Package Manager Specification

Stephen P. Bennett
spb@exherbo.org

Christian Faulhammer
fauli@gentoo.org

Ciaran McCreesh
ciaran.mccreesh@googlemail.com

Ulrich Müller
ulm@gentoo.org

21st June 2025

This version corresponds to commit e4269e8 on branch eapi-9.

Contents

1 Introduction **11**
- 1.1 Aims and motivation . . . 11
- 1.2 Rationale . . . 11
- 1.3 Reporting issues . . . 11
- 1.4 Conventions . . . 12
- 1.5 Acknowledgements . . . 12

2 EAPIs **13**
- 2.1 Definition . . . 13
- 2.2 Defined EAPIs . . . 13
- 2.3 Reserved EAPIs . . . 13

3 Names and versions **14**
- 3.1 Restrictions upon names . . . 14
  - 3.1.1 Category names . . . 14
  - 3.1.2 Package names . . . 14
  - 3.1.3 Slot names . . . 14
  - 3.1.4 USE flag names . . . 14
  - 3.1.5 Repository names . . . 14
  - 3.1.6 Eclass names . . . 14
  - 3.1.7 License names . . . 15
  - 3.1.8 Keyword names . . . 15
  - 3.1.9 EAPI names . . . 15
- 3.2 Version specifications . . . 15
- 3.3 Version comparison . . . 15
- 3.4 Uniqueness of versions . . . 15

4 Tree layout **18**
- 4.1 Top level . . . 18
- 4.2 Category directories . . . 18
- 4.3 Package directories . . . 19
- 4.4 The profiles directory . . . 19
  - 4.4.1 The profiles.desc file . . . 20
  - 4.4.2 The thirdpartymirrors file . . . 20
  - 4.4.3 use.desc and related files . . . 21
  - 4.4.4 The updates directory . . . 21
- 4.5 The licenses directory . . . 21
- 4.6 The eclass directory . . . 21
- 4.7 The metadata directory . . . 22
  - 4.7.1 The metadata cache . . . 22

5 Profiles **23**
- 5.1 General principles . . . 23
- 5.2 Files that make up a profile . . . 23
  - 5.2.1 The parent file . . . 23

5.2.2 The eapi file . . . 23
5.2.3 deprecated . . . 24
5.2.4 make.defaults . . . 24
5.2.5 Simple line-based files . . . 24
5.2.6 packages . . . 24
5.2.7 packages.build . . . 25
5.2.8 package.mask . . . 25
5.2.9 package.provided . . . 25
5.2.10 package.use . . . 25
5.2.11 use.stable and package.use.stable . . . 25
5.2.12 USE masking and forcing . . . 26
5.3 Profile variables . . . 26
5.3.1 Incremental variables . . . 28
5.3.2 Specific variables and their meanings . . . 28

6 Ebuild file format 30

7 Ebuild-defined variables 31
7.1 Metadata invariance . . . 31
7.2 Mandatory ebuild-defined variables . . . 31
7.3 Optional ebuild-defined variables . . . 31
7.3.1 EAPI . . . 32
7.3.2 SRC_URI . . . 33
7.3.3 Keywords . . . 33
7.3.4 USE state constraints . . . 34
7.3.5 Properties . . . 34
7.3.6 Restrict . . . 34
7.3.7 RDEPEND value . . . 34
7.4 Magic ebuild-defined variables . . . 35

8 Dependencies 36
8.1 Dependency classes . . . 36
8.2 Dependency specification format . . . 37
8.2.1 All-of dependency specifications . . . 38
8.2.2 USE-conditional dependency specifications . . . 38
8.2.3 Any-of dependency specifications . . . 38
8.2.4 Exactly-one-of dependency specifications . . . 38
8.2.5 At-most-one-of dependency specifications . . . 39
8.3 Package dependency specifications . . . 39
8.3.1 Operators . . . 39
8.3.2 Block operator . . . 40
8.3.3 Slot dependencies . . . 40
8.3.4 2-style and 4-style USE dependencies . . . 40

9 Ebuild-defined functions 42
9.1 List of functions . . . 42
9.1.1 Initial working directories . . . 42
9.1.2 pkg_pretend . . . 42
9.1.3 pkg_setup . . . 43
9.1.4 src_unpack . . . 43
9.1.5 src_prepare . . . 44
9.1.6 src_configure . . . 44
9.1.7 src_compile . . . 45
9.1.8 src_test . . . 46
9.1.9 src_install . . . 46
9.1.10 pkg_preinst . . . 47
9.1.11 pkg_postinst . . . 47

9.1.12 pkg_prerm . . . 47
9.1.13 pkg_postrm . . . 47
9.1.14 pkg_config . . . 47
9.1.15 pkg_info . . . 47
9.1.16 pkg_nofetch . . . 48
9.1.17 Default phase functions . . . 48
9.2 Call order . . . 48

10 Eclasses **50**
10.1 The inherit command . . . 50
10.2 Eclass-defined metadata keys . . . 50
10.3 EXPORT_FUNCTIONS . . . 50

11 The ebuild environment **52**
11.1 Defined variables . . . 52
11.1.1 USE and IUSE handling . . . 57
11.1.2 REPLACING_VERSIONS and REPLACED_BY_VERSION . . . 58
11.1.3 Offset-prefix variables . . . 58
11.1.4 Path variables and trailing slash . . . 59
11.2 The state of variables between functions . . . 59
11.3 The state of the system between functions . . . 59

12 Available commands **61**
12.1 System commands . . . 61
12.1.1 Guaranteed system commands . . . 61
12.2 Commands provided by package dependencies . . . 61
12.3 Ebuild-specific commands . . . 62
12.3.1 Failure behaviour and related commands . . . 62
12.3.2 Banned commands . . . 62
12.3.3 Sandbox commands . . . 62
12.3.4 Package manager query commands . . . 63
12.3.5 Output commands . . . 63
12.3.6 Error commands . . . 64
12.3.7 Patch commands . . . 65
12.3.8 Build commands . . . 65
12.3.9 Installation commands . . . 68
12.3.10 Commands affecting install destinations . . . 71
12.3.11 Commands controlling manipulation of files in the staging area . . . 72
12.3.12 USE list functions . . . 73
12.3.13 Text list functions . . . 75
12.3.14 Version manipulation and comparison commands . . . 75
12.3.15 Misc commands . . . 76
12.3.16 Debug commands . . . 79
12.3.17 Reserved commands and variables . . . 79

13 Merging and unmerging **80**
13.1 Overview . . . 80
13.2 Directories . . . 80
13.2.1 Permissions . . . 80
13.2.2 Empty directories . . . 80
13.3 Regular files . . . 80
13.3.1 Permissions . . . 81
13.3.2 File modification times . . . 81
13.3.3 Configuration file protection . . . 81
13.4 Symlinks . . . 82
13.4.1 Rewriting . . . 82
13.5 Hard links . . . 82

13.6 Other files 82

14 Metadata cache **83**
14.1 Directory contents 83
14.2 Legacy cache file format 83
14.3 md5-dict cache file format 84

Glossary **85**

Bibliography **86**

A metadata.xml **87**

B Unspecified items **88**

C Historical curiosities **89**
C.1 Long-obsolete features 89
C.2 Retroactive changes 89

D Feature availability by EAPI **91**

E Differences between EAPIs **93**

F Desk reference **97**

Algorithms

3.1 Version comparison top-level logic 16
3.2 Version comparison logic for numeric components 16
3.3 Version comparison logic for each numeric component after the first 16
3.4 Version comparison logic for letter components 16
3.5 Version comparison logic for suffixes 17
3.6 Version comparison logic for each suffix 17
3.7 Version comparison logic for revision components 17
5.1 `USE` masking logic 27
12.1 `eapply` logic 65
12.2 `econf --libdir` logic 67
12.3 `einstalldocs` logic 78
12.4 Library directory logic 78

Listings

9.1 `src_unpack` . . . 43
9.2 `src_prepare`, format 6 . . . 44
9.3 `src_prepare`, format 8 . . . 44
9.4 `src_configure` . . . 45
9.5 `src_compile`, format 0 . . . 45
9.6 `src_compile`, format 1 . . . 45
9.7 `src_compile`, format 2 . . . 45
9.8 `src_install`, format 4 . . . 46
9.9 `src_install`, format 6 . . . 47
10.1 `EXPORT_FUNCTIONS` example: `foo.eclass` . . . 51
11.1 Environment state between functions . . . 60
12.1 `einstall` command . . . 68
12.2 Create a relative path for `dosym -r` . . . 70
C.1 If-else use blocks . . . 89

Tables

4.1 EAPIs supporting a directory for `package.mask` . . . 20
4.2 Naming rules for files in `updates` directory for EAPIs . . . 21

5.1 Default EAPI for profiles . . . 24
5.2 EAPIs supporting directories for profile files . . . 24
5.3 EAPIs supporting `package.provided` in profiles . . . 25
5.4 EAPIs supporting `use.stable` and `package.use.stable` in profiles . . . 26
5.5 Profile directory support for masking/forcing use flags in stable versions only . . . 26
5.6 Profile-defined `IUSE` injection for EAPIs . . . 28
5.7 Profile-defined unsetting of variables in EAPIs . . . 28

6.1 Bash version and options . . . 30

7.1 EAPIs supporting `IUSE` defaults . . . 32
7.2 EAPIs supporting various ebuild-defined variables . . . 32
7.3 EAPIs supporting `SRC_URI` arrows and selective URI restrictions . . . 33
7.4 EAPIs with `RDEPEND=DEPEND` default . . . 35
7.5 EAPIs supporting `DEFINED_PHASES` . . . 35

8.1 Dependency classes required to be satisfied for a particular phase function . . . 36
8.2 Summary of other interfaces related to dependency classes . . . 37
8.3 Prefix values for `DEPEND` . . . 37
8.4 EAPIs supporting additional dependency types . . . 37
8.5 EAPIs supporting `REQUIRED_USE ??` groups . . . 38
8.6 Matching of empty dependency groups in EAPIs . . . 38
8.7 Support for `SLOT` dependencies and sub-slots in EAPIs . . . 39
8.8 EAPIs supporting `USE` dependencies . . . 39
8.9 Exclamation mark strengths for EAPIs . . . 40

9.1 Initial working directory in `pkg_*` phase functions for EAPIs . . . 43
9.2 EAPIs with `S` to `WORKDIR` fallbacks . . . 43
9.3 EAPIs supporting `pkg_pretend` . . . 43
9.4 `src_prepare` support and behaviour for EAPIs . . . 44
9.5 EAPIs supporting `src_configure` . . . 44
9.6 `src_compile` behaviour for EAPIs . . . 45
9.7 `src_test` behaviour for EAPIs . . . 46
9.8 `src_install` behaviour for EAPIs . . . 46
9.9 EAPIs supporting `pkg_info` on non-installed packages . . . 48
9.10 EAPIs supporting `default_` phase functions . . . 48

10.1 EAPIs accumulating `PROPERTIES` and `RESTRICT` across eclasses . . . 51

11.1 EAPIs with variables exported to the environment . . . 52
11.2 Defined variables . . . 53
11.3 EAPIs supporting various added env variables . . . 57
11.4 EAPIs supporting various removed env variables . . . 57

11.5 EAPIs supporting offset-prefix env variables . . . 57
11.6 Locale settings for EAPIs . . . 58
11.7 EAPIs supporting offset-prefix . . . 59
11.8 Variables that always or never end with a trailing slash . . . 59

12.1 System commands for EAPIs . . . 61
12.2 EAPI command failure behaviour . . . 62
12.3 Banned commands . . . 63
12.4 Package manager query command options supported by EAPIs . . . 63
12.5 Output commands for EAPIs . . . 64
12.6 Properties of `die` command in EAPIs . . . 64
12.7 EAPIs supporting `pipestatus` . . . 64
12.8 Patch commands for EAPIs . . . 66
12.9 Extra `econf` arguments for EAPIs . . . 67
12.10 EAPIs supporting `dodoc -r` . . . 71
12.11 EAPIs supporting `doheader` and `newheader` . . . 71
12.12 EAPIs supporting symlinks for `doins` . . . 71
12.13 `doman` language support options for EAPIs . . . 71
12.14 EAPIs supporting stdin for `new*` commands . . . 71
12.15 `domo` destination path in EAPIs . . . 72
12.16 EAPIs supporting `dosym -r` . . . 72
12.17 Commands respecting `insopts` for EAPIs . . . 72
12.18 Commands respecting `exeopts` for EAPIs . . . 72
12.19 Commands controlling manipulation of files in the staging area in EAPIs . . . 73
12.20 EAPI behaviour for use queries not in `IUSE_EFFECTIVE` . . . 74
12.21 `usev`, `use_with` and `use_enable` arguments for EAPIs . . . 74
12.22 EAPIs supporting `usex` and `in_iuse` . . . 74
12.23 EAPIs supporting version manipulation commands . . . 76
12.24 `unpack` behaviour for EAPIs . . . 77
12.25 `unpack` extensions for EAPIs . . . 77
12.26 Misc commands for EAPIs . . . 79

13.1 Preservation of file modification times (mtimes) . . . 81
13.2 Rewriting of absolute symlinks in EAPIs . . . 82

D.1 Features in EAPIs . . . 91

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.

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 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 `gentoo-pms@lists.gentoo.org` mailing list. 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.

1.4 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.

1.5 Acknowledgements

Thanks to Mike Kelly (package manager provided utilities, section 12.3), Danny van Dyk (ebuild functions, chapter 9), David Leverton (various sections), Petteri Räty (environment state, section 11.2), Michał Górny (various sections), Andreas K. Hüttel (stable use masking, section 5.2.12), Zac Medico (sub-slots, section 7.2) and James Le Cuirot (build dependencies, section 11.1) for contributions. Thanks also to Mike Frysinger and Brian Harring for proof-reading and suggestions for fixes and/or clarification.

Chapter 2

EAPIs

2.1 Definition

An EAPI can be thought of as a ‘version’ of this specification to which a package conforms. An EAPI value is a string as per section 3.1.9, and is part of an ebuild’s metadata.

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.

EAPIs are also used for profile directories, as described in section 5.2.2.

2.2 Defined EAPIs

This specification defines EAPIs ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, and ‘9’. EAPI ‘0’ is the ‘original’ base EAPI. Each of the later EAPIs contains a number of extensions to its predecessor.

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

2.3 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.

Chapter 3

Names and versions

3.1 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.

3.1.1 Category names

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

3.1.2 Package names

A package name may contain any of the characters [`A-Za-z0-9+_-`]. It must not begin with a hyphen or a plus sign, and must not end in a hyphen followed by anything matching the version syntax described in section 3.2.

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

3.1.3 Slot names

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

3.1.4 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 11.1.1.

Note: Usage of the at-sign is deprecated. It was previously required for `LINGUAS`.

3.1.5 Repository names

A repository name may contain any of the characters [`A-Za-z0-9_-`]. It must not begin with a hyphen. In addition, every repository name must also be a valid package name.

3.1.6 Eclass names

An eclass name may contain any of the characters [`A-Za-z0-9_.-`]. It must begin with a letter or an underscore. In addition, an eclass cannot be named `default`.

3.1.7 License names

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

3.1.8 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.

3.1.9 EAPI names

An EAPI name may contain any of the characters `[A-Za-z0-9+_.-]`. It must not begin with a hyphen, a dot or a plus sign.

3.2 Version specifications

The package manager must neither impose fixed limits upon the number of version components, nor upon the length of any component. Package managers should indicate or reject any version that is invalid according to the rules below.

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

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

This may be followed by zero or more of the suffixes `_alpha`, `_beta`, `_pre`, `_rc` or `_p`, each of which may optionally be followed by an unsigned integer. Suffix and integer count as separate version components.

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

3.3 Version comparison

Version specifications are compared component by component, moving from left to right, as detailed in algorithm 3.1 and sub-algorithms. If a sub-algorithm returns a decision, then that is the result of the whole comparison; if it terminates without returning a decision, the process continues from the point from which it was invoked.

3.4 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`.

Algorithm 3.1 Version comparison top-level logic

1: let A and B be the versions to be compared
2: compare numeric components using algorithm 3.2
3: compare letter components using algorithm 3.4
4: compare suffixes using algorithm 3.5
5: compare revision components using algorithm 3.7
6: **return** $A = B$

Algorithm 3.2 Version comparison logic for numeric components

1: define the notations An_k and Bn_k to mean the k^{th} numeric component of A and B respectively, using 0-based indexing
2: **if** $An_0 > Bn_0$ using integer comparison **then**
3: **return** $A > B$
4: **else if** $An_0 < Bn_0$ using integer comparison **then**
5: **return** $A < B$
6: **end if**
7: let Ann be the number of numeric components of A
8: let Bnn be the number of numeric components of B
9: **for all** i such that $i \geq 1$ and $i < Ann$ and $i < Bnn$, in ascending order **do**
10: compare An_i and Bn_i using algorithm 3.3
11: **end for**
12: **if** $Ann > Bnn$ **then**
13: **return** $A > B$
14: **else if** $Ann < Bnn$ **then**
15: **return** $A < B$
16: **end if**

Algorithm 3.3 Version comparison logic for each numeric component after the first

1: **if** either An_i or Bn_i has a leading `0` **then**
2: let An'_i be An_i with any trailing `0`s removed
3: let Bn'_i be Bn_i with any trailing `0`s removed
4: **if** $An'_i > Bn'_i$ using ASCII stringwise comparison **then**
5: **return** $A > B$
6: **else if** $An'_i < Bn'_i$ using ASCII stringwise comparison **then**
7: **return** $A < B$
8: **end if**
9: **else**
10: **if** $An_i > Bn_i$ using integer comparison **then**
11: **return** $A > B$
12: **else if** $An_i < Bn_i$ using integer comparison **then**
13: **return** $A < B$
14: **end if**
15: **end if**

Algorithm 3.4 Version comparison logic for letter components

1: let Al be the letter component of A if any, otherwise the empty string
2: let Bl be the letter component of B if any, otherwise the empty string
3: **if** $Al > Bl$ using ASCII stringwise comparison **then**
4: **return** $A > B$
5: **else if** $Al < Bl$ using ASCII stringwise comparison **then**
6: **return** $A < B$
7: **end if**

Algorithm 3.5 Version comparison logic for suffixes

define the notations As_k and Bs_k to mean the k^{th} suffix of A and B respectively, using 0-based indexing
let Asn be the number of suffixes of A
let Bsn be the number of suffixes of B
for all i such that $i \geq 0$ and $i < Asn$ and $i < Bsn$, in ascending order **do**
    compare As_i and Bs_i using algorithm 3.6
end for
if $Asn > Bsn$ **then**
    if As_{Bsn} is of type `_p` **then**
        return $A > B$
    else
        return $A < B$
    end if
else if $Asn < Bsn$ **then**
    if Bs_{Asn} is of type `_p` **then**
        return $A < B$
    else
        return $A > B$
    end if
end if

Algorithm 3.6 Version comparison logic for each suffix

if As_i and Bs_i are of the same type (`_alpha` vs `_beta` etc) **then**
    let As'_i be the integer part of As_i if any, otherwise `0`
    let Bs'_i be the integer part of Bs_i if any, otherwise `0`
    if $As'_i > Bs'_i$, using integer comparison **then**
        return $A > B$
    else if $As'_i < Bs'_i$, using integer comparison **then**
        return $A < B$
    end if
else if the type of As_i is greater than the type of Bs_i using the ordering `_alpha` < `_beta` < `_pre` < `_rc` < `_p` **then**
    return $A > B$
else
    return $A < B$
end if

Algorithm 3.7 Version comparison logic for revision components

let Ar be the integer part of the revision component of A if any, otherwise `0`
let Br be the integer part of the revision component of B if any, otherwise `0`
if $Ar > Br$ using integer comparison **then**
    return $A > B$
else if $Ar < Br$ using integer comparison **then**
    return $A < B$
end if

Chapter 4

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.

4.1 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 4.2.
- A `profiles` directory, described in section 4.4.
- A `licenses` directory (optional), described in section 4.5.
- An `eclass` directory (optional), described in section 4.6.
- A `metadata` directory (optional), described in section 4.7.
- 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.

4.2 Category directories

Each category provided by the repository (see also: the `profiles/categories` file, section 4.4) 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 4.3. 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).

4.3 Package directories

A package directory contains the following:

- Zero or more ebuilds. These are as described in chapter 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]. Can be omitted if the file would be empty.
- A `files` directory, containing any support files needed by the ebuilds. Optional.

Any ebuild in a package directory must be named `name-ver.ebuild`, where `name` is the (unqualified) package name, and `ver` is the package's version. 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.

4.4 The profiles directory

The profiles directory shall contain zero or more profile directories as described in chapter 5, 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.

The profiles directory may contain an `eapi` file. This file, if it exists, must contain a single line with the name of an EAPI. This specifies the EAPI to use when handling the profiles directory; a package manager must not attempt to use any repository whose profiles directory requires an EAPI it does not support. If no `eapi` file is present, EAPI 0 shall be used.

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; this does not apply to the `eapi` file.

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.

eapi See above.

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 package dependency specifications (using the directory's EAPI). 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.

PACKAGE-MASK-DIR

For EAPIs listed in table 4.1 as supporting it, `package.mask` can be a directory instead of a regular file. Files contained in that directory, unless their name begins with a dot, will be concatenated in order of their filename in the POSIX locale and the result will be processed as if it were a single file. Any subdirectories will be ignored.

profiles.desc Described below in section 4.4.1.

Table 4.1: EAPIs supporting a directory for `package.mask`

EAPI	`package.mask` can be a directory?
0, 1, 2, 3, 4, 5, 6	No
7, 8, 9	Yes

repo_name Contains, on a single line, the name of this repository. The repository name must conform to section 3.1.5.

thirdpartymirrors Described below in section 4.4.2.

use.desc Contains descriptions of valid global USE flags for this repository. The format is described in section 4.4.3.

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 4.4.3.

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 lower case, whose possible values the file describes. The format of each file is as for `use.desc`, described in section 4.4.3. The `USE_EXPAND` name is *not* included as a prefix here.

updates/ This directory is described in section 4.4.4.

4.4.1 The profiles.desc file

`profiles.desc` is a line-based file, with the standard commenting rules from section 4.4, 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.

4.4.2 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.

Table 4.2: Naming rules for files in `updates` directory for EAPIs

EAPI	Files per quarter year?
0, 1, 2, 3, 4, 5, 6, 7	Yes
8, 9	No

4.4.3 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`.

4.4.4 The updates directory

UPDATES-FILENAMES

The `updates` directory is used to inform the package manager that a package has moved categories, names, or that a version has changed SLOT. For EAPIs so specified by table 4.2, 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`. For other EAPIs, all regular files in this directory will be processed, unless their name begins with a dot.

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`.

It is unspecified in what order the files in the `updates` directory are processed. Lines within each file are processed in ascending order.

At any given time, a name that appears as the origin of a move may not be used as a qualified package name in the repository. A slot that appears as the origin of a slot move may not be used by packages matching the spec of that slot move.

4.5 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 7.3, and will contain the complete text of the license in human-readable form. Plain text format is strongly preferred but not required.

4.6 The eclass directory

The `eclass` directory shall contain copies of the eclasses provided by this repository. The format of these files is described in chapter 10. It may also contain, in their own directory, support files needed by these eclasses.

4.7 The metadata directory

The `metadata` directory contains various repository-level metadata that are 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.

4.7.1 The metadata cache

The `metadata/cache` directory may contain a cached form of all important ebuild metadata variables. The contents of this directory are described in chapter 14.

Chapter 5

Profiles

5.1 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.

5.2 Files that make up a profile

5.2.1 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 make use of line continuations. Blank lines and those beginning with a `#` are discarded.

5.2.2 The eapi file

A profile directory may contain an `eapi` file. This file, if it exists, must contain a single line with the name of an EAPI. This specifies the EAPI to use when handling the directory in question; a package manager must not attempt to use any profile using a directory which requires an EAPI it does not support.

If no `eapi` file is present, the default depends on the EAPI of the top-level profiles directory (see section 4.4). That EAPI shall be used if table 5.1 lists it as "top-level". Otherwise, EAPI 0 shall be used.

PROFILE-EAPI-DEFAULT

The EAPI is neither inherited via the `parent` file nor in subdirectories.

Table 5.1: Default EAPI for profiles

EAPI	Default EAPI?
0, 1, 2, 3, 4, 5, 6, 7, 8	0
9	Top-level

Table 5.2: EAPIs supporting directories for profile files

EAPI	Supports directories for profile files?
0, 1, 2, 3, 4, 5, 6	No
7, 8, 9	Yes

5.2.3 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.

5.2.4 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 5.3.

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_` only. 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. A quoted string may be continued over multiple physical lines by either a simple newline or a backslash-escaped newline.
- Backslashes, except for line continuations, are not allowed.

5.2.5 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. Blank lines and those beginning with a # are discarded.

In EAPIs listed in table 5.2 as supporting directories for profile files, any of the files `package.mask`, `package.use`, `use.*` and `package.use.*` mentioned below can be a directory instead of a regular file. Files contained in that directory, unless their name begins with a dot, will be concatenated in order of their filename in the POSIX locale and the result will be processed as if it were a single file. Any subdirectories will be ignored. PROFILE-FILE-DIRS

5.2.6 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

Table 5.3: EAPIs supporting `package.provided` in profiles

EAPI	Supports `package.provided`?
0, 1, 2, 3, 4, 5, 6	Optionally
7, 8, 9	No

specification prefixed by * denotes that it 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.

5.2.7 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.

5.2.8 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. In some EAPIs, `package.mask` can be a directory instead of a regular file as per section 5.2.5.

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 4.4).

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.

5.2.9 package.provided

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. Supported in EAPIs as per table 5.3.

5.2.10 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 the directory's EAPI. In some EAPIs, `package.use` can be a directory instead of a regular file as per section 5.2.5.

5.2.11 use.stable and package.use.stable

USE-STABLE

The `use.stable` and `package.use.stable` files may be used to override the default USE flags specified by `make.defaults`. They only apply to packages that are merged due to a stable keyword in the sense of section 7.3.3. Each line in `use.stable` contains a USE flag to enable; the `-flag` syntax indicates that the flag should be disabled. The `package.use.stable` file uses the same format as `package.use`. `USE_EXPAND` values may be enabled or disabled by using `expand_name_value`.

Stable restrictions are applied exactly when the following condition holds: If every stable keyword in `KEYWORDS` were replaced with its tilde-prefixed counterpart (see section 7.3.3), then the resulting `KEYWORDS` setting would prevent installation of the package.

Table 5.4: EAPIs supporting `use.stable` and `package.use.stable` in profiles

EAPI	Supports `use.stable`?	Supports `package.use.stable`?
0, 1, 2, 3, 4, 5, 6, 7, 8	No	No
9	Yes	Yes

Table 5.5: Profile directory support for masking/forcing use flags in stable versions only

EAPI	Supports masking/forcing use flags in stable versions?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

If a flag appears in more than one of `package.use`, `use.stable` and `package.use.stable`, then `package.use.stable` takes precedence over `package.use`, which in turn takes precedence over `use.stable`.

These files are supported in EAPIs as per table 5.4. They can be directories instead of regular files as per section 5.2.5.

5.2.12 USE masking and forcing

This section covers the eight files `use.mask`, `use.force`, `use.stable.mask`, `use.stable.force`, `package.use.mask`, `package.use.force`, `package.use.stable.mask`, and `package.use.stable.force`. They are described together because they interact in a non-trivial manner. In some EAPIs, these files can be directories instead of regular files as per section 5.2.5.

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.

In profile directories with an EAPI supporting stable masking, as listed in table 5.5, the same is true for `use.stable.mask`, `use.stable.force`, `package.use.stable.mask` and `package.use.stable.force`. These files, however, only act on packages that are merged due to a stable keyword in the sense of section 7.3.3. Thus, these files can be used to restrict the feature set deemed stable in a package. STABLEMASK

The precise manner in which the eight files 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 5.1.

Stable restrictions ("stable keyword in use" in algorithm 5.1) are applied exactly when the following condition holds: If every stable keyword in `KEYWORDS` were replaced with its tilde-prefixed counterpart (see section 7.3.3), then the resulting `KEYWORDS` setting would prevent installation of the package.

The logic for `use.force`, `use.stable.force`, `package.use.force`, and `package.use.stable.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.

5.3 Profile variables

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

Algorithm 5.1 `USE` masking logic

```
let masked = false
for each profile in the inheritance tree, depth first do
    if use.mask contains flag then
        let masked = true
    else if use.mask contains -flag then
        let masked = false
    end if
    if stable keyword in use then
        if use.stable.mask contains flag then
            let masked = true
        else if use.stable.mask contains -flag then
            let masked = false
        end if
    end if
    for each line in package.use.mask, in order, for which the spec matches package do
        if line contains flag then
            let masked = true
        else if line contains -flag then
            let masked = false
        end if
    end for
    if stable keyword in use then
        for each line in package.use.stable.mask, in order, for which the spec matches package do
            if line contains flag then
                let masked = true
            else if line contains -flag then
                let masked = false
            end if
        end for
    end if
end for
```

Table 5.6: Profile-defined IUSE injection for EAPIs

EAPI	Supports profile-defined IUSE injection?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

Table 5.7: Profile-defined unsetting of variables in EAPIs

EAPI	Supports ENV_UNSET?
0, 1, 2, 3, 4, 5, 6	No
7, 8, 9	Yes

5.3.1 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

If the package manager supports any EAPI listed in table 5.6 as using profile-defined IUSE injection, the following variables must also be treated incrementally; otherwise, the following variables may or may not be treated incrementally:

- IUSE_IMPLICIT
- USE_EXPAND_IMPLICIT
- USE_EXPAND_UNPREFIXED

If the package manager supports any EAPI listed in table 5.7 as using ENV_UNSET, the following variable must also be treated incrementally; otherwise, it may or may not be treated incrementally:

- ENV_UNSET

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.

5.3.2 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 4.4 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 section 13.3.3.

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, exported to the ebuild environment, and whose contents are to be expanded into the USE variable as passed to ebuilds. See section 11.1.1 for details.

USE_EXPAND_UNPREFIXED Similar to `USE_EXPAND`, but no prefix is used. If the repository contains any package using an EAPI supporting profile-defined `IUSE` injection (see table 5.6), this list must contain at least `ARCH`. See section 11.1.1 for details.

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

USE_EXPAND_IMPLICIT, IUSE_IMPLICIT Used to inject implicit values into `IUSE`. See section 11.1.1 for details. `USE_EXPAND_IMPLICIT` contains a subset of names from `USE_EXPAND` and `USE_EXPAND_UNPREFIXED`.

ENV_UNSET Contains a whitespace-delimited list of variables that the package manager shall unset. See section 11.1 for details.

In addition, for EAPIs listed in table 5.6 as supporting profile defined `IUSE` injection, the following variables have special handling as described in section 11.1.1:

- All variables named in `USE_EXPAND` and `USE_EXPAND_UNPREFIXED`.
- All `USE_EXPAND_VALUES_${v}` variables, where `${v}` is a value in `USE_EXPAND_IMPLICIT`.

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 6

Ebuild file format

BASH-VERSION

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 as listed in table 6.1, or any later version. If possible, the package manager should set the shell's compatibility level to the exact version specified. It must ensure that any such compatibility settings (e. g. the `BASH_COMPAT` variable) are not exported to external programs.

The file creation mask (`umask`) is set to `022` in the shell execution environment. It is *not* saved between phase functions but always reset to this initial value.

FAILGLOB

For EAPIs listed such in table 6.1, the `failglob` option of bash is set in the global scope of ebuilds. If set, failed pattern matches during filename expansion result in an error when the ebuild is being sourced.

Name reference variables (introduced in bash version 4.3) must not be used, except in local scope.

The file encoding must be UTF-8 with Unix-style newlines. When sourced, the ebuild must define certain variables and functions (see chapters 7 and 9 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.

Table 6.1: Bash version and options

EAPI	**Bash version**	`failglob` **in global scope?**
0, 1, 2, 3, 4, 5	3.2	No
6, 7	4.2	Yes
8	5.0	Yes
9	5.2	Yes

Chapter 7

Ebuild-defined variables

Note: This chapter 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.

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

7.1 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.

7.2 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.

SLOT The package's slot. Must be a valid slot name, as per section 3.1.3. May be defined by an eclass. Must not be empty.

In EAPIs shown in table 8.7 as supporting sub-slots, the `SLOT` variable may contain an optional sub-slot part that follows the regular slot and is delimited by a `/` character. The sub-slot must be a valid slot name, as per section 3.1.3. The sub-slot is used to represent cases in which an upgrade to a new version of a package with a different sub-slot may require dependent packages to be rebuilt. When the sub-slot part is omitted from the `SLOT` definition, the package is considered to have an implicit sub-slot which is equal to the regular slot.

7.3 Optional ebuild-defined variables

Ebuilds may define any of the following variables. Unless otherwise stated, any of them may be defined by an eclass.

EAPI The EAPI. See below in section 7.3.1.

HOMEPAGE The URI or URIs for a package's homepage, including protocols. See section 8.2 for full syntax.

Table 7.1: EAPIs supporting IUSE defaults

EAPI	Supports IUSE defaults?
0	No
1, 2, 3, 4, 5, 6, 7, 8, 9	Yes

Table 7.2: EAPIs supporting various ebuild-defined variables

EAPI	Supports PROPERTIES?	Supports REQUIRED_USE?
0, 1, 2, 3	Optionally	No
4, 5, 6, 7, 8, 9	Yes	Yes

SRC_URI A list of source URIs for the package. Valid protocols are http://, https://, ftp:// and mirror:// (see section 4.4.2 for mirror behaviour). Fetch restricted packages may include URL parts consisting of just a filename. See section 7.3.2 for description and section 8.2 for full syntax.

LICENSE The package's license. Each text token must be a valid license name, as per section 3.1.7, and must correspond to a tree "licenses/" entry (see section 4.5). See section 8.2 for full syntax.

KEYWORDS A whitespace separated list of keywords for the ebuild. Each token must be a valid keyword name, as per section 3.1.8. See section 7.3.3 for full syntax.

IUSE The USE flags used by the ebuild. 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. See section 11.1.1 for discussion on which values must be listed in this variable.

In EAPIs shown in table 7.1 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. IUSE-DEFAULTS

REQUIRED_USE Zero or more assertions that must be met by the configuration of USE flags to be valid for this ebuild. See section 7.3.4 for description and section 8.2 for full syntax. Only in EAPIs listed in table 7.2 as supporting REQUIRED_USE. REQUIRED-USE

PROPERTIES Zero or more properties for this package. See section 7.3.5 for value meanings and section 8.2 for full syntax. For EAPIs listed in table 7.2 as having optional support, ebuilds must not rely upon the package manager recognising or understanding this variable in any way. PROPERTIES

RESTRICT Zero or more behaviour restrictions for this package. See section 7.3.6 for value meanings and section 8.2 for full syntax.

DEPEND See chapter 8.

BDEPEND See chapter 8.

RDEPEND See chapter 8. For some EAPIs, RDEPEND has special behaviour for its value if unset and when used with an eclass. See section 7.3.7 for details.

PDEPEND See chapter 8.

IDEPEND See chapter 8.

7.3.1 EAPI

An empty or unset EAPI value is equivalent 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

Table 7.3: EAPIs supporting SRC_URI arrows and selective URI restrictions

EAPI	Supports SRC_URI arrows?	Supports selective URI restrictions?
0, 1	No	No
2, 3, 4, 5, 6, 7	Yes	No
8, 9	Yes	Yes

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.

If the EAPI is to be specified in an ebuild, the EAPI variable must be assigned to precisely once. The assignment must not be preceded by any lines other than blank lines or those that start with optional whitespace (spaces or tabs) followed by a # character, and the line containing the assignment statement must match the following regular expression:

```
^[ \t]*EAPI=(['"]?)([A-Za-z0-9+_.-]*)\1[ \t]*([ \t]#.*)?$
```

The package manager must determine the EAPI of an ebuild by parsing its first non-blank and non-comment line, using the above regular expression. If it matches, the EAPI is the substring matched by the capturing parentheses (0 if empty), otherwise it is 0. For a recognised EAPI, the package manager must make sure that the EAPI value obtained by sourcing the ebuild with bash is identical to the EAPI obtained by parsing. The ebuild must be treated as invalid if these values are different.

Eclasses must not attempt to modify the EAPI variable.

7.3.2 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 4.4.2 for mirror details.

The RESTRICT metadata key can be used to impose additional restrictions upon downloading—see section 7.3.6 for details. Fetch restricted packages may use a simple filename instead of a full URI.

In EAPIs listed in table 7.3 as supporting arrows, if an arrow is used, the filename used when saving to DISTDIR shall instead be the name on the right of the arrow. When consulting mirrors (except for those explicitly listed on the left of the arrow, if mirror:// is used), the filename to the right of the arrow shall be requested instead of the filename in the URI. SRC-URI-ARROWS

In EAPIs listed in table 7.3 as supporting selective URI restrictions, the URI protocol can be prefixed by an additional fetch+ or mirror+ term. If the ebuild is fetch restricted, the fetch+ prefix undoes the fetch restriction for the URI (but not the implied mirror restriction). If the ebuild is fetch or mirror restricted, the mirror+ prefix undoes both fetch and mirror restrictions for the URI. URI-RESTRICT

7.3.3 Keywords

Keywords are used to indicate levels of stability of a package on a respective architecture arch. The following conventions are used:

- arch: Both the package version and the ebuild are widely tested, known to work and not have any serious issues on the indicated platform. This is referred to as a *stable keyword*.

- `~arch`: The package version and the ebuild are believed to work and do not have any known serious bugs, but more testing is required before the package version is considered suitable for obtaining a stable keyword. This is referred to as an *unstable keyword* or a *testing keyword*.
- No keyword: It is not known whether the package will work, or insufficient testing has occurred.
- `-arch`: The package version will not work on the architecture.

The `-*` keyword is used to indicate package versions which are not worth trying to test on unlisted architectures.

An empty `KEYWORDS` variable indicates uncertain functionality on any architecture.

7.3.4 USE state constraints

`REQUIRED_USE` contains a list of assertions that must be met by the configuration of `USE` flags to be valid for this ebuild. In order to be matched, a `USE` flag in a terminal element must be enabled (or disabled if it has an exclamation mark prefix).

If the package manager encounters a package version where `REQUIRED_USE` assertions are not met, it must treat this package version as if it was masked. No phase functions must be called.

It is an error for a flag to be used if it is not included in `IUSE_EFFECTIVE`.

7.3.5 Properties

The following tokens are permitted inside `PROPERTIES`:

interactive The package may require interaction with the user via the tty.

live The package uses "live" source code that may vary each time that the package is installed.

test_network The package manager may run tests that require an internet connection, even if the ebuild has `RESTRICT=test`.

test_privileged The package manager may run tests that require superuser privileges, even if the ebuild has `RESTRICT=test`.

Package managers may recognise other tokens. Ebuilds may not rely upon any token being supported.

7.3.6 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. Implies `mirror`.

strip No stripping of debug symbols from files to be installed may be performed. In EAPIs listed in table 12.19 as supporting controllable stripping, this behaviour may be altered by the `dostrip` command.

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

test The `src_test` phase must not be run.

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

7.3.7 RDEPEND value

RDEPEND-DEPEND

In EAPIs listed in table 7.4 as having `RDEPEND=DEPEND`, if `RDEPEND` is unset (but not if it is set to an empty string) in an ebuild, when generating metadata the package manager must treat its value as being equal to the value of `DEPEND`.

Table 7.4: EAPIs with RDEPEND=DEPEND default

EAPI	RDEPEND=DEPEND?
0, 1, 2, 3	Yes
4, 5, 6, 7, 8, 9	No

Table 7.5: EAPIs supporting DEFINED_PHASES

EAPI	Supports DEFINED_PHASES?
0, 1, 2, 3	Optionally
4, 5, 6, 7, 8, 9	Yes

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 treated as being part of RDEPEND.

7.4 Magic ebuild-defined variables

The following variables must be defined by `inherit` (see section 10.1), 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 7.1, `inherit` may not be used conditionally, except upon constant conditions.

The following is a special variable defined by the package manager for internal use and may or may not be available in the ebuild environment:

DEFINED_PHASES A space separated arbitrarily ordered list of phase names (e. g. `configure setup unpack`) whose phase functions are defined by the ebuild or an eclass inherited by the ebuild. If no phase functions are defined, a single hyphen is used instead of an empty string. For EAPIs listed in table 7.5 as having optional DEFINED_PHASES support, package managers may not rely upon the metadata cache having this variable defined, and must treat an empty string as "this information is not available". DEFINED-PHASES

Note: Thus, by extension of section 7.1, phase functions must not be defined based upon any variant condition.

For EAPIs listed in table 11.1 with the property that variables are not exported, the package manager must not export any of the variables specified in this section to the environment.

Chapter 8

Dependencies

8.1 Dependency classes

There are three classes of dependencies supported by ebuilds:

- Build dependencies (DEPEND). These must be installed and usable before the pkg_setup phase function is executed as a part of source build and throughout all src_* phase functions executed as part of that build. These may not be installed at all if a binary package is being merged.
- Runtime dependencies (RDEPEND). These must be installed and usable before the results of an ebuild merging are treated as usable.
- Post dependencies (PDEPEND). These must be installed at some point before the package manager finishes the batch of installs.

Additionally, in EAPIs listed in table 8.4 as supporting BDEPEND, the build dependencies are split into two subclasses: BDEPEND

- BDEPEND build dependencies that are binary compatible with the native build system (CBUILD). The ebuild is allowed to call binary executables installed by this kind of dependency.
- DEPEND build dependencies that are binary compatible with the system being built (CHOST). The ebuild must not execute binary executables installed by this kind of dependency.

Additionally, in EAPIs listed in table 8.4 as supporting IDEPEND, install-time dependencies can be specified. These dependencies are binary compatible with the native build system (CBUILD). Ebuilds are allowed to call them in pkg_preinst and pkg_postinst. Ebuilds may also call them in pkg_prerm and pkg_postrm but must not rely on them being available. IDEPEND

Table 8.1: Dependency classes required to be satisfied for a particular phase function

Phase function	Satisfied dependency classes
pkg_pretend, pkg_info, pkg_nofetch	None (ebuilds can rely only on the packages in the system set)
pkg_setup	Same as src_unpack if executed as part of source build, same as pkg_pretend otherwise
src_unpack, src_prepare, src_configure, src_compile, src_test, src_install	DEPEND, BDEPEND
pkg_preinst, pkg_postinst, pkg_prerm, pkg_postrm	RDEPEND, IDEPEND
pkg_config	RDEPEND, PDEPEND

Table 8.2: Summary of other interfaces related to dependency classes

	BDEPEND, IDEPEND	DEPEND	RDEPEND, PDEPEND
Binary compatible with	CBUILD	CHOST	CHOST
Base unprefixed path	/	${SYSROOT}	${ROOT}
Relevant offset-prefix	${BROOT}	See table 8.3	${EPREFIX}
Path combined with prefix	${BROOT}	${ESYSROOT}	${EROOT}
PM query command option	-b	-d	-r

Table 8.3: Prefix values for DEPEND

If SYSROOT is:	${ROOT}	Empty, and ROOT is non-empty	Other
Then offset-prefix is:	${EPREFIX}	${BROOT}	Empty
And ESYSROOT is:	${EROOT}	${BROOT}	${SYSROOT}

Table 8.4: EAPIs supporting additional dependency types

EAPI	Supports BDEPEND?	Supports IDEPEND?
0, 1, 2, 3, 4, 5, 6	No	No
7	Yes	No
8, 9	Yes	Yes

Table 8.1 lists dependencies which must be satisfied before a particular phase function is executed. Table 8.2 summarises additional interfaces related to the dependency classes.

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

8.2 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, BDEPEND, RDEPEND, PDEPEND, IDEPEND.
- A URI, in the form proto://host/path. Permitted in HOMEPAGE and SRC_URI. In EAPIs listed in table 7.3 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 7.3.2.
- A flat filename. Permitted in SRC_URI.
- A license name (e. g. GPL-2). Permitted in LICENSE.
- A use flag name, optionally preceded by an exclamation mark. Permitted in REQUIRED_USE.
- A simple string. Permitted in PROPERTIES and RESTRICT.
- An all-of group, which consists of an open parenthesis, followed by whitespace, followed by one 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 one 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, BDEPEND, RDEPEND, PDEPEND, IDEPEND, LICENSE, REQUIRED_USE.
- An exactly-one-of group, which has the same format as the any-of group, but begins with the string ^^ instead. Permitted in REQUIRED_USE.

Table 8.5: EAPIs supporting `REQUIRED_USE ??` groups

EAPI	Supports `REQUIRED_USE ??` groups?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

Table 8.6: Matching of empty dependency groups in EAPIs

EAPI	Empty `||` and `^^` groups are matched?
0, 1, 2, 3, 4, 5, 6	Yes
7, 8, 9	No

- An at-most-one-of group, which has the same format as the any-of group, but begins with the string `??` instead. Permitted in `REQUIRED_USE` in EAPIs listed in table 8.5 as supporting `REQUIRED_USE ??` groups. AT-MOST-ONE-OF
- 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 one 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.

In particular, note that whitespace is not optional.

8.2.1 All-of dependency specifications

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

8.2.2 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.

It is an error for a flag to be used if it is not included in `IUSE_EFFECTIVE` as described in section 11.1.1.

8.2.3 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.

In EAPIs specified in table 8.6, an empty any-of group counts as being matched. EMPTY-DEP-GROUPS

8.2.4 Exactly-one-of dependency specifications

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

In an exactly-one-of group, exactly one immediate child element must be matched.

In EAPIs specified in table 8.6, an empty exactly-one-of group counts as being matched.

Table 8.7: Support for SLOT dependencies and sub-slots in EAPIs

EAPI	Supports SLOT dependencies?	Supports sub-slots?
0	No	No
1, 2, 3, 4	Named only	No
5, 6, 7, 8, 9	Named and operator	Yes

Table 8.8: EAPIs supporting USE dependencies

EAPI	Supports USE dependencies?
0, 1	No
2, 3	2-style
4, 5, 6, 7, 8, 9	4-style

8.2.5 At-most-one-of dependency specifications

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

In an at-most-one-of group, at most one immediate child element must be matched.

8.3 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 8.3.1, followed immediately by `category/package`, followed by a hyphen, followed by a version specification.

In EAPIs shown in table 8.7 as supporting SLOT dependencies, either of the above formats may additionally be suffixed by a `:slot` restriction, as described in section 8.3.3. A package manager must warn or error if slot dependencies are used with an EAPI not supporting SLOT dependencies.

In EAPIs shown in table 8.8 as supporting 2-style or 4-style USE dependencies, a specification may additionally be suffixed by at most one 2-style or 4-style `[use]` restriction, as described in section 8.3.4. A package manager must warn or error if this feature is used with an EAPI not supporting use dependencies. USE-DEPS

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

8.3.1 Operators

The following operators are available:

< Strictly less than the specified version.

<= 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, then only the given number of version components is used for comparison, i. e. the asterisk acts as a wildcard for any further components. 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 when revision parts are ignored.

>= Greater than or equal to the specified version.

Table 8.9: Exclamation mark strengths for EAPIs

EAPI	!	!!
0, 1	Unspecified	Forbidden
2, 3, 4, 5, 6, 7, 8, 9	Weak	Strong

> Strictly greater than the specified version.

8.3.2 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. As an exception, weak blocks on the package version of 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 8.9. BANG-STRENGTH

8.3.3 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. SLOT-DEPS

In EAPIs shown in table 8.7 as supporting sub-slots, a slot dependency may contain an optional sub-slot part that follows the regular slot and is delimited by a `/` character. SUB-SLOT

An operator slot dependency consists of a colon followed by one of the following operators: SLOT-OPERATOR-DEPS

***** Indicates that any slot value is acceptable. In addition, for runtime dependencies, indicates that the package will not break if the matched package is uninstalled and replaced by a different matching package in a different slot.

= Indicates that any slot value is acceptable. In addition, for runtime dependencies, indicates that the package will break unless a matching package with slot and sub-slot equal to the slot and sub-slot of the best version installed as a build-time (`DEPEND`) dependency is available.

slot= Indicates that only a specific slot value is acceptable, and otherwise behaves identically to the = operator. The specified slot must not contain a sub-slot part.

To implement the equals slot operators = and *slot*=, the package manager will need to store the slot/sub-slot pair of the best installed version of the matching package. This syntax is only for package manager use and must not be used by ebuilds. The package manager may do this by inserting the appropriate slot/sub-slot pair between the colon and equals sign when saving the package's dependencies. The sub-slot part must not be omitted here (when the `SLOT` variable omits the sub-slot part, the package is considered to have an implicit sub-slot which is equal to the regular slot).

Whenever the equals slot operator is used in an enabled dependency group, the dependencies (`DEPEND`) must ensure that a matching package is installed at build time. It is invalid to use the equals slot operator inside `PDEPEND` or inside any-of dependency specifications.

8.3.4 2-style and 4-style USE dependencies

A 2-style or 4-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.

USE-DEP-DEFAULTS

In a 4-style use dependency, the flag name may immediately be followed by a *default* specified by either `(+)` or `(-)`. The former indicates that, when applying the use dependency to a package that does not have the flag in question in `IUSE_REFERENCEABLE`, the package manager shall behave as if the flag were present and enabled; the latter, present and disabled.

Unless a 4-style default is specified, it is an error for a use dependency to be applied to an ebuild which does not have the flag in question in `IUSE_REFERENCEABLE`.

Note: By extension of the above, a default that could reference an ebuild using an EAPI not supporting profile `IUSE` injections cannot rely upon any particular behaviour for flags that would not have to be part of `IUSE`.

It is an error for an ebuild to use a conditional use dependency when that ebuild does not have the flag in `IUSE_EFFECTIVE`.

Chapter 9

Ebuild-defined functions

9.1 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. All functions may assume that they have read access to all system libraries, binaries and configuration files that are accessible to normal users, as well as write access to the temporary directories specified by the `T`, `TMPDIR` and `HOME` variables (see section 11.1). Most functions must assume only that they have additional write access to the package's working directory (the `WORKDIR` variable); exceptions are noted below.

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.

9.1.1 Initial working directories

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, then for EAPIs listed in table 9.1 as having an empty directory, it must be set to a dedicated directory that is empty at the start of the function and may be read-only. For other EAPIs, it may be set to anything. The ebuild must not rely upon a particular location for it. The ebuild *may* assume that the initial working directory for any phase is a trusted location that may only be written to by a privileged user and group.

PHASE-FUNCTION-DIR

Some functions are described as having an initial working directory of `S` with an error or fallback to `WORKDIR`. For EAPIs listed in table 9.2 as having the fallback, this means that if `S` is not a directory before the start of the phase function, the initial working directory shall be `WORKDIR` instead. For EAPIs where it is a conditional error, if `S` is not a directory before the start of the phase function, it is a fatal error, unless all of the following conditions are true, in which case the fallback to `WORKDIR` is used:

S-WORKDIR-FALLBACK

- The `A` variable contains no items.
- The phase function in question is not in `DEFINED_PHASES`.
- None of the phase functions `unpack`, `prepare`, `configure`, `compile`, `test` or `install`, if supported by the EAPI in question and occurring prior to the phase about to be executed, are in `DEFINED_PHASES`.

9.1.2 pkg_pretend

The `pkg_pretend` function is only called for EAPIs listed in table 9.3 as supporting it.

PKG-PRETEND

Table 9.1: Initial working directory in `pkg_*` phase functions for EAPIs

EAPI	Initial working directory?
0, 1, 2, 3, 4, 5, 6, 7	Any
8, 9	Empty

Table 9.2: EAPIs with `S` to `WORKDIR` fallbacks

EAPI	Fallback to `WORKDIR` permitted?
0, 1, 2, 3	Always
4, 5, 6, 7, 8, 9	Conditional error

Table 9.3: EAPIs supporting `pkg_pretend`

EAPI	Supports `pkg_pretend`?
0, 1, 2, 3	No
4, 5, 6, 7, 8, 9	Yes

The `pkg_pretend` function may be used to carry out sanity checks early on in the install process. For example, if an ebuild requires a particular kernel configuration, it may perform that check in `pkg_pretend` and call `eerror` and then `die` with appropriate messages if the requirement is not met.

`pkg_pretend` is run separately from the main phase function sequence, and does not participate in any kind of environment saving. There is no guarantee that any of an ebuild's dependencies will be met at this stage, and no guarantee that the system state will not have changed substantially before the next phase is executed.

`pkg_pretend` must not write to the filesystem.

9.1.3 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.

9.1.4 src_unpack

The `src_unpack` function extracts all of the package's sources. In EAPIs lacking `src_prepare`, it may also apply patches and set 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 in listing 9.1.

Listing 9.1 `src_unpack`

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

Table 9.4: src_prepare support and behaviour for EAPIs

EAPI	Supports src_prepare?	Format
0, 1	No	Not applicable
2, 3, 4, 5	Yes	no-op
6, 7	Yes	6
8, 9	Yes	8

Table 9.5: EAPIs supporting src_configure

EAPI	Supports src_configure?
0, 1	No
2, 3, 4, 5, 6, 7, 8, 9	Yes

9.1.5 src_prepare

The src_prepare function is only called for EAPIs listed in table 9.4 as supporting it. The src_prepare function can be used for post-unpack source preparation. SRC-PREPARE

The initial working directory is S, with an error or fallback to WORKDIR as discussed in section 9.1.1.

For EAPIs listed in table 9.4 as using format 6 or 8, the default implementation used when the ebuild lacks the src_prepare function shall behave as in listing 9.2 or listing 9.3, respectively.

For other EAPIs supporting src_prepare, the default implementation used when the ebuild lacks the src_prepare function is a no-op.

Listing 9.2 src_prepare, format 6

```
src_prepare() {
    if [[ $(declare -p PATCHES 2>/dev/null) == "declare -a"* ]]; then
        [[ -n ${PATCHES[@]} ]] && eapply "${PATCHES[@]}"
    else
        [[ -n ${PATCHES} ]] && eapply ${PATCHES}
    fi
    eapply_user
}
```

Listing 9.3 src_prepare, format 8

```
src_prepare() {
    if [[ ${PATCHES@a} == *a* ]]; then
        [[ -n ${PATCHES[@]} ]] && eapply -- "${PATCHES[@]}"
    else
        [[ -n ${PATCHES} ]] && eapply -- ${PATCHES}
    fi
    eapply_user
}
```

9.1.6 src_configure

The src_configure function is only called for EAPIs listed in table 9.5 as supporting it. SRC-CONFIGURE

The initial working directory is S, with an error or fallback to WORKDIR as discussed in section 9.1.1.

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 in listing 9.4.

Table 9.6: `src_compile` behaviour for EAPIs

EAPI	Format
0	0
1	1
2, 3, 4, 5, 6, 7, 8, 9	2

Listing 9.4 `src_configure`

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

9.1.7 src_compile

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

The initial working directory is `S`, with an error or fallback to `WORKDIR` as discussed in section 9.1.1.

For EAPIs listed in table 9.6 as using format 0, 1 or 2, the default implementation used when the ebuild lacks the `src_prepare` function shall behave as in listing 9.5, listing 9.6 or listing 9.7, respectively.

Listing 9.5 `src_compile`, format 0

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

Listing 9.6 `src_compile`, format 1

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

Listing 9.7 `src_compile`, format 2

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

Table 9.7: `src_test` behaviour for EAPIs

EAPI	Supports parallel tests?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

Table 9.8: `src_install` behaviour for EAPIs

EAPI	Format
0, 1, 2, 3	no-op
4, 5	4
6, 7, 8, 9	6

9.1.8 src_test

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

The initial working directory is `S`, with an error or fallback to `WORKDIR` as discussed in section 9.1.1.

The default implementation used when the ebuild lacks the `src_test` function must, if tests are enabled, run `emake check` if and only if such a target is available, or if not run `emake test` if and only if such a target is available. In both cases, if `emake` returns non-zero the build must be aborted.

For EAPIs listed in table 9.7 as not supporting parallel tests, the `emake` command must be called with option `-j1`. PARALLEL-TESTS

The `src_test` function may be disabled by `RESTRICT`. See section 7.3.6. It may be disabled by user too, using a PM-specific mechanism.

9.1.9 src_install

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

The initial working directory is `S`, with an error or fallback to `WORKDIR` as discussed in section 9.1.1.

For EAPIs listed in table 9.8 as using format 4 or 6, the default implementation used when the ebuild lacks the `src_prepare` function shall behave as in listing 9.8 or listing 9.9, respectively.

For other EAPIs, the default implementation used when the ebuild lacks the `src_install` function is a no-op.

Listing 9.8 `src_install`, format 4

```
src_install() {
    if [[ -f Makefile ]] || [[ -f GNUmakefile ]] || [[ -f makefile ]]; then
        emake DESTDIR="${D}" install
    fi

    if ! declare -p DOCS >/dev/null 2>&1; then
        local d
        for d in README* ChangeLog AUTHORS NEWS TODO CHANGES \
                 THANKS BUGS FAQ CREDITS CHANGELOG; do
            [[ -s "${d}" ]] && dodoc "${d}"
        done
    elif [[ $(declare -p DOCS) == "declare -a"* ]]; then
        dodoc "${DOCS[@]}"
    else
        dodoc ${DOCS}
    fi
}
```

Listing 9.9 `src_install`, format 6

```
src_install() {
    if [[ -f Makefile ]] || [[ -f GNUmakefile ]] || [[ -f makefile ]]; then
        emake DESTDIR="${D}" install
    fi
    einstalldocs
}
```

9.1.10 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` variables.

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

9.1.11 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` variable.

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

9.1.12 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` variable.

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

9.1.13 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` variable.

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

9.1.14 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`.

9.1.15 pkg_info

The `pkg_info` function may be called by the package manager when displaying information about an installed package. In EAPIs listed in table 9.9 as supporting `pkg_info` on non-installed packages, it may also be called by the package manager when displaying information about a non-installed package. In this case, ebuild authors should note that dependencies may not be installed.

PKG-INFO

`pkg_info` must not write to the filesystem.

Table 9.9: EAPIs supporting `pkg_info` on non-installed packages

EAPI	Supports `pkg_info` on non-installed packages?
0, 1, 2, 3	No
4, 5, 6, 7, 8, 9	Yes

Table 9.10: EAPIs supporting `default_` phase functions

EAPI	Supports `default_` functions in phases
0, 1	None
2, 3	`pkg_nofetch`, `src_unpack`, `src_prepare`, `src_configure`, `src_compile`, `src_test`
4, 5, 6, 7, 8, 9	`pkg_nofetch`, `src_unpack`, `src_prepare`, `src_configure`, `src_compile`, `src_test`, `src_install`

9.1.16 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.

9.1.17 Default phase functions

DEFAULT-PHASE-FUNCS

In EAPIs listed in table 9.10 as supporting `default_` phase functions, a function named `default_<phase-function>`) that behaves as the default implementation for that EAPI shall be defined when executing any ebuild phase function listed in the table. Ebuilds must not call these functions except when in the phase in question.

9.2 Call order

The call order for installing a package is:

- `pkg_pretend` (only for EAPIs listed in table 9.3), which is called outside of the normal call order process.
- `pkg_setup`
- `src_unpack`
- `src_prepare` (only for EAPIs listed in table 9.4)
- `src_configure` (only for EAPIs listed in table 9.5)
- `src_compile`
- `src_test` (except if `RESTRICT=test` or disabled by user)
- `src_install`
- `pkg_preinst`
- `pkg_postinst`

The call order for uninstalling a package is:

- `pkg_prerm`
- `pkg_postrm`

The call order for upgrading, downgrading or reinstalling a package is:

- `pkg_pretend` (only for EAPIs listed in table 9.3), which is called outside of the normal call order process.
- `pkg_setup`
- `src_unpack`
- `src_prepare` (only for EAPIs listed in table 9.4)

- `src_configure` (only for EAPIs listed in table 9.5)
- `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`

Note: When up- or downgrading a package in EAPI 0 or 1, the last four phase functions can alternatively be called in the order `pkg_preinst`, `pkg_postinst`, `pkg_prerm`, `pkg_postrm`. This behaviour is deprecated.

The `pkg_config`, `pkg_info` and `pkg_nofetch` functions are not called in a normal sequence. The `pkg_pretend` function is called some unspecified time before a (possibly hypothetical) 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 10

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 4.6. 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. `<name>` must be a valid eclass name, as per section 3.1.6.

10.1 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.

10.2 Eclass-defined metadata keys

ACCUMULATE-VARS

The `IUSE`, `REQUIRED_USE`, `DEPEND`, `BDEPEND`, `RDEPEND`, `PDEPEND` and `IDEPEND` 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. For EAPIs listed in table 10.1 as accumulating `PROPERTIES` and `RESTRICT`, the same is true for these variables. 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 7.3.7 are applied.

10.3 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

Table 10.1: EAPIs accumulating PROPERTIES and RESTRICT across eclasses

EAPI	Accumulates PROPERTIES?	Accumulates RESTRICT?
0, 1, 2, 3, 4, 5, 6, 7	No	No
8, 9	Yes	Yes

Listing 10.1 EXPORT_FUNCTIONS example: foo.eclass

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

EXPORT_FUNCTIONS src_compile
```

that an ebuild inherits 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 10.1 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 <eclass>_<phase-function>, where <eclass> is the name of the eclass.

If EXPORT_FUNCTIONS is called multiple times for the same phase function, the last call takes precedence. Eclasses may not rely upon any particular behaviour if they inherit another eclass after calling EXPORT_FUNCTIONS.

Chapter 11

The ebuild environment

11.1 Defined variables

The package manager must define the following variables. Not all variables are universally meaningful; variables that are not meaningful in a given phase or in global scope 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 section 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.

These variables are either exported to the environment or kept as unexported shell variables, as specified for EAPIs in table 11.1; exceptions are `TMPDIR` and `HOME` which are always exported. In EAPIs where variables are not exported, the package manager must pass those that are required by ebuild-specific external commands (see section 12.3) in an implementation-defined manner.

EXPORT-VARS

Variables listed in section 5.3.2 as having specific meanings or special handling, and that are set in the active profiles' `make.defaults` files, obey the same export rules specified for EAPIs in table 11.1. To clarify, this behaviour is governed by the EAPI of the ebuild, not that of the profile. Except where otherwise noted, all other variables set in the active profiles' `make.defaults` files must be exported to the environment.

Table 11.1: EAPIs with variables exported to the environment

EAPI	Variables exported?
0, 1, 2, 3, 4, 5, 6, 7, 8	Yes
9	No

Table 11.2: Defined variables

Variable	Legal in	Consistent?	Description
P	All	No[1]	Package name and version, without the revision part. For example, vim-7.0.174.
PF	All	No[1]	Package name, version, and revision (if any), for example vim-7.0.174-r1.
PN	All	No[1]	Package name, for example vim.
CATEGORY	All	No[1]	The package's category, for example app-editors.
PV	All	Yes	Package version, with no revision. For example 7.0.174.
PR	All	Yes	Package revision, or r0 if none exists.
PVR	All	Yes	Package version and revision (if any), for example 7.0.174 or 7.0.174-r1.
A	src_*, pkg_nofetch	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 SRC_URI. Does not include any that are disabled because of USE conditionals. The value is calculated from the base names of each element of the SRC_URI ebuild metadata variable.
AA	AA[2]	src_*, pkg_nofetch	Yes	All source files that could be available for the package, including any that are disabled in A because of USE conditionals. The value is calculated from the base names of each element of the SRC_URI ebuild metadata variable. Only for EAPIs listed in table 11.4 as supporting AA.
FILESDIR	src_*, global scope[3]	Yes	The full path to a directory where the files from the package's files directory (used for small support files or patches) are available. See section 4.3. 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.
DISTDIR	src_*, global scope[3]	Yes	The full path to the directory in which the files in the A variable are stored.
WORKDIR	src_*, global scope[3]	Yes	The full path to the ebuild's working directory, where all build data should be contained.

[1] May change if a package has been updated (see section 4.4.4).
[2] This variable is generally considered deprecated. However, ebuilds must still assume that the package manager sets it in the EAPIs supporting it. For example, a few configure scripts use this variable to find the aalib package; ebuilds calling such configure scripts must thus work around this.
[3] Not necessarily present when installing from a binary package. Ebuilds must not access the directory in global scope.

	Variable	Legal in	Consistent?	Description
	S	src_*	Yes	The full path to the temporary build directory, used by src_compile, src_install etc. Defaults to ${WORKDIR}/${P}. May be modified by ebuilds. If S is assigned in the global scope of an ebuild, then the restrictions of section 11.2 for global variables apply.
PORTDIR	PORTDIR	src_*	No	The full path to the master repository's base directory. Only for EAPIs listed in table 11.4 as supporting PORTDIR.
ECLASSDIR	ECLASSDIR	src_*	No	The full path to the master repository's eclass directory. Only for EAPIs listed in table 11.4 as supporting ECLASSDIR.
	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. The presence of a trailing slash is EAPI dependent as listed in table 11.8.
	EROOT	pkg_*	No	Contains the concatenation of the paths in the ROOT and EPREFIX variables, for convenience. See also the EPREFIX variable. Only for EAPIs listed in table 11.5 as supporting EROOT. The presence of a trailing slash is EAPI dependent as listed in table 11.8.
SYSROOT	SYSROOT	src_*, pkg_setup[4]	No	The absolute path to the root directory containing build dependencies satisfied by DEPEND. Only for EAPIs listed in table 11.3 as supporting SYSROOT.
	ESYSROOT	src_*, pkg_setup[4]	No	Contains the concatenation of the SYSROOT path and applicable prefix value, as determined by table 8.3. Only for EAPIs listed in table 11.5 as supporting ESYSROOT.
BROOT	BROOT	src_*, pkg_setup,[4] pkg_preinst, pkg_postinst, pkg_prerm, pkg_postrm[5]	No	The absolute path to the root directory containing build dependencies satisfied by BDEPEND and IDEPEND, typically executable build tools. This includes any applicable offset prefix. Only for EAPIs listed in table 11.3 as supporting BROOT.
	T	All	Partially[6]	The full path to a temporary directory for use by the ebuild.

[4]Not necessarily present when installing from a binary package.

[5]Legal in pkg_*inst and pkg_*rm only for EAPIs listed in table 8.4 as supporting IDEPEND.

[6]Consistent 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.

Variable	Legal in	Consistent?	Description
TMPDIR	All	Partially[6]	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. TMPDIR is always exported to the environment.
HOME	All	Partially[6]	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. HOME is always exported to the environment.
EPREFIX	All	Yes	The normalised offset-prefix path of an offset installation. When EPREFIX is not set in the calling environment, EPREFIX defaults to the built-in offset-prefix that was set during installation of the package manager. When a different EPREFIX value than the built-in value is set in the calling environment, a cross-prefix build is performed where using the existing utilities, a package is built for the given EPREFIX, akin to ROOT. See also section 11.1.3. Only for EAPIs listed in table 11.5 as supporting EPREFIX.
D	src_*	Yes	Contains the full path to the image directory into which the package should be installed. Ebuilds must not attempt to access the directory in src_* phases other than src_install. The presence of a trailing slash is EAPI dependent as listed in table 11.8.
D (continued)	pkg_preinst	No[7]	Contains the full path to the image directory of the package about to be merged. The presence of a trailing slash is EAPI dependent as listed in table 11.8.
ED	src_*, pkg_preinst	See D	Contains the concatenation of the paths in the D and EPREFIX variables, for convenience. See also the EPREFIX variable. Only for EAPIs listed in table 11.5 as supporting ED. Ebuilds must not attempt to access the directory in src_* phases other than src_install. The presence of a trailing slash is EAPI dependent as listed in table 11.8.
DESTTREE	DESTTREE	src_install	No	Controls the location where dobin, dolib, domo, and dosbin install things. Only for EAPIs listed in table 11.4 as supporting DESTTREE. In all other EAPIs, this is retained as a conceptual variable inaccessible from the ebuild environment.
INSDESTTREE	INSDESTTREE	src_install	No	Controls the location where doins installs things. Only for EAPIs listed in table 11.4 as supporting INSDESTTREE. In all other EAPIs, this is retained as a conceptual variable inaccessible from the ebuild environment.
USE	All	Yes	A whitespace-delimited list of all active USE flags for this ebuild. See section 11.1.1 for details.

[7] May not necessarily have the same value that it had in the src_* phases.

	Variable	Legal in	Consistent?	Description
	EBUILD_PHASE	src_*, pkg_*	No	Takes one of the values config, setup, nofetch, unpack, prepare, configure, compile, test, install, preinst, postinst, prerm, postrm, info, pretend according to the top level ebuild function that was executed by the package manager. Behaviour is unspecified when the ebuild is being sourced for other (e. g. metadata or QA) purposes.
EBUILD-PHASE-FUNC	EBUILD_PHASE_FUNC	src_*, pkg_*	No	Takes one of the values pkg_config, pkg_setup, pkg_nofetch, src_unpack, src_prepare, src_configure, src_compile, src_test, src_install, pkg_preinst, pkg_postinst, pkg_prerm, pkg_postrm, pkg_info, pkg_pretend according to the top level ebuild function that was executed by the package manager. Behaviour is unspecified when the ebuild is being sourced for other (e. g. metadata or QA) purposes. Only for EAPIs listed in table 11.3 as supporting EBUILD_PHASE_FUNC.
KV	KV	All	Yes	The version of the running kernel at the time the ebuild was first executed, as returned by the uname -r command or equivalent. May be modified by ebuilds. Only for EAPIs listed in table 11.4 as supporting KV.
MERGE-TYPE	MERGE_TYPE	pkg_*	No	The type of package that is being merged. Possible values are: source if building and installing a package from source, binary if installing a binary package, and buildonly if building a binary package without installing it. Only for EAPIs listed in table 11.3 as supporting MERGE_TYPE.
	REPLACING_VERSIONS	pkg_preinst, pkg_postinst (pkg_pretend, pkg_setup)	Yes	A list of all versions of this package (including revision, if specified), whitespace separated with no leading or trailing whitespace, that are being replaced (uninstalled or overwritten) as a result of this install. See section 11.1.2, especially for the phases in which the variable is legal. Only for EAPIs listed in table 11.3 as supporting REPLACING_VERSIONS.
	REPLACED_BY_VERSION	pkg_prerm, pkg_postrm	Yes	The single version of this package (including revision, if specified) that is replacing us, if we are being uninstalled as part of an install, or an empty string otherwise. See section 11.1.2. Only for EAPIs listed in table 11.3 as supporting REPLACED_BY_VERSION.

Table 11.3: EAPIs supporting various added env variables

EAPI	MERGE_ TYPE?	REPLACING_ VERSIONS?	REPLACED_ BY_VERSION?	EBUILD_ PHASE_FUNC?	SYSROOT?	BROOT?
0, 1, 2, 3	No	No	No	No	No	No
4	Yes	Yes	Yes	No	No	No
5, 6	Yes	Yes	Yes	Yes	No	No
7, 8, 9	Yes	Yes	Yes	Yes	Yes	Yes

Table 11.4: EAPIs supporting various removed env variables

EAPI	AA?	KV?	PORTDIR?	ECLASSDIR?	DESTTREE?	INSDESTTREE?
0, 1, 2, 3	Yes	Yes	Yes	Yes	Yes	Yes
4, 5, 6	No	No	Yes	Yes	Yes	Yes
7, 8, 9	No	No	No	No	No	No

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. In addition, any variable whose name appears in the ENV_UNSET variable must be unset, for EAPIs listed in table 5.7 as supporting ENV_UNSET. ENV-UNSET

The package manager must ensure that the LC_CTYPE and LC_COLLATE locale categories are equivalent to the POSIX locale, as far as characters in the ASCII range (U+0000 to U+007F) are concerned. Only for EAPIs listed in such a manner in table 11.6. LOCALE-SETTINGS

11.1.1 USE and IUSE handling

This section discusses the handling of four variables:

IUSE is the variable calculated from the IUSE values defined in ebuilds and eclasses.

IUSE_REFERENCEABLE is a variable calculated from IUSE and a variety of other sources described below. It is purely a conceptual variable; it is inaccessible from the ebuild environment. Values in IUSE_REFERENCEABLE may legally be used in queries from other packages about an ebuild's state (for example, for use dependencies).

IUSE_EFFECTIVE is another conceptual, inaccessible variable. Values in IUSE_EFFECTIVE are those which an ebuild may legally use in queries about itself (for example, for the use function, and for use in dependency specification conditional blocks).

USE is a variable calculated by the package manager and exported to the ebuild environment.

In all cases, the values of IUSE_REFERENCEABLE and IUSE_EFFECTIVE are undefined during metadata generation.

Table 11.5: EAPIs supporting offset-prefix env variables

EAPI	EPREFIX?	EROOT?	ED?	ESYSROOT?
0, 1, 2	No	No	No	No
3, 4, 5, 6	Yes	Yes	Yes	No
7, 8, 9	Yes	Yes	Yes	Yes

Table 11.6: Locale settings for EAPIs

EAPI	Sane LC_CTYPE and LC_COLLATE?
0, 1, 2, 3, 4, 5	Undefined
6, 7, 8, 9	Yes

For EAPIs listed in table 5.6 as not supporting profile defined IUSE injection, IUSE_REFERENCEABLE is equal to the calculated IUSE value, and IUSE_EFFECTIVE contains the following values:

- All values in the calculated IUSE value.
- All possible values for the ARCH variable.
- All legal use flag names whose name starts with the lower-case equivalent of any value in the profile USE_EXPAND variable followed by an underscore.

For EAPIs listed in table 5.6 as supporting profile defined IUSE injection, IUSE_REFERENCEABLE and IUSE_EFFECTIVE are equal and contain the following values: PROFILE-IUSE-INJECT

- All values in the calculated IUSE value.
- All values in the profile IUSE_IMPLICIT variable.
- All values in the profile variable named USE_EXPAND_VALUES_${v}, where ${v} is any value in the intersection of the profile USE_EXPAND_UNPREFIXED and USE_EXPAND_IMPLICIT variables.
- All values for ${lower_v}_${x}, where ${x} is all values in the profile variable named USE_EXPAND_VALUES_${v}, where ${v} is any value in the intersection of the profile USE_EXPAND and USE_EXPAND_IMPLICIT variables and ${lower_v} is the lower-case equivalent of ${v}.

The USE variable is set by the package manager. For each value in IUSE_EFFECTIVE, USE shall contain that value if the flag is to be enabled for the ebuild in question, and shall not contain that value if it is to be disabled. In EAPIs listed in table 5.6 as not supporting profile defined IUSE injection, USE may contain other flag names that are not relevant for the ebuild.

For EAPIs listed in table 5.6 as supporting profile defined IUSE injection, the variables named in USE_EXPAND and USE_EXPAND_UNPREFIXED shall have their profile-provided values reduced to contain only those values that are present in IUSE_EFFECTIVE.

For EAPIs listed in table 5.6 as supporting profile defined IUSE injection, the package manager must save the calculated value of IUSE_EFFECTIVE when installing a package. Details are beyond the scope of this specification.

11.1.2 REPLACING_VERSIONS and REPLACED_BY_VERSION

In EAPIs listed in table 11.3 as supporting it, the REPLACING_VERSIONS variable shall be defined in pkg_preinst and pkg_postinst. In addition, it *may* be defined in pkg_pretend and pkg_setup, although ebuild authors should take care to handle binary package creation and installation correctly when using it in these phases. REPLACE-VERSION-VARS

REPLACING_VERSIONS is a list, not a single optional value, to handle pathological cases such as installing foo-2:2 to replace foo-2:1 and foo-3:2.

In EAPIs listed in table 11.3 as supporting it, the REPLACED_BY_VERSION variable shall be defined in pkg_prerm and pkg_postrm. It shall contain at most one value.

11.1.3 Offset-prefix variables

Table 11.7 lists the EAPIs which support offset-prefix installations. This support was initially added in EAPI 3, in the form of three extra variables. Two of these, EROOT and ED, are convenience variables using the variable EPREFIX. In EAPIs that do not support an offset-prefix, the installation offset is hardwired to /usr. In offset-prefix supporting EAPIs the installation offset is set as ${EPREFIX}/usr and hence can be adjusted using the variable EPREFIX. Note that the behaviour of OFFSET-PREFIX-VARS

Table 11.7: EAPIs supporting offset-prefix

EAPI	Supports offset-prefix?
0, 1, 2	No
3, 4, 5, 6, 7, 8, 9	Yes

Table 11.8: Variables that always or never end with a trailing slash

EAPI	Ends with a trailing slash?	
	`ROOT`, `EROOT`	`D`, `ED`
0, 1, 2, 3, 4, 5, 6	always	always
7, 8, 9	never	never

offset-prefix aware and agnostic is the same when `EPREFIX` is set to the empty string in offset-prefix aware EAPIs. The latter do have the variables `ED` and `EROOT` properly set, though.

11.1.4 Path variables and trailing slash

Unless specified otherwise, the paths provided through package manager variables do not end with a trailing slash. Consequently, the system root directory will be represented by the empty string. A few exceptions to this rule are listed in table 11.8 along with applicable EAPIs.

For EAPIs where those variables are defined to always end with a trailing slash, the package manager guarantees that a trailing slash will always be appended to the path in question. If the path specifies the system root directory, it will consist of a single slash (`/`).

For EAPIs where those variables are defined to never end with a trailing slash, the package manager guarantees that a trailing slash will never be present. If the path specifies the system root directory, it will be empty. TRAILING-SLASH

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.

Note: `pkg_pretend` is *not* part of the normal call sequence, and does not take part in environment saving.

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 section 7.1). 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 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.

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"
}
```

- 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

Available commands

This chapter 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.

12.1 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.

12.1.1 Guaranteed system commands

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

- All builtin commands in GNU bash, version as listed in table 6.1.
- `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, version as listed in table 12.1. GNU-PATCH
- `find` and `xargs` must be available, and must support all forms of invocations valid for GNU findutils version 4.4 or later. Only for EAPIs listed in table 12.1 as requiring GNU find. GNU-FIND

12.2 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. The availability of commands provided by the

Table 12.1: System commands for EAPIs

EAPI	GNU `find`?	GNU `patch` version
0, 1, 2, 3, 4	Undefined	Any
5, 6	Yes	Any
7, 8, 9	Yes	2.7

Table 12.2: EAPI command failure behaviour

EAPI	Command failure behaviour	Supports `nonfatal`?	`nonfatal` is both a function and an external command?
0, 1, 2, 3	Non-zero exit	No	n/a
4, 5, 6	Aborts	Yes	No
7, 8, 9	Aborts	Yes	Yes

particular types of dependencies is explained in section 8.1.

12.3 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. Unless otherwise specified, it is an error if an ebuild calls any of these commands in global scope.

Unless otherwise noted, any output of these commands ends with a newline.

12.3.1 Failure behaviour and related commands

Where a command is listed as having EAPI dependent failure behaviour, a failure shall either result in a non-zero exit status or abort the build process, as determined by table 12.2. DIE-ON-FAILURE

The following commands affect this behaviour:

nonfatal Takes one or more arguments and executes them as a command, preserving the exit status. If this results in a command being called that would normally abort the build process due to a failure, instead a non-zero exit status shall be returned. Only in EAPIs listed in table 12.2 as supporting `nonfatal`. NONFATAL

In EAPIs listed in table 12.2 as having `nonfatal` defined both as a shell function and as an external command, the package manager must provide both implementations to account for calling directly in ebuild scope or through `xargs`.

Explicit `die` or `assert` commands only respect `nonfatal` when called with the `-n` option and in EAPIs supporting this option, see table 12.6.

12.3.2 Banned commands

Some commands are banned in some EAPIs. If a banned command is called, the package manager must abort the build process indicating an error. BANNED-COMMANDS

12.3.3 Sandbox commands

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

addread Add a path to the permitted read list.

addwrite Add a path to the permitted write list.

addpredict Add a path 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 path to the deny list.

Table 12.3: Banned commands

EAPI	Command banned?					
	`dohard`	`dosed`	`einstall`	`dohtml`	`dolib`	`libopts`
0, 1, 2, 3	No	No	No	No	No	No
4, 5	Yes	Yes	No	No	No	No
6	Yes	Yes	Yes	No	No	No
7, 8, 9	Yes	Yes	Yes	Yes	Yes	Yes

EAPI	Command banned?				
	`useq`	`hasv`	`hasq`	`assert`	`domo`
0, 1, 2, 3, 4, 5, 6, 7	No	No	No	No	No
8	Yes	Yes	Yes	No	No
9	Yes	Yes	Yes	Yes	Yes

Table 12.4: Package manager query command options supported by EAPIs

EAPI	`--host-root`?	`-b`?	`-d`?	`-r`?
0, 1, 2, 3, 4	No	No	No	No
5, 6	Yes	No	No	No
7, 8, 9	No	Yes	Yes	Yes

12.3.4 Package manager query commands

These commands are used to extract information about the 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.

In EAPIs listed in table 12.4 as supporting option `--host-root`, this flag as the first argument will cause the query to apply to the host root. Otherwise, it applies to `ROOT`. PM-QUERY-OPTIONS

In EAPIs listed in table 12.4 as supporting options `-b`, `-d` and `-r`, these mutually exclusive flags as the first argument will cause the query to apply to locations targetted by `BDEPEND`, `DEPEND` and `RDEPEND`, respectively. When none of these options are given, `-r` is assumed.

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

best_version Takes exactly one package dependency specification as an argument. If a matching package is installed, prints `category/package-version` of the highest matching version; otherwise, prints an empty string. The exit code is unspecified.

12.3.5 Output commands

These commands display messages to the user. Unless otherwise stated, the entire argument list is used as a message, with backslash-escaped characters interpreted as for the `echo -e` command of bash, notably `\t` for a horizontal tab, `\n` for a new line, and `\\` for a literal backslash. These commands must be implemented internally as shell functions and may be called in global scope. Ebuilds must not run any of these commands once the current phase function has returned.

Unless otherwise noted, output may be sent to stderr or some other appropriate facility. In EAPIs listed in table 12.5 as not allowing stdout output, using stdout as an output facility is forbidden. OUTPUT-NO-STDOUT

einfo Displays an informational message.

einfon Displays an informational message without a trailing newline.

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

Table 12.5: Output commands for EAPIs

EAPI	Commands can output to stdout?	Supports `eqawarn`?
0, 1, 2, 3, 4, 5, 6	Yes	No
7, 8, 9	No	Yes

Table 12.6: Properties of `die` command in EAPIs

EAPI	Supports `die -n`?	`die` works in subshell?
0, 1, 2, 3, 4, 5	No	No
6	Yes	No
7, 8, 9	Yes	Yes

Table 12.7: EAPIs supporting `pipestatus`

EAPI	Supports `pipestatus`?
0, 1, 2, 3, 4, 5, 6, 7, 8	No
9	Yes

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

eqawarn Display a QA warning message intended for ebuild developers. The package manager may provide appropriate mechanisms to skip those messages for normal users. Must not go to stdout. Only available in EAPIs listed in table 12.5 as supporting `eqawarn`. EQAWARN

eerror 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 `eend`.

eend 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, prints a success indicator; otherwise, prints the message followed by a failure indicator. Returns its first argument as exit status.

12.3.6 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 If called under the `nonfatal` command (as per section 12.3.1) and with `-n` as its first parameter, displays a failure message provided in its following argument and then returns a non-zero exit status. Only in EAPIs listed in table 12.6 as supporting option `-n`. Otherwise, displays a failure message provided in its first and only argument, and then aborts the build process. NONFATAL-DIE

In EAPIs listed in table 12.6 as not providing subshell support, `die` is *not* guaranteed to work correctly if called from a subshell environment. SUBSHELL-DIE

assert Checks the shell's pipe status array, and if any element is non-zero (indicating failure), calls `die`, passing any parameters to it. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

pipestatus Checks the shell's pipe status array, i. e. the exit status of the command(s) in the most recently executed foreground pipeline. Returns shell true (0) if all elements are zero, or the last non-zero element otherwise. If called with `-v` as the first argument, also outputs the pipe status array as a space-separated list. Only available in EAPIs listed in table 12.7 as supporting `pipestatus`. PIPESTATUS

12.3.7 Patch commands

These commands are used during the `src_prepare` phase to apply patches to the package's sources. Ebuilds must not run any of these commands once the current phase function has returned.

eapply Takes zero or more GNU patch options, followed by one or more file or directory paths. Processes options and applies all patches found in specified locations according to algorithm 12.1. If applying the patches fails, it aborts the build using `die`, unless run using `nonfatal`, in which case it returns non-zero exit status. Only available in EAPIs listed in table 12.8 as supporting `eapply`. EAPPLY

Algorithm 12.1 `eapply` logic

```
1: if any parameter is equal to "--" then
2:    collect all parameters before the first "--" in the options array
3:    collect all parameters after the first "--" in the files array
4: else if any parameter that begins with a hyphen follows one that does not then
5:    abort the build process with an error
6: else
7:    collect all parameters beginning with a hyphen in the options array
8:    collect all remaining parameters in the files array
9: end if
10: if the files array is empty then
11:    abort the build process with an error
12: end if
13: for all x in the files array do
14:    if $x is a directory then
15:       if not any files match $x/*.diff or $x/*.patch then
16:          abort the build process with an error
17:       end if
18:       for all files f matching $x/*.diff or $x/*.patch, sorted in POSIX locale do
19:          call patch -p1 -f -g0 --no-backup-if-mismatch "${options[@]}" < "$f"
20:          if child process returned with non-zero exit status then
21:             return immediately with that status
22:          end if
23:       end for
24:    else
25:       call patch -p1 -f -g0 --no-backup-if-mismatch "${options[@]}" < "$x"
26:       if child process returned with non-zero exit status then
27:          return immediately with that status
28:       end if
29:    end if
30: end for
31: return shell true (0)
```

eapply_user Takes no arguments. Package managers supporting it apply user-provided patches to the source tree in the current working directory. Exact behaviour is implementation defined and beyond the scope of this specification. Package managers not supporting it must implement the command as a no-op. Returns shell true (0) if patches applied successfully, or if no patches were provided. Otherwise, aborts the build process, unless run using `nonfatal`, in which case it returns non-zero exit status. Only available in EAPIs listed in table 12.8 as supporting `eapply_user`. In EAPIs where it is supported, `eapply_user` must be called once in the `src_prepare` phase. For any subsequent calls, the command will do nothing and return 0. EAPPLY-USER

12.3.8 Build commands

These commands are used during the `src_configure`, `src_compile`, `src_test`, 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.

Table 12.8: Patch commands for EAPIs

EAPI	eapply?	eapply_user?
0, 1, 2, 3, 4, 5	No	No
6, 7, 8, 9	Yes	Yes

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`, after the default options below. `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: ECONF-OPTIONS

- `--prefix` must default to `${EPREFIX}/usr` unless overridden by `econf`'s caller.
- `--mandir` must be `${EPREFIX}/usr/share/man`
- `--infodir` must be `${EPREFIX}/usr/share/info`
- `--datadir` must be `${EPREFIX}/usr/share`
- `--datarootdir` must be `${EPREFIX}/usr/share`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--datarootdir` occurs in the output of `configure --help`.
- `--sysconfdir` must be `${EPREFIX}/etc`
- `--localstatedir` must be `${EPREFIX}/var/lib`
- `--docdir` must be `${EPREFIX}/usr/share/doc/${PF}`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--docdir` occurs in the output of `configure --help`.
- `--htmldir` must be `${EPREFIX}/usr/share/doc/${PF}/html`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--htmldir` occurs in the output of `configure --help`.
- `--with-sysroot` must be `${ESYSROOT:-/}`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--with-sysroot` occurs in the output of `configure --help`.
- `--build` must be the value of the `CBUILD` environment variable. This option will only be passed if `CBUILD` is non-empty.
- `--host` must be the value of the `CHOST` environment variable.
- `--target` must be the value of the `CTARGET` environment variable. This option will only be passed if `CTARGET` is non-empty.
- `--libdir` must be set according to algorithm 12.2.
- `--disable-dependency-tracking`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--disable-dependency-tracking` occurs in the output of `configure --help`.
- `--disable-silent-rules`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if the string `--disable-silent-rules` occurs in the output of `configure --help`.
- `--disable-static`, if the EAPI is listed in table 12.9 as using it. This option will only be passed if both strings `--enable-static` and `--enable-shared` occur in the output of `configure --help`.

Table 12.9: Extra `econf` arguments for EAPIs

EAPI	--datarootdir	--docdir	--htmldir	--with-sysroot
0, 1, 2, 3, 4, 5	No	No	No	No
6	No	Yes	Yes	No
7	No	Yes	Yes	Yes
8, 9	Yes	Yes	Yes	Yes

EAPI	--disable-dependency-tracking	--disable-silent-rules	--disable-static
0, 1, 2, 3	No	No	No
4	Yes	No	No
5, 6, 7	Yes	Yes	No
8, 9	Yes	Yes	Yes

For the option names beginning with `with-`, `disable-` or `enable-`, a string in `configure --help` output matches only if it is not immediately followed by any of the characters `[A-Za-z0-9+_.-]`.

Note that the `${EPREFIX}` component represents the same offset-prefix as described in table 11.2. It facilitates offset-prefix installations which is supported by EAPIs listed in table 11.5. When no offset-prefix installation is in effect, `EPREFIX` becomes the empty string, making the behaviour of `econf` equal for both offset-prefix supporting and agnostic EAPIs.

`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`, unless run using `nonfatal`, in which case it must return non-zero exit status.

Algorithm 12.2 `econf --libdir` logic

```
let prefix=${EPREFIX}/usr
if the caller specified --exec-prefix=$ep then
    let prefix=$ep
else if the caller specified --prefix=$p then
    let prefix=$p
end if
let libdir=
if the ABI environment variable is set then
    let libvar=LIBDIR_$ABI
    if the environment variable named by libvar is set then
        let libdir=the value of the variable named by libvar
    end if
end if
if libdir is non-empty then
    pass --libdir=$prefix/$libdir to configure
end if
```

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 12.1.1. `emake` must be an external program and cannot be a function or alias—it must be callable from e. g. `xargs`. Failure behaviour is EAPI dependent as per section 12.3.1.

einstall A shortcut for the command given in listing 12.1. Any arguments given to `einstall` are passed verbatim to `emake`, as shown. Failure behaviour is EAPI dependent as per section 12.3.1. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

The variable ED is defined as in table 11.2 and depends on the use of an offset-prefix. When such offset-prefix is absent, ED is equivalent to D. ED is always available in EAPIs that support offset-prefix installations as listed in table 11.5, hence EAPIs lacking offset-prefix support should use D instead of ED in the command given in listing 12.1. Variable libdir is an auxiliary local variable whose value is determined by algorithm 12.4.

Listing 12.1 einstall command

```
emake \
    prefix="${ED}"/usr \
    datadir="${ED}"/usr/share \
    mandir="${ED}"/usr/share/man \
    infodir="${ED}"/usr/share/info \
    libdir="${ED}"/usr/${libdir} \
    localstatedir="${ED}"/var/lib \
    sysconfdir="${ED}"/etc \
    -j1 \
    "$@" \
    install
```

12.3.9 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 including the offset-prefix ED in offset-prefix aware EAPIs, or just the staging directory D in offset-prefix agnostic EAPIs. Existing destination files are overwritten. These commands must all be external programs and not bash functions or aliases—that is, they must be callable from xargs. Calling any of these commands without a filename parameter is an error. 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 transfers file ownership to the superuser or its equivalent on the system or installation at hand. In a non-offset-prefix installation this ownership is 0:0, while in an offset-prefix aware installation this may be e. g. joe:users. Failure behaviour is EAPI dependent as per section 12.3.1.

doconfd Installs the given config files into /etc/conf.d/, by default with file mode 0644. For EAPIs listed in table 12.17 as respecting insopts in doconfd, the install options set by the most recent insopts call override the default. Failure behaviour is EAPI dependