Other lines may be present following these; their meanings are not defined here.

1. Build-time dependencies (DEPEND)

2. Run-time dependencies (RDEPEND)
3. Slot (SLOT)
4. Source tarball URIs (SRC_URI)
5. RESTRICT
6. Package homepage (HOMEPAGE)
7. Package license (LICENSE)
8. Package description (DESCRIPTION)
9. Package keywords (KEYWORDS)
10. Inherited eclasses (INHERITED)
11. Use flags that this package respects (IUSE)
12. No longer used; this line is to be ignored.
13. Post dependencies (PDEPEND)
14. Old-style virtuals provided by this package (PROVIDE)
15. The ebuild API version to which this package conforms (EAPI)
16. Blank lines to pad the file to 22 lines long

Future EAPIs may define new variables, remove existing variables, change the line number or format used for a particular variable, add or reduce the total length of the file and so on. Any future EAPI that uses this cache format will continue to place the EAPI value on line 15 if such a concept makes sense for that EAPI, and will place a value that is clearly not a supported EAPI on line 15 if it does not.

Chapter 4

Profiles

4.1 General principles

Generally, a profile defines information specific to a certain ‘type’ of system—it lies somewhere between repository-level defaults and user configuration in that the information it contains is not necessarily applicable to all machines, but is sufficiently general that it should not be left to the user to configure it. Some parts of the profile can be overridden by user configuration, some only by another profile.

The format of a profile is relatively simple. Each profile is a directory containing any number of the files described in this chapter, and possibly inheriting another profile. The files themselves follow a few basic conventions as regards inheritance and format; these are described in the next section. It may also contain any number of subdirectories containing other profiles.

4.2 Files that make up a profile

4.2.1 The parent file

A profile may contain a `parent` file. Each line must contain a relative path to another profile which will be considered as one of this profile’s parents. Any settings from the parent are inherited by this profile, and can be overridden by it. Precise rules for how settings are combined with the parent profile vary between files, and are described below. Parents are handled depth first, left to right, with duplicate parent paths being sourced for every time they are encountered.

It is illegal for a profile’s parent tree to contain cycles. Package manager behaviour upon encountering a cycle is undefined.

This file must not contain comments, blank lines or make use of line continuations.

4.2.2 deprecated

If a profile contains a file named `deprecated`, it is treated as such. The first line of this file should contain the path from the `profiles` directory of the repository to a valid profile that is the recommended upgrade path from this profile. The remainder of the file can contain any text, which may be displayed to users using this profile by the package manager. This file is not inherited—profiles which inherit from a deprecated profile are *not* deprecated.

This file must not contain comments or make use of line continuations.

4.2.3 `make.defaults`

`make.defaults` is used to define defaults for various environment and configuration variables. This file is unusual in that it is not combined at a file level with the parent—instead, each variable is combined or overridden individually as described in section 4.3.

The file itself is a line-based key-value format. Each line contains a single `VAR="value"` entry, where the value must be double quoted. A variable name must start with one of `a-zA-Z` and may contain `a-zA-Z0-9_`. Additional syntax, which is a small subset of bash syntax, is allowed as follows:

- Variables to the right of the equals sign in the form `${foo}` or `$foo` are recognised and expanded from variables previously set in this or earlier `make.defaults` files.
- One logical line may be continued over multiple physical lines by escaping the newline with a backslash. This is also permitted inside quoted strings.
- Backslashes, except for line continuations, are not allowed.

4.2.4 `virtuals`

The `virtuals` file defines default providers for “old-style” virtual packages. It is a simple line-based file, with each line containing two whitespace-delimited tokens. The first is a virtual package name (for example, `virtual/alsa`) and the second is a qualified package name. Blank lines and those beginning with a `#` character are ignored. When attempting to resolve a virtual name to a concrete package, the specification defined in the active profile’s `virtuals` list should be used if no provider is already installed.

The `virtuals` file is inherited in the simplest manner: all entries from the parent profile are loaded, then entries from the current profile. If a virtual package name appears in both, the entry in the parent profile is discarded.

4.2.5 `use.defaults`

The `use.defaults` file is used to implement ‘autouse’—switching USE flags on or off depending upon which packages are installed. It is considered deprecated, and is not used by default by any current package manager. It is mentioned here for completeness only, and its format is not discussed.

4.2.6 Simple line-based files

These files are a simple one-item-per-line list, which is inherited in the following manner: the parent profile’s list is taken, and the current profile’s list appended. If any line begins with a hyphen, then any lines previous to it whose contents are equal to the remainder of that line are removed from the list. Once again, blank lines and those beginning with a `#` are discarded.

4.2.7 `packages`

The `packages` file is used to define the ‘system set’ for this profile. After the above rules for inheritance and comments are applied, its lines must take one of two forms: a package dependency specification prefixed by `*` denotes that the atom forms part of the system set. A package dependency specification on its own may also appear for legacy reasons, but should be ignored when calculating the system set.

4.2.8 `packages.build`

The `packages.build` file is used by Gentoo’s Catalyst tool to generate stage1 tarballs, and has no relevance to the operation of a package manager. It is thus outside the scope of this document, but is mentioned here for completeness.

4.2.9 package.mask

`package.mask` is used to prevent packages from being installed on a given profile. Each line contains one package dependency specification; anything matching this specification will not be installed unless unmasked by the user's configuration.

Note that the `-spec` syntax can be used to remove a mask in a parent profile, but not necessarily a global mask (from `profiles/package.mask`, section 3.4).

Note: Portage currently treats `profiles/package.mask` as being on the leftmost branch of the inherit tree when it comes to `-lines`. This behaviour may not be relied upon.

4.2.10 package.provided

`package.provided` is used to tell the package manager that a certain package version should be considered to be provided by the system regardless of whether it is actually installed. Because it has severe adverse effects on USE-based and slot-based dependencies, its use is strongly deprecated and package manager support must be regarded as purely optional.

4.2.11 package.use

The `package.use` file may be used by the package manager to override the default USE flags specified by `make.defaults` on a per package basis. The format is to have a package dependency specification, and then a space delimited list of USE flags to enable. A USE flag in the form of `-flag` indicates that the package should have the USE flag disabled. The package dependency specification is limited to the forms defined by EAPI 0.

4.2.12 USE masking and forcing

This section covers the four files `use.mask`, `use.force`, `package.use.mask` and `package.use.force`. They are described together because they interact in a non-trivial manner.

Simply speaking, `use.mask` and `use.force` are used to say that a given USE flag must never or always, respectively, be enabled when using this profile. `package.use.mask` and `package.use.force` do the same thing on a per-package, or per-version, basis. The precise manner in which they interact is less simple, and is best described in terms of the algorithm used to determine whether a flag is masked for a given package version. This is described in Algorithm 1.

The logic for `use.force` and `package.use.force` is identical. If a flag is both masked and forced, the mask is considered to take precedence.

USE_EXPAND values may be forced or masked by using `expand_name_value`.

A package manager may treat ARCH values that are not the current architecture as being masked.

4.3 Profile variables

This section documents variables that have special meaning, or special behaviour, when defined in a profile's `make.defaults` file.

Algorithm 1 USE masking logic

```
1: let masked = false
2: for each profile in the inheritance tree, depth first do
3:   if use.mask contains flag then
4:     let masked = true
5:   else if use.mask contains -flag then
6:     let masked = false
7:   end if
8:   for each line in package.use.mask, in order, for which the spec matches package do
9:     if line contains flag then
10:      let masked = true
11:     else if line contains -flag then
12:      let masked = false
13:     end if
14:   end for
15: end for
```

4.3.1 Incremental Variables

Incremental variables must stack between parent and child profiles in the following manner: Beginning with the highest parent profile, tokenise the variable's value based on whitespace and concatenate the lists. Then, for any token *T* beginning with a hyphen, remove it and any previous tokens whose value is equal to *T* with the hyphen removed, or, if *T* is equal to *-**, remove all previous values. Note that because of this treatment, the order of tokens in the final result is arbitrary, not necessarily related to the order of tokens in any given profile. The following variables must be treated in this fashion:

- USE
- USE_EXPAND
- USE_EXPAND_HIDDEN
- CONFIG_PROTECT
- CONFIG_PROTECT_MASK
- Any variable whose name is listed in USE_EXPAND

Other variables, except where they affect only package-manager-specific functionality (such as Portage's FEATURES variable), must not be treated incrementally—later definitions shall completely override those in parent profiles.

4.3.2 Specific variables and their meanings

The following variables have specific meanings when set in profiles.

ARCH The system's architecture. Must be a value listed in `profiles/arch.list`; see section 3.4 for more information. Must be equal to the primary `KEYWORD` for this profile.

CONFIG_PROTECT, CONFIG_PROTECT_MASK Contain whitespace-delimited lists used to control the configuration file protection. Described more fully in chapter 12.3.2.

USE Defines the list of default USE flags for this profile. Flags may be added or removed by the user's configuration. `USE_EXPAND` values must not be specified in this way.

USE_EXPAND Defines a list of variables which are to be treated incrementally and whose contents are to be expanded into the USE variable as passed to `ebuild`. Expansion is done as per Algorithm 2. So, for example, if `USE_EXPAND` contains 'ALSA_CARDS', and the `ALSA_CARDS` variable contains 'foo', 'alsa_cards_foo' will be appended to USE.

USE_EXPAND_HIDDEN Contains a (possibly empty) subset of names from `USE_EXPAND`. The package manager may use this set as a hint to avoid displaying uninteresting or unhelpful information to an end user.

Algorithm 2 USE_EXPAND logic

```
for each variable  $V$  listed in USE_EXPAND do  
  for each token  $T$  in  $V$  do  
    append  $v_T$  to USE, where  $v$  is the lowercase of  $V$   
  end for  
end for
```

Any other variables set in `make.defaults` must be passed on into the ebuild environment as-is, and are not required to be interpreted by the package manager.

Chapter 5

Old-Style Virtual Packages

Old-style virtuals are pseudo-packages—they can be depended upon or installed, but do not exist in the ebuild repository. An old-style virtual requires several things in the repository: at least one ebuild must list the virtual in its `PROVIDE` variable, and there must be at least one entry in a profiles `virtuals` file listing the default provider for each profile—see sections 8.3 and 4.2.4 for specifics on these two. Old-style virtuals require special handling as regards dependencies; this is described below.

All old-style virtuals must use the category `virtual`. Not all packages using the `virtual` category may be assumed to be old style virtuals.

Note: A *new-style* virtual is simply an ebuild which install no files and use its dependency strings to select providers. By convention, and to ease migration, these are also placed in the `virtual` category.

5.1 Dependencies on virtual packages

When a dependency on a virtual package is encountered, it must be resolved into a real package before it can be satisfied. There are two factors that affect this process: whether a package providing the virtual is installed, and the `virtuals` file in the active profile (section 4.2.4). If a package is already installed which satisfies the virtual requirement (via `PROVIDE`), then it should be used to satisfy the dependency. Otherwise, the profiles `virtuals` file (section 4.2.4) should be consulted to choose an appropriate provider.

Dependencies on old style virtuals must not use any kind of version restriction.

Blocks on provided virtuals have special behaviour documented in section 9.2.4.

Chapter 6

Ebuild File Format

The ebuild file format is in its basic form a subset of the format of a bash script. The interpreter is assumed to be GNU bash, version 3.0 or later. The file encoding must be UTF-8 with Unix-style newlines. When sourced, the ebuild must define certain variables and functions (see sections 8 and 10 for specific information), and must not call any external programs, write anything to standard output or standard error, or modify the state of the system in any way.

Chapter 7

Eclasses

Eclasses serve to store common code that is used by more than one ebuild, which greatly aids maintainability and reduces the tree size. However, due to metadata cache issues, care must be taken in their use. In format they are similar to an ebuild, and indeed are sourced as part of any ebuild using them. The interpreter is therefore the same, and the same requirements for being parseable hold.

Eclasses must be located in the `eclass` directory in the top level of the repository—see section 3.6. Each eclass is a single file named `<name>.eclass`, where `<name>` is the name of this eclass, used by `inherit` and `EXPORT_FUNCTIONS` among other places.

7.1 The `inherit` command

An ebuild wishing to make use of an eclass does so by using the `inherit` command in global scope. This will cause the eclass to be sourced as part of the ebuild—any function or variable definitions in the eclass will appear as part of the ebuild, with exceptions for certain metadata variables, as described below.

The `inherit` command takes one or more parameters, which must be the names of eclasses (excluding the `.eclass` suffix and the path). For each parameter, in order, the named eclass is sourced.

Eclasses may end up being sourced multiple times.

The `inherit` command must also ensure that:

- The `ECLASS` variable is set to the name of the current eclass, when sourcing that eclass.
- Once all inheriting has been done, the `INHERITED` metadata variable contains the name of every eclass used, separated by whitespace.

7.2 Eclass-defined Metadata Keys

The `IUSE`, `DEPEND`, `RDEPEND` and `PDEPEND` variables are handled specially when set by an eclass. They must be accumulated across eclasses, appending the value set by each eclass to the resulting value after the previous one is loaded. Then the eclass-defined value is appended to that defined by the ebuild. In the case of `RDEPEND`, this is done after the implicit `RDEPEND` rules in section 8.4 are applied.

7.3 `EXPORT_FUNCTIONS`

There is one command available in the eclass environment that is neither available nor meaningful in ebuilds—`EXPORT_FUNCTIONS`. This can be used to alias ebuild phase functions from the eclass so that an ebuild inherits

Listing 7.1: EXPORT_FUNCTIONS example: foo.eclass

```
foo_src_compile()
{
    econf --enable-gerbil \
        $(use_enable fnord) \
        || die "econf failed"
    emake gerbil || die "Couldn't make a gerbil"
    emake || die "emake failed"
}

EXPORT_FUNCTIONS src_compile
```

a default definition whilst retaining the ability to override and call the eclass-defined version from it. The use of it is best illustrated by an example; this is given in listing 7.1 and is a snippet from a hypothetical `foo.eclass`.

This example defines an eclass `src_compile` function and uses `EXPORT_FUNCTIONS` to alias it. Then any ebuild that inherits `foo.eclass` will have a default `src_compile` defined, but should the author wish to override it he can access the function in `foo.eclass` by calling `foo_src_compile`.

`EXPORT_FUNCTIONS` must only be used on ebuild phase functions. The function that is aliased must be named `eclassname_phasefunctionname`, where `eclassname` is the name of the eclass.

`EXPORT_FUNCTIONS` must be used at most once per eclass.

Chapter 8

Ebuild-defined Variables

Note: This section describes variables that may or must be defined by ebuilds. For variables that are passed from the package manager to the ebuild, see section 11.1.

8.1 Metadata invariance

All ebuild-defined variables discussed in this chapter must be defined independently of any system, profile or tree dependent data, and must not vary depending upon the ebuild phase. In particular, ebuild metadata can and will be generated on a different system from that upon which the ebuild will be used, and the ebuild must generate identical metadata every time it is used.

Globally defined ebuild variables without a special meaning must similarly not rely upon variable data.

8.2 Mandatory Ebuild-defined Variables

All ebuilds must define at least the following variables:

DESCRIPTION A short human-readable description of the package’s purpose. May be defined by an eclass. Must not be empty.

HOMEPAGE The URI or URIs for a package’s homepage, including protocols. May be defined by an eclass. See section 9 for full syntax.

IUSE The `USE` flags used by the ebuild. Historically, `USE_EXPAND` values and `ARCH` were not included; package managers should support this for backwards compatibility reasons. Ebuilds must list only flags used by the ebuild itself. Any eclass that works with `USE` flags must also set `IUSE`, listing only the variables used by that eclass. The package manager is responsible for merging these values.

In EAPIs shown in table 8.1 as supporting `IUSE` defaults, any use flag name in `IUSE` may be prefixed by at most one of a plus or a minus sign. If such a prefix is present, the package manager may use it as a suggestion as to the default value of the use flag if no other configuration overrides it.

KEYWORDS A whitespace separated list of keywords for the ebuild. Each token must be a valid keyword name, as per section 2.1.6. May include `-*`, which indicates that the package will only work on explicitly listed archs. May include `-arch`, which indicates that the package will not work on the specified arch. May be empty, which indicates uncertain functionality on any architecture. May be defined in an eclass.

LICENSE The package’s license. Each text token must correspond to a tree “licenses/” entry (see section 3.5). See section 9 for full syntax. May be defined by an eclass.

Table 8.1: EAPIs supporting IUSE defaults

EAPI	Supports IUSE defaults?
0	No
1	Yes
2	Yes

SLOT The package’s slot. Must be a valid slot name, as per section 2.1.3. May be defined by an eclass. Must not be empty.

SRC_URI A list of source URIs for the package. Valid protocols are `http://`, `https://`, `ftp://` and `mirror://` (see section 3.4.2 for mirror behaviour). Fetch restricted packages may include URL parts consisting of just a filename. See section 9 for full syntax.

If any of these variables are undefined, or if any of these variables are set to invalid values, the package manager’s behaviour is undefined; ideally, an error in one ebuild should not prevent operations upon other ebuilds or packages.

8.3 Optional Ebuild-defined Variables

Ebuilds may define any of the following variables:

DEPEND See section 9.

EAPI The EAPI. See below for defaults.

PDEPEND See section 9.

PROVIDE Zero or more qualified package names of any *old style* virtuals provided by this package. See section 9 for full syntax.

RDEPEND See section 9. **RDEPEND** has special behaviour for its value if unset and when used with an eclass. See section 8.4 for details.

RESTRICT Zero or more behaviour restrictions for this package. See section 9.2.5 for value meanings and section 9 for full syntax.

S The path to the temporary build directory, used by `src_compile`, `src_install` etc. Defaults to `${WORKDIR}/${P}`.

8.3.1 EAPI

An empty **EAPI** value is equal to 0. Ebuilds must not assume that they will get a particular one of these two values if they are expecting one of these two values.

The package manager must either pre-set the **EAPI** variable to 0 or ensure that it is unset before sourcing the ebuild for metadata generation. When using the ebuild for other purposes, the package manager must either pre-set **EAPI** to the value specified by the ebuild’s metadata or ensure that it is unset.

If any of these variables are set to invalid values, the package manager’s behaviour is undefined; ideally, an error in one ebuild should not prevent operations upon other ebuilds or packages.

8.4 RDEPEND value

If **RDEPEND** is unset (but not if it is set to an empty string) in an ebuild, the package manager must set its value to be equal to the value of **DEPEND**.

When dealing with eclasses, only values set in the ebuild itself are considered for this behaviour; any `DEPEND` or `RDEPEND` set in an eclass does not change the implicit `RDEPEND=$DEPEND` for the ebuild portion, and any `DEPEND` value set in an eclass does not get added to `RDEPEND`.

8.5 Magic Ebuild-defined Variables

The following variables must be defined by `inherit`, and may be considered to be part of the ebuild's metadata:

ECLASS The current eclass, or unset if there is no current eclass. This is handled magically by `inherit` and must not be modified manually.

INHERITED List of inherited eclass names. Again, this is handled magically by `inherit`.

Note: Thus, by extension of section 8.1, `inherit` may not be used conditionally, except upon constant conditions.

Chapter 9

Dependencies

9.1 Dependency Classes

There are three classes of dependencies supported by ebuilds:

- Build dependencies (DEPEND). These must be installed and usable before the ebuild is installed.
- Runtime dependencies (RDEPEND). These must be installed and usable before the ebuild is treated as usable.
- Post dependencies (PDEPEND). These must be installed at some point.

In addition, SRC_URI, HOMEPAGE, PROVIDE, RESTRICT and LICENSE use dependency-style specifications to specify their values.

9.2 Dependency Specification Format

The following elements are recognised in at least one class of specification. All elements must be surrounded on both sides by whitespace, except at the start and end of the string.

- A package dependency specification. Permitted in DEPEND, RDEPEND, PDEPEND.
- A simple qualified package name. Permitted in PROVIDE (and inside DEPEND etc via the previous item).
- A URI, in the form `proto://host/path`. Permitted in SRC_URI and HOMEPAGE. In EAPIs listed in table 9.1 as supporting SRC_URI arrows, may optionally be followed by whitespace, then `->`, then whitespace, then a simple filename when in SRC_URI. For SRC_URI behaviour, see section 9.2.6.
- A flat filename. Permitted in SRC_URI.
- A license name (e.g. GPL-2). Permitted in LICENSE.
- A simple string. Permitted in RESTRICT.
- An all-of group, which consists of an open parenthesis, followed by whitespace, followed by zero or more of (a dependency item of any kind followed by whitespace), followed by a close parenthesis. More formally: `all-of ::= '(' whitespace (item whitespace)* ')'`. Permitted in all specification style variables.
- An any-of group, which consists of the string `||`, followed by whitespace, followed by an open parenthesis, followed by whitespace, followed by zero or more of (a dependency item of any kind followed by whitespace), followed by a close parenthesis. More formally: `any-of ::= '||' whitespace '(' whitespace (item whitespace)* ')'`. Permitted in DEPEND, RDEPEND, PDEPEND, LICENSE.
- A use-conditional group, which consists of an optional exclamation mark, followed by a use flag name, followed by a question mark, followed by whitespace, followed by an open parenthesis, followed by whitespace, followed by zero or more of (a dependency item of any kind followed by whitespace), followed by a close parenthesis. More formally: `use-conditional ::= '!?' flag-name '?' whitespace '(' whitespace (item whitespace)* ')'`. Permitted in all specification style variables.

Table 9.1: EAPIs supporting SRC_URI arrows

EAPI	Supports SRC_URI arrows?
0	No
1	No
2	Yes

In particular, note that whitespace is not optional.

9.2.1 All-of Dependency Specifications

In an all-of group, all of the child elements must be matched.

9.2.2 Use-conditional Dependency Specifications

In a use-conditional group, if the associated use flag is enabled (or disabled if it has an exclamation mark prefix), all of the child elements must be matched.

9.2.3 Any-of Dependency Specifications

Any use-conditional group that is an immediate child of an any-of group, if not enabled (disabled for an exclamation mark prefixed use flag name), is not considered a member of the any-of group for match purposes.

In an any-of group, at least one immediate child element must be matched. A blocker is considered to be matched if its associated package dependency specification is not matched.

An empty any-of group counts as being matched.

9.2.4 Package Dependency Specifications

A package dependency can be in one of the following base formats. A package manager must warn or error on non-compliant input.

- A simple `category/package` name.
- An operator, as described in section 9.2.4, followed immediately by `category/package`, followed by a hyphen, followed by a version specification.

In EAPIs shown in table 9.2 as supporting SLOT dependencies, either of the above formats may additionally be suffixed by a `:slot` restriction, as described in section 9.2.4. A package manager must warn or error if slot dependencies are used with an EAPI not supporting SLOT dependencies.

In EAPIs shown in table 9.3 as supporting 2-style USE dependencies, a specification may additionally be suffixed by at most one 2-style `[use]` restriction, as described in section 9.2.4. A package manager must warn or error if this feature is used with an EAPI not supporting use dependencies.

Note: Order is important. The slot restriction must come before use dependencies.

Operators

The following operators are available:

< Strictly less than the specified version.

Table 9.2: EAPIs supporting SLOT dependencies

EAPI	Supports SLOT dependencies?
0	No
1	Yes
2	Yes

Table 9.3: EAPIs supporting USE dependencies

EAPI	Supports USE dependencies?
0	No
1	No
2	2-style

<= Less than or equal to the specified version.

= Exactly equal to the specified version. Special exception: if the version specified has an asterisk immediately following it, a string prefix comparison is used instead. When an asterisk is used, the specification must remain valid if the asterisk were removed. (An asterisk used with any other operator is illegal.)

~ Equal to the specified version, except the revision part of the matching package may be greater than the revision part of the specified version (-r0 is assumed if no revision is explicitly stated).

>= Greater than or equal to the specified version.

> Strictly greater than the specified version.

Block Operator

If the specification is prefixed with one or two exclamation marks, the named dependency is a block rather than a requirement—that is to say, the specified package must not be installed, with the following exceptions:

- Blocks on a package provided exclusively by the ebuild do not count.
- Blocks on the ebuild itself do not count.

There are two strengths of block: weak and strong. A weak block may be ignored by the package manager, so long as any blocked package will be uninstalled later on. A strong block must not be ignored. The mapping from one or two exclamation marks to strength is described in table 9.4.

Slot Dependencies

A named slot dependency consists of a colon followed by a slot name. A specification with a named slot dependency matches only if the slot of the matched package is equal to the slot specified. If the slot of the package to match cannot be determined (e.g. because it is not a supported EAPI), the match is treated as unsuccessful.

Table 9.4: Exclamation mark strengths for EAPIs

EAPI	!	!!
0	Unspecified	Forbidden
1	Unspecified	Forbidden
2	Weak	Strong

2-Style Use Dependencies

A 2-style use dependency consists of one of the following:

[opt] The flag must be enabled.

[opt=] The flag must be enabled if the flag is enabled for the package with the dependency, or disabled otherwise.

[!opt=] The flag must be disabled if the flag is enabled for the package with the dependency, or enabled otherwise.

[opt?] The flag must be enabled if the flag is enabled for the package with the dependency.

[!opt?] The flag must be disabled if the use flag is disabled for the package with the dependency.

[-opt] The flag must be disabled.

Multiple requirements may be combined using commas, e.g. `[first, -second, third?]`.

When multiple requirements are specified, all must match for a successful match.

It is an error for a use dependency to be applied to an ebuild which does not have the flag in question in `IUSE`, or for an ebuild to use a conditional use dependency when that ebuild does not have the flag in `IUSE`.

9.2.5 Restrict

The following tokens are permitted inside `RESTRICT`:

mirror The package's `SRC_URI` entries may not be mirrored, and mirrors should not be checked when fetching.

fetch The package's `SRC_URI` entries may not be downloaded automatically. If entries are not available, `pkg_nofetch` is called.

strip No stripping of debug symbols from files to be installed may be performed.

userpriv The package manager may not drop root privileges when building the package.

test The `src_test` phase must not be run.

sandbox The `sandbox` tool must not be used when building the package.

Package managers may recognise other tokens, but ebuilds may not rely upon them being supported.

9.2.6 SRC_URI

All filename components that are enabled (i.e. not inside a use-conditional block that is not matched) in `SRC_URI` must be available in the `DISTDIR` directory. In addition, these components are used to make the `A` and `AA` variables.

If a component contains a full URI with protocol, that download location must be used. Package managers may also consult mirrors for their files.

The special `mirror://` protocol must be supported. See section 3.4.2 for mirror details.

If a simple filename rather than a full URI is provided, the package manager can only use mirrors to download the file.

The `RESTRICT` metadata key can be used to impose additional restrictions upon downloading—see section 9.2.5 for details.

Chapter 10

Ebuild-defined Functions

10.1 List of Functions

The following is a list of functions that an ebuild, or eclass, may define, and which will be called by the package manager as part of the build and/or install process. In all cases the package manager must provide a default implementation of these functions; unless otherwise stated this must be a no-op. Most functions must assume only that they have write access to the package's working directory (the `WORKDIR` environment variable; see section 11.1), and the temporary directory `T`; exceptions are noted below. All functions may assume that they have read access to all system libraries, binaries and configuration files that are accessible to normal users.

Some functions may assume that their initial working directory is set to a particular location; these are noted below. If no initial working directory is mandated, it may be set to anything and the ebuild must not rely upon a particular location for it.

The environment for functions run outside of the build sequence (that is, `pkg_config`, `pkg_info`, `pkg_prerm` and `pkg_postrm`) must be the environment used for the build of the package, not the current configuration.

Ebuilds must not call nor assume the existence of any phase functions.

10.1.1 `pkg_setup`

The `pkg_setup` function sets up the ebuild's environment for all following functions, before the build process starts. Further, it checks whether any necessary prerequisites not covered by the package manager, e.g. that certain kernel configuration options are fulfilled.

`pkg_setup` must be run with full filesystem permissions, including the ability to add new users and/or groups to the system.

10.1.2 `src_unpack`

The `src_unpack` function extracts all of the package's sources, applies patches and sets up the package's build system for further use.

The initial working directory must be `WORKDIR`, and the default implementation used when the ebuild lacks the `src_unpack` function shall behave as:

```
src_unpack() {
    if [[ -n ${A} ]]; then
        unpack ${A}
    fi
}
```

Table 10.1: EAPIs supporting `src_prepare`

EAPI	Supports <code>src_prepare</code> ?
0	No
1	No
2	Yes

Table 10.2: EAPIs supporting `src_configure`

EAPI	Supports <code>src_configure</code> ?
0	No
1	No
2	Yes

10.1.3 `src_prepare`

The `src_prepare` function is only called for EAPIs listed in table 10.1 as supporting it.

The `src_prepare` function can be used for post-unpack source preparation. The default implementation does nothing.

The initial working directory must be `S` if that exists, falling back to `WORKDIR` otherwise.

10.1.4 `src_configure`

The `src_configure` function is only called for EAPIs listed in table 10.2 as supporting it.

The initial working directory must be `S` if that exists, falling back to `WORKDIR` otherwise.

The `src_configure` function configures the package's build environment. The default implementation used when the ebuild lacks the `src_configure` function shall behave as:

```
src_configure() {
    if [[ -x ${ECONF_SOURCE:-.}/configure ]]; then
        econf
    fi
}
```

10.1.5 `src_compile`

The `src_compile` function configures the package's build environment in EAPIs lacking `src_configure`, and builds the package in all EAPIs.

The initial working directory must be `S` if that exists, falling back to `WORKDIR` otherwise.

For EAPIs listed in table 10.3 as using format 0, the default implementation used when the ebuild lacks the `src_compile` function shall behave as:

```
src_compile() {
    if [[ -x ./configure ]]; then
        econf
    fi
    if [[ -f Makefile ]] || [[ -f GNUmakefile ]] || [[ -f makefile ]]; then
        emake || die "emake failed"
    fi
}
```

Table 10.3: `src_compile` behaviour for EAPIs

EAPI	Format
0	0
1	1
2	2

For EAPIs listed in table 10.3 as using format 1, the default implementation used when the ebuild lacks the `src_compile` function shall behave as:

```
src_compile() {
    if [[ -x ${ECONF_SOURCE:-.}/configure ]]; then
        econf
    fi
    if [[ -f Makefile ]] || [[ -f GNUmakefile ]] || [[ -f makefile ]]; then
        emake || die "emake failed"
    fi
}
```

For EAPIs listed in table 10.3 as using format 2, the default implementation used when the ebuild lacks the `src_compile` function shall behave as:

```
src_compile() {
    if [[ -f Makefile ]] || [[ -f GNUmakefile ]] || [[ -f makefile ]]; then
        emake || die "emake failed"
    fi
}
```

10.1.6 `src_test`

The `src_test` function runs unit tests for the newly built but not yet installed package as provided.

The initial working directory must be `S` if that exists, falling back to `WORKDIR` otherwise. The default implementation used when the ebuild lacks the `src_test` function must, if tests are enabled, run `make check` if and only if such a target is available, or if not run `make test`, if and only if such a target is available. In both cases, if `make` returns non-zero the build must be aborted.

The `src_test` function may be disabled by `RESTRICT`. See section 9.2.5.

10.1.7 `src_install`

The `src_install` function installs the package's content to a directory specified in `$D`.

The initial working directory must be `S` if that exists, falling back to `WORKDIR` otherwise. The default implementation used when the ebuild lacks the `src_install` function is a no-op.

10.1.8 `pkg_preinst`

The `pkg_preinst` function performs any special tasks that are required immediately before merging the package to the live filesystem. It must not write outside of the directories specified by the `ROOT` and `D` environment variables.

`pkg_preinst` must be run with full access to all files and directories below that specified by the `ROOT` and `D` environment variables.

10.1.9 pkg_postinst

The `pkg_postinst` function performs any special tasks that are required immediately after merging the package to the live filesystem. It must not write outside of the directory specified in the `ROOT` environment variable.

`pkg_postinst`, like, `pkg_preinst`, must be run with full access to all files and directories below that specified by the `ROOT` environment variable.

10.1.10 pkg_prerm

The `pkg_prerm` function performs any special tasks that are required immediately before unmerging the package from the live filesystem. It must not write outside of the directory specified by the `ROOT` environment variable.

`pkg_prerm` must be run with full access to all files and directories below that specified by the `ROOT` environment variable.

10.1.11 pkg_postrm

The `pkg_postrm` function performs any special tasks that are required immediately after unmerging the package from the live filesystem. It must not write outside of the directory specified by the `ROOT` environment variable.

`pkg_postrm` must be run with full access to all files and directories below that specified by the `ROOT` environment variable.

10.1.12 pkg_config

The `pkg_config` function performs any custom steps required to configure a package after it has been fully installed. It is the only ebuild function which may be interactive and prompt for user input.

`pkg_config` must be run with full access to all files and directories inside of `ROOT`.

10.1.13 pkg_info

The `pkg_info` function may be called by the package manager when displaying information about an installed package.

`pkg_info` must not write to the filesystem.

10.1.14 pkg_nofetch

The `pkg_nofetch` function is run when the fetch phase of an fetch-restricted ebuild is run, and the relevant source files are not available. It should direct the user to download all relevant source files from their respective locations, with notes concerning licensing if applicable.

`pkg_nofetch` must require no write access to any part of the filesystem.

10.1.15 default_Phase Functions

In EAPIs listed in table 10.4 as supporting `default_` phase functions, a function named `default_${EBUILD_PHASE}` that behaves as the default implementation for that EAPI shall be defined when executing any given `EBUILD_PHASE`.

Ebuilds must not call these functions except when in the phase in question.

Table 10.4: EAPIs supporting `default_phase` functions

EAPI	Supports <code>default_phase</code> functions?
0	No
1	No
2	Yes

10.2 Call Order

The call order for installing a package is:

- `pkg_setup`
- `src_unpack`
- `src_prepare` (only for EAPIs listed in table 10.1)
- `src_configure` (only for EAPIs listed in table 10.2)
- `src_compile`
- `src_test` (except if `RESTRICT=test`)
- `src_install`
- `pkg_preinst`
- `pkg_postinst`

The call order for uninstalling a package is:

- `pkg_prerm`
- `pkg_postrm`

The call order for reinstalling a package is:

- `pkg_setup`
- `src_unpack`
- `src_prepare` (only for EAPIs listed in table 10.1)
- `src_configure` (only for EAPIs listed in table 10.2)
- `src_compile`
- `src_test` (except if `RESTRICT=test`)
- `src_install`
- `pkg_preinst`
- `pkg_prerm` for the package being replaced
- `pkg_postrm` for the package being replaced
- `pkg_postinst`

The call order for upgrading or downgrading a package is:

- `pkg_setup`
- `src_unpack`
- `src_prepare` (only for EAPIs listed in table 10.1)
- `src_configure` (only for EAPIs listed in table 10.2)
- `src_compile`
- `src_test` (except if `RESTRICT=test`)
- `src_install`
- `pkg_preinst`
- `pkg_postinst`
- `pkg_prerm` for the package being replaced
- `pkg_postrm` for the package being replaced

The `pkg_config`, `pkg_info` and `pkg_nofetch` functions are not called in a normal sequence.

For installing binary packages, the `src` phases are not called.

When building binary packages that are not to be installed locally, the `pkg_preinst` and `pkg_postinst` functions are not called.

Chapter 11

The Ebuild Environment

11.1 Defined Variables

The package manager must define the following environment variables. Not all variables are meaningful in all phases; variables that are not meaningful in a given phase may be unset or set to any value. Ebuilds must not attempt to modify any of these variables, unless otherwise specified.

Because of their special meanings, these variables may not be preserved consistently across all phases as would normally happen due to environment saving (see 11.2). For example, `EBUILD_PHASE` is different for every phase, and `ROOT` may have changed between the various different `pkg_*` phases. Ebuilds must recalculate any variable they derive from an inconsistent variable.

Table 11.1: Defined variables

Variable	Legal in	Consistent?	Description
P	all	No ¹	Package name and version, without the revision part. For example, <code>vim-7.0.174</code> .
PN	all	ditto	Package name, for example <code>vim</code> .
CATEGORY	all	ditto	The package's category, for example <code>app-editors</code> .
PV	all	Yes	Package version, with no revision. For example <code>7.0.174</code> .
PR	all	Yes	Package revision, or <code>r0</code> if none exists.
PVR	all	Yes	Package version and revision, for example <code>7.0.174-r0</code> or <code>7.0.174-r1</code> .
PF	all	Yes	Package name, version, and revision, for example <code>vim-7.0.174-r1</code> .
A	<code>src_*</code>	Yes	All source files available for the package, whitespace separated with no leading or trailing whitespace, and in the order in which the item first appears in a matched component of <code>SRC_URI</code> . Does not include any that are disabled because of USE conditionals. The value is calculated from the base names of each element of the <code>SRC_URI</code> ebuild metadata variable.
AA ²	<code>src_*</code>	Yes	All source files that could be available for the package, including any that are disabled in <code>A</code> because of USE conditionals. The value is calculated from the base names of each element of the <code>SRC_URI</code> ebuild metadata variable.

¹May change if a package has been updated (see 3.4.4)

²This variable is generally considered deprecated. However, ebuilds must still assume that the package manager sets it. For example, a few

Variable	Legal in	Consistent?	Description
FILESDIR	src_* ³	No	The full path to the package's files directory, used for small support files or patches. See section 3.3. May or may not exist; if a repository provides no support files for the package in question then an ebuild must be prepared for the situation where FILESDIR points to a non-existent directory.
PORTDIR	ditto	No	The full path to the master repository's base directory.
DISTDIR	ditto	No	The full path to the directory in which the files in the A variable are stored.
ECLASSDIR	ditto	No	The full path to the master repository's eclass directory.
ROOT	pkg_*	No	The absolute path to the root directory into which the package is to be merged. Phases which run with full filesystem access must not touch any files outside of the directory given in ROOT. Also of note is that in a cross-compiling environment, binaries inside of ROOT will not be executable on the build machine, so ebuilds must not call them. ROOT must be non-empty and end in a trailing slash.
T	All	Partially ⁴	The full path to a temporary directory for use by the ebuild.
TMPDIR	All	Ditto	Must be set to the location of a usable temporary directory, for any applications called by an ebuild. Must not be used by ebuilds directly; see T above.
HOME	All	Ditto	The full path to an appropriate temporary directory for use by any programs invoked by the ebuild that may read or modify the home directory.
D	src_install	No	Contains the full path to the image directory into which the package should be installed. Must be non-empty and end in a trailing slash.
D (continued)	pkg_preinst, pkg_postinst	Yes	Contains the full path to the image that is about to be or has just been merged. Must be non-empty and end in a trailing slash.
IMAGE ⁵	pkg_preinst, pkg_postinst	Yes	Equal to D.
INSDESTTREE	src_install	No	Controls the location where doins installs things.
USE	All	Yes	A whitespace-delimited list of all active USE flags for this ebuild, including those originating from variables in USE_EXPAND.
EBUILD_PHASE	All	No	Takes one of the values config, setup, nofetch, unpack, prepare, configure, compile, test, install, preinst, postinst, prerm, postrm, info according to the top level ebuild function that was executed by the package manager. May be unset or any single word that is not any of the above when the ebuild is being sourced for other (e.g. metadata or QA) purposes.
WORKDIR	src_*	Yes	The full path to the ebuild's working directory, in which all build data should be contained.

configure scripts use this variable to find the aalib package; ebuilds calling such configure scripts must thus work around this.

³Not necessarily present when installing from a binary package

⁴Consistent and preserved across a single connected sequence of install or uninstall phases, but not between install and uninstall. When reinstalling a package, this variable must have different values for the install and the replacement.

⁵Deprecated in favour of D.

Variable	Legal in	Consistent?	Description
KV	All	Yes	The version of the running kernel at the time the ebuild was first executed, as returned by the <code>uname -r</code> command or equivalent. May be modified by ebuilds.

All variables set in the active profiles' `make.defaults` files must be exported to the ebuild environment. `CHOST`, `CBUILD` and `CTARGET`, if not set by profiles, must contain either an appropriate machine tuple (the definition of appropriate is beyond the scope of this specification) or be unset.

`PATH` must be initialized by the package manager to a “usable” default. The exact value here is left up to interpretation, but it should include the equivalent “`sbin`” and “`bin`” and any package manager specific directories.

`GZIP`, `BZIP`, `BZIP2`, `CDPATH`, `GREP_OPTIONS`, `GREP_COLOR` and `GLOBIGNORE` must not be set.

11.2 The state of variables between functions

Exported and default scope variables are saved between functions. A non-local variable set in a function earlier in the call sequence must have its value preserved for later functions, including functions executed as part of a later `uninstall`. Variables that were exported must remain exported in later functions; variables with default visibility may retain default visibility or be exported.

Variables with special meanings to the package manager are excluded from this rule.

Global variables must only contain invariant values (see 8.1). If a global variable's value is invariant, it may have the value that would be generated at any given point in the build sequence.

This is demonstrated by code listing 11.1.

11.3 Available commands

This section documents the commands available to an ebuild. Unless otherwise specified, they may be aliases, shell functions, or executables in the ebuild's `PATH`.

When an ebuild is being sourced for metadata querying rather than for a build (that is to say, when none of the `src_` or `pkg_` functions are to be called), no external command may be executed. The package manager may take steps to enforce this.

11.3.1 System commands

Any ebuild not listed in the system set for the active profile(s) may assume the presence of every command that is always provided by the system set for that profile. However, it must target the lowest common denominator of all systems on which it might be installed—in most cases this means that the only packages that can be assumed to be present are those listed in the `base` profile or equivalent, which is inherited by all available profiles. If an ebuild requires any applications not provided by the system profile, or that are provided conditionally based on `USE` flags, appropriate dependencies must be used to ensure their presence.

Guaranteed system commands

The following commands must always be available in the ebuild environment:

- All builtin commands in GNU bash, version 3.0.
- `sed` must be available, and must support all forms of invocations valid for GNU `sed` version 4 or later.
- `patch` must be available, and must support all inputs valid for GNU `patch`.

Listing 11.1: Environment state between functions

```
GLOBAL_VARIABLE="a"

src_compile()
{
    GLOBAL_VARIABLE="b"
    DEFAULT_VARIABLE="c"
    export EXPORTED_VARIABLE="d"
    local LOCAL_VARIABLE="e"
}

src_install(){
    [[ ${GLOBAL_VARIABLE} == "a" ]] \
        || [[ ${GLOBAL_VARIABLE} == "b" ]] \
        || die "broken env saving for globals"

    [[ ${DEFAULT_VARIABLE} == "c" ]] \
        || die "broken env saving for default"

    [[ ${EXPORTED_VARIABLE} == "d" ]] \
        || die "broken env saving for exported"

    [[ $(printenv EXPORTED_VARIABLE) == "d" ]] \
        || die "broken env saving for exported"

    [[ -z ${LOCAL_VARIABLE} ]] \
        || die "broken env saving for locals"
}
```

11.3.2 Commands provided by package dependencies

In some cases a package's build process will require the availability of executables not provided by the core system, a common example being autotools. Commands provided by dependencies are available in the following cases:

- In the `src` phases, any command provided by a package listed in `DEPEND` is available.
- In the `pkg` phases, at least one of the following conditions must be met:
 - Any command provided by a package listed in `DEPEND` is available.
 - Any command provided by a package listed in `RDEPEND` is available.

11.3.3 Ebuild-specific Commands

The following commands will always be available in the ebuild environment, provided by the package manager. Except where otherwise noted, they may be internal (shell functions or aliases) or external commands available in `PATH`; where this is not specified, ebuilds may not rely upon either behaviour.

Sandbox commands

These commands affect the behaviour of the sandbox. Each command takes a single directory as argument. Ebuilds must not run any of these commands once the current phase function has returned.

addread Add a directory to the permitted read list.

addwrite Add a directory to the permitted write list.

addpredict Add a directory to the predict list. Any write to a location in this list will be denied, but will not trigger access violation messages or abort the build process.

adddeny Add a directory to the deny list.

Package manager query commands

These commands are used to extract information about the host system. Ebuilds must not run any of these commands in parallel with any other package manager command. Ebuilds must not run any of these commands once the current phase function has returned.

has_version Takes exactly one package dependency specification as an argument. Returns true if a package matching the atom is installed in `$ROOT`, and false otherwise.

best_version Takes exactly one package dependency specification as an argument. If a matching package is installed, prints the category, package name and version of the highest matching version.

Output commands

These commands display messages to the user. Unless otherwise stated, the entire argument list is used as a message, as in the simple invocations of `echo`. Ebuilds must not run any of these commands once the current phase function has returned. Unless otherwise noted, output may be sent to `stdout`, `stderr` or some other appropriate facility.

einfo Displays an informational message.

einfof Displays an informational message without a trailing newline.

eelog Displays an informational message of slightly higher importance. The package manager may choose to log `eelog` messages by default where `einfo` messages are not, for example.

ewarn Displays a warning message. Must not go to `stdout`.

error Displays an error message. Must not go to stdout.

ebegin Displays an informational message. Should be used when beginning a possibly lengthy process, and followed by a call to `epend`.

end Indicates that the process begun with an `ebegin` message has completed. Takes one fixed argument, which is a numeric return code, and an optional message in all subsequent arguments. If the first argument is 0, print a success indicator; otherwise, print the message followed by a failure indicator.

Error commands

These commands are used when an error is detected that will prevent the build process from completing. Ebuilds must not run any of these commands once the current phase function has returned.

die Displays a failure message provided in its first and only argument, and then aborts the build process. `die` is *not* guaranteed to work correctly if called from a subshell environment.

assert Checks the value of the shell's pipe status variable, and if any component is non-zero (indicating failure), calls `die` with its first argument as a failure message.

Build commands

These commands are used during the `src_compile` and `src_install` phases to run the package's build commands. Ebuilds must not run any of these commands once the current phase function has returned.

econf Calls the program's `./configure` script. This is designed to work with GNU Autoconf-generated scripts. Any additional parameters passed to `econf` are passed directly to `./configure`. `econf` will look in the current working directory for a configure script unless the `ECONF_SOURCE` environment variable is set, in which case it is taken to be the directory containing it. `econf` must pass the following options to the configure script:

- `--prefix` must default to `/usr` unless overridden by `econf`'s caller.
- `--mandir` must be `/usr/share/man`
- `--infodir` must be `/usr/share/info`
- `--datadir` must be `/usr/share`
- `--sysconfdir` must be `/etc`
- `--localstatedir` must be `/var/lib`
- `--host` must be the value of the `CHOST` environment variable.
- `--libdir` must be set according to Algorithm 3.

`econf` must be implemented internally—that is, as a bash function and not an external script. Should any portion of it fail, it must abort the build using `die`.

emake Calls the `$MAKE` program, or GNU `make` if the `MAKE` variable is unset. Any arguments given are passed directly to the `make` command, as are the user's chosen `MAKEOPTS`. Arguments given to `emake` override user configuration. See also section 11.3.1. `emake` must be an external program and cannot be a function or alias—it must be callable from e.g. `xargs`.

einstall A shortcut for the command given in Listing 11.2. Any arguments given to `einstall` are passed verbatim to `emake`, as shown.

Algorithm 3 `econf --libdir` logic

```
1: let prefix=/usr
2: if the caller specified --prefix=$p then
3:   let prefix=$p
4: end if
5: let libdir=
6: if the ABI environment variable is set then
7:   let libvar=LIBDIR_ABI
8:   if the environment variable named by libvar is set then
9:     let libdir=the value of the variable named by libvar
10:  end if
11: end if
12: if libdir is non-empty then
13:   pass --libdir=$prefix/$libdir to configure
14: end if
```

Listing 11.2: `install` command

```
emake \
  prefix="${D}"/usr \
  mandir="${D}"/usr/share/man \
  infodir="${D}"/usr/share/info \
  libdir="${D}"/usr/$(get_libdir) \
  "$@" \
  install
```

Installation commands

These commands are used to install files into the staging area, in cases where the package's `make install` target cannot be used or does not install all needed files. Except where otherwise stated, all filenames created or modified are relative to the staging directory, given by `{D}`. These commands must all be external programs and not bash functions or aliases—that is, they must be callable from `xargs`. `Ebuilds` must not run any of these commands once the current phase function has returned.

dobin Installs the given files into `DESTTREE/bin`, where `DESTTREE` defaults to `/usr`. Gives the files mode `0755` and ownership `root:root`.

doconfd Installs the given files into `/etc/conf.d/`.

dodir Creates the given directories, by default with file mode `0755`. This can be overridden by setting `DIROPTIONS` with the `diropts` function.

dodoc Installs the given files into a subdirectory under `/usr/share/doc/$PF/`. The subdirectory is set by the most recent call to `docinto`. If `docinto` has not yet been called, instead installs to the directory `/usr/share/doc/$PF/`.

doexe Installs the given files into the directory specified by the most recent `exeinto` call. If `exeinto` has not yet been called, behaviour is undefined.

dohard Takes two parameters. Creates a hardlink from the second to the first.

dohtml Installs the given HTML files into a subdirectory under `/usr/share/doc/$PF/`. The subdirectory is `html` by default, but this can be changed by the `$DOCDESTTREE` variable. Files to be installed automatically are determined by extension and the default extensions are `css`, `gif`, `htm`, `html`, `jpeg`, `jpg`, `js` and `png`. These default extensions can be extended or reduced (see below). The options that can be passed to `dohtml` are as follows:

- r — enables recursion into directories.
- V — enables verbosity.
- A — adds file type extensions to the default list.
- a — sets file type extensions to only those specified.
- f — list of files that are able to be installed.
- x — list of directories that files will not be installed from(only used in conjunction with -r).
- p — sets a document prefix for installed files.

doinfo Installs a GNU Info file into the `/usr/share/info` area.

doinitd Installs an initscript into `/etc/init.d`.

doins Takes any number of files as arguments and installs them into `$INSDESTTREE`. If the first argument is `-r`, then operates recursively, descending into any directories given.

dolib For each argument, installs it into the appropriate library directory as determined by Algorithm 4. Any symlinks are installed into the same directory as relative links to their original target.

dolib.so As for `dolib`. Installs the file with mode `0755`.

dolib.a As for `dolib`. Installs the file with mode `0644`.

Algorithm 4 Determining the library directory

```

1: if CONF_LIBDIR_OVERRIDE is set in the environment then
2:   return CONF_LIBDIR_OVERRIDE
3: end if
4: if CONF_LIBDIR is set in the environment then
5:   let LIBDIR_default=CONF_LIBDIR
6: else
7:   let LIBDIR_default="lib"
8: end if
9: if ABI is set in the environment then
10:  let abi=ABI
11: else if DEFAULT_ABI is set in the environment then
12:  let abi=DEFAULT_ABI
13: else
14:  let abi="default"
15: end if
16: return the value of LIBDIR_$abi

```

doman Installs a man page into the appropriate subdirectory of `/usr/share/man` depending upon its parent section suffix (e.g. `foo.1` goes to `/usr/share/man/man1/foo.1`. In EAPIs listed in table 11.2 as supporting language codes, a man page with name of the form `foo.lang.1` shall go to `/usr/share/man/lang/man1/foo.1`, where `lang` refers to a pair of lower-case ASCII letters optionally followed by an underscore and a pair of upper-case ASCII letters.

domo Installs a `.mo` file into the appropriate subdirectory of `DESTTREE/share/locale`, generated by taking the basename of the file, removing the `.*` suffix, and appending `/LC_MESSAGES`.

dosbin As `dobin`, but installs to `DESTTREE/sbin`.

dosym Creates a symbolic link named as for its second parameter, pointing to the first. If the directory containing the new link does not exist, creates it.

fowners Acts as for `chown`, but takes paths relative to the image directory.

fperms Acts as for `chmod`, but takes paths relative to the image directory.

Table 11.2: EAPIs supporting domain languages

EAPI	Supports domain languages?
0	No
1	No
2	Yes

newbin As for `dobin`, but takes two parameters. The first is the file to install; the second is the new filename under which it will be installed.

newconfd As for `doconfd`, but takes two parameters as for `newbin`.

newdoc As above, for `dodoc`.

newenvd As above, for `doenvd`.

newexe As above, for `doexe`.

newinitd As above, for `doinitd`.

newins As above, for `doins`.

newman As above, for `doman`.

newsbin As above, for `dosbin`.

keepdir Creates a directory as for `dodir`, and an empty file whose name starts with `.keep` in that directory to ensure that the directory does not get removed by the package manager should it be empty at any point.

Commands affecting install destinations

The following commands are used to set the various destination trees, all relative to `#{D}`, used by the above installation commands. They must be shell functions or aliases, due to the need to set variables read by the above commands. Ebuilds must not run any of these commands once the current phase function has returned.

into Sets the value of `DESTTREE` for future invocations of the above utilities. Creates the directory under `#{D}`, using `install -d` with no additional options, if it does not already exist.

insinto Sets the value of `INSDESTTREE` for future invocations of the above utilities. May create the directory, as specified for `into`.

exeinto Sets the install path for `doexe` and `newexe`. May create the directory, as specified for `into`.

docinto Sets the install subdirectory for `dodoc` et al. May create the directory, as specified for `into`.

insopts Sets the options passed by `doins` et al. to the `install` command.

diropts Sets the options passed by `dodir` et al. to the `install` command.

exeopts Sets the options passed by `doexe` et al. to the `install` command.

libopts Sets the options passed by `dolib` et al. to the `install` command.

List Functions

These functions work on variables containing whitespace-delimited lists (e.g. `USE`). Ebuilds must not run any of these commands once the current phase function has returned. Ebuilds must not call any function that operates upon `USE` to query a value that is not either listed in `IUSE`, a `USE_EXPAND` value or an `ARCH` value; package manager behaviour is undefined if such a call is made.

use Returns shell true (0) if the first argument (a `USE` flag name) is enabled, false otherwise. If the flag name is prefixed with `!`, returns true if the flag is disabled, and false if it is enabled.

usev The same as `use`, but also prints the flag name if it is enabled.

useq Deprecated synonym for `use`.

has Returns shell true (0) if the first argument (a word) is found in the list of subsequent arguments, false otherwise.

hasv The same as `has`, but also prints the first argument if found.

hasq Deprecated synonym for `has`.

use_with Has one-, two-, and three-argument forms. The first argument is a USE flag name, the second a configure option name (`$opt`), defaulting to the same as the first argument if not provided, and the third is a string value (`$value`), defaulting to nothing. If the USE flag is set, outputs `-with-$opt=$value` if the third argument was provided, and `-with-$opt` otherwise. If the flag is not set, then it outputs `-without-$opt`.

use_enable Works the same as `use_with()`, but outputs `-enable-` or `-disable-` instead of `-with-` or `-without-`.

Misc Commands

The following commands are always available in the ebuild environment, but don't really fit in any of the above categories. Ebuilds must not run any of these commands once the current phase function has returned.

dosed Takes any number of arguments, which can be files or `sed` expressions. For each argument, if it names, relative to `D` a file which exists, then `sed` is run with the current expression on that file. Otherwise, the current expression is set to the text of the argument. The initial value of the expression is `s:${D}::g`.

unpack Unpacks one or more source archives, in order, into the current directory. After unpacking, must ensure that all filesystem objects inside the current working directory (but not the current working directory itself) have permissions `a+r, u+w, go-w` and that all directories under the current working directory additionally have permissions `a+x`.

All arguments to `unpack` must be either a filename without path, in which case `unpack` looks in `DISTDIR` for the file, or start with the string `./`, in which case `unpack` uses the argument as a path relative to the working directory.

Must be able to unpack the following file formats, if the relevant binaries are available:

- tar files (`*.tar`). Ebuilds must ensure that GNU tar installed.
- gzip-compressed tar files (`*.tar.gz`, `*.tgz`, `*.tar.Z`, `*.tbz`). Ebuilds must ensure that GNU gzip and GNU tar are installed.
- bzip2-compressed tar files (`*.tar.bz2`, `*.tbz2`, `*.tar.bz`). Ebuilds must ensure that bzip2 and GNU tar are installed.
- zip files (`*.zip`, `*.ZIP`, `*.jar`). Ebuilds must ensure that Info-ZIP Unzip is installed.
- gzip files (`*.gz`, `*.Z`, `*.z`). Ebuilds must ensure that GNU gzip is installed.
- bzip2 files (`*.bz`, `*.bz2`). Ebuilds must ensure that bzip2 is installed.
- 7zip files (`*.7z`, `*.7Z`). Ebuilds must ensure that P7ZIP is installed.
- rar files (`*.rar`, `*.RAR`). Ebuilds must ensure that RARLAB's unrar is installed.
- LHA archives (`*.LHA`, `*.LHa`, `*.lha`, `*.lhz`). Ebuilds must ensure that the lha program is installed.
- ar archives (`*.a`, `*.deb`). Ebuilds must ensure that GNU binutils is installed.

Table 11.3: EAPIs supporting the default function

EAPI	Supports default function?
0	No
1	No
2	Yes

- lzma archives (`*.lzma`). Ebuilds must ensure that LZMA Utils is installed.

It is up to the ebuild to ensure that the relevant external utilities are available, whether by being in the system set or via dependencies.

inherit See section 7.1.

default Calls the `default_${EBUILD_PHASE}` function. Only available in EAPIs listed in table 11.3.

Debug Commands

The following commands are available for debugging. Normally all of these commands should be no ops; a package manager may provide a special debug mode where these commands instead do something. Ebuilds must not run any of these commands once the current phase function has returned.

debug-print If in a special debug mode, the arguments should be outputted or recorded using some kind of debug logging.

debug-print-function Calls `debug-print` with `$1: entering function` as the first argument and the remaining arguments as additional arguments.

debug-print-section Calls `debug-print` with `now in section $*`.

Reserved Commands and Variables

Except where documented otherwise, all functions and variables that contain any of the following strings (ignoring case) are reserved for package manager use and may not be used or relied upon by ebuilds:

- abort
- dyn
- ebuild
- hook
- paludis
- portage
- prep

11.4 The state of the system between functions

For the sake of this section:

- Variancy is any package manager action that modifies either `ROOT` or `/` in any way that isn't merely a simple addition of something that doesn't alter other packages. This includes any non-default call to any `pkg` phase function except `pkg_setup`, a merge of any package or an unmerge of any package.
- As an exception, changes to `DISTDIR` do not count as variancy.
- The `pkg_setup` function may be assumed not to introduce variancy. Thus, ebuilds must not perform variant actions in this phase.

The following exclusivity and invariancy requirements are mandated:

- No variancy shall be introduced at any point between a package's `pkg_setup` being started up to the point that that package is merged, except for any variancy introduced by that package.
- There must be no variancy between a package's `pkg_setup` and a package's `pkg_postinst`, except for any variancy introduced by that package.
- Any non-default `pkg` phase function must be run exclusively.
- Each phase function must be called at most once during the build process for any given package.

Chapter 12

Merging and Unmerging

Note: In this chapter, *file* and *regular file* have their Unix meanings.

12.1 Overview

The merge process merges the contents of the `D` directory onto the filesystem under `ROOT`. This is not a straight copy; there are various subtleties which must be addressed.

The unmerge process removes an installed package's files. It is not covered in detail in this specification.

12.2 Directories

Directories are merged recursively onto the filesystem. The method used to perform the merge is not specified, so long as the end result is correct. In particular, merging a directory may alter or remove the source directory under `D`.

Ebuilds must not attempt to merge a directory on top of any existing file that is not either a directory or a symlink to a directory.

12.2.1 Permissions

The owner, group and mode (including set*id and sticky bits) of the directory must be preserved, except as follows:

- Any directory owned by the user used to perform the build must become owned by the root user.
- Any directory whose group is the primary group of the user used to perform the build must have its group be that of the root user.

On SELinux systems, the SELinux context must also be preserved. Other directory attributes, including modification time, may be discarded.

12.2.2 Empty Directories

Behaviour upon encountering an empty directory is undefined. Ebuilds must not attempt to install an empty directory.

12.3 Regular Files

Regular files are merged onto the filesystem (but see the notes on configuration file protection, below). The method used to perform the merge is not specified, so long as the end result is correct. In particular, merging a regular file may alter or remove the source file under `D`.

Ebuilds must not attempt to merge a regular file on top of any existing file that is not either a regular file or a symlink to a regular file.

12.3.1 Permissions

The owner, group and mode (including set*id and sticky bits) of the file must be preserved, except as follows:

- Any file owned by the user used to perform the build must become owned by the root user.
- Any file whose group is the primary group of the user used to perform the build must have its group be that of the root user.
- The package manager may reduce read and write permissions on executable files that have a set*id bit set.

On SELinux systems, the SELinux context must also be preserved. Other file attributes, including modification time, may be discarded.

12.3.2 Configuration File Protection

The package manager must provide a means to prevent user configuration files from being overwritten by any package updates. The profile variables `CONFIG_PROTECT` and `CONFIG_PROTECT_MASK` (section 4.3) control the paths for which this must be enforced.

In order to ensure interoperability with configuration update tools, the following scheme must be used by all package managers when merging any regular file:

1. If the directory containing the file to be merged is not listed in `CONFIG_PROTECT`, and is not a subdirectory of any such directory, and if the file is not listed in `CONFIG_PROTECT`, the file is merged normally.
2. If the directory containing the file to be merged is listed in `CONFIG_PROTECT_MASK`, or is a subdirectory of such a directory, or if the file is listed in `CONFIG_PROTECT_MASK`, the file is merged normally.
3. If no existing file with the intended filename exists, or the existing file has identical content to the one being merged, the file is installed normally.
4. Otherwise, prepend the filename with `._cfg0000_`. If no file with the new name exists, then the file is merged with this name.
5. Otherwise, increment the number portion (to form `._cfg0001_<name>`) and repeat step 4. Continue this process until a usable filename is found.
6. If 9999 is reached in this way, behaviour is undefined.

12.4 Symlinks

Symlinks are merged as symlinks onto the filesystem. The link destination for a merged link shall be the same as the link destination for the link under `D`, except as noted below. The method used to perform the merge is not specified, so long as the end result is correct; in particular, merging a symlink may alter or remove the symlink under `D`.

Ebuilds must not attempt to merge a symlink on top of a directory.

12.4.1 Rewriting

Any absolute symlink whose link starts with `D` must be rewritten with the leading `D` removed. The package manager should issue a notice when doing this.

12.5 Hard links

A hard link may be merged either as a single file with links or as multiple independent files.

12.6 Other Files

Ebuilds must not attempt to install any other type of file (FIFOs, device nodes etc).

Chapter 13

Glossary

This section contains explanations of some of the terms used in this document whose meaning may not be immediately obvious.

qualified package name A package name along with its associated category. For example, `app-editors/vim` is a qualified package name.

old-style virtual An old-style virtual is a psuedo-package which exists if it is listed in an ebuild's `PROVIDE` variable. See chapter 5.

new-style virtual A new-style virtual is a normal package in the `virtual` category which installs no files and uses its dependency requirements to pull in a 'provider'. This is more flexible than the old-style virtuals described above, and requires no special package manager code.

stand-alone repository An (ebuild) repository which is intended to function on its own as the only, or primary, repository on a system. Contrast with *slave repository* below.

slave repository, non-stand-alone repository An (ebuild) repository which is not complete enough to function on its own, but needs one or more *master repositories* to satisfy dependencies and provide repository-level support files. Known in Portage as an overlay.

master repository See above.

Appendix A

metadata.xml

The `metadata.xml` file is used to contain extra package- or category-level information beyond what is stored in ebuild metadata. Its exact format is strictly beyond the scope of this document, and is described in the DTD file located at <http://www.gentoo.org/dtd/metadata.dtd>.

Appendix B

Unspecified Items

The following items are not specified by this document, and must not be relied upon by ebuilds. This is, of course, an incomplete list—it covers only the things that the authors know have been abused in the past.

- The `FEATURES` variable. This is Portage specific.
- Similarly, any `PORTAGE_` variable not explicitly listed.
- Any Portage configuration file.
- The VDB (`/var/db/pkg`). Ebuilds must not access this or rely upon it existing or being in any particular format.
- The `portageq` command. The `has_version` and `best_version` commands are available as functions.
- The `emerge` command.
- Binary packages.
- The `PORTDIR_OVERLAY` variable, and overlay behaviour in general.

Appendix C

Historical Curiosities

The items described in this chapter are included for information only. They were deprecated or abandoned long before EAPI was introduced. Ebuilds must not use these features, and package managers should not be changed to support them.

C.1 If-else use blocks

Historically, Portage supported if-else use conditionals, as shown by listing C.1. The block before the colon would be taken if the condition was met, and the block after the colon would be taken if the condition was not met.

This feature was deprecated and removed from the tree long before the introduction of EAPI.

C.2 cvs Versions

Portage has very crude support for CVS packages. The package `foo` could contain a file named `foo-cvs.1.2.3.ebuild`. This version would order *higher* than any non-CVS version (including `foo-2.ebuild`). This feature has not seen real world use and breaks versioned dependencies, so it must not be used.

Listing C.1: If-else use blocks

```
DEPEND="
  flag? (
    taken/if-true
  ) : (
    taken/if-false
  )
"
```

Appendix D

Feature Availability by EAPI

Note: This chapter is informative and for convenience only. Refer to the main text for specifics.

Table D.1: Features in EAPIs

Feature	Reference	EAPIs		
		0	1	2
IUSE defaults	table 8.1	No	Yes	Yes
SRC_URI arrows	table 9.1	No	No	Yes
Slot dependencies	table 9.2	No	Yes	Yes
Use dependencies	table 9.3	No	No	2-style
! blockers	table 9.4	Unspecified	Unspecified	Weak
!! blockers	table 9.4	Forbidden	Forbidden	Strong
src_prepare	table 10.1	No	No	Yes
src_configure	table 10.2	No	No	Yes
src_compile style	table 10.3	0	1	2
default_phase functions	table 10.4	No	No	Yes
doman languages	table 11.2	No	No	Yes
default function	table 11.3	No	No	Yes

Appendix E

Differences Between EAPIs

Note: This chapter is informative and for convenience only. Refer to the main text for specifics.

EAPI 0

EAPI 0 is the base EAPI.

EAPI 1

EAPI 1 is EAPI 0 with the following changes:

- IUSE defaults, table 8.1.
- Slot dependencies, table 9.2.
- Different `src_compile` implementation, table 10.3.

EAPI 2

EAPI 2 is EAPI 1 with the following changes:

- SRC_URI arrows, table 9.1.
- Use dependencies, table 9.3.
- ! and !! blockers, table 9.4.
- `src_prepare`, table 10.1.
- `src_configure`, table 10.2.
- Different `src_compile` implementation, table 10.3.
- `default_phase` functions, table 10.4.
- domain languages support, table 11.2.
- `default` function, table 11.3.

Bibliography

- [1] Marius Mauch. GLEP 44: Manifest2 format. <http://glep.gentoo.org/glep-0044.html>, December 2005.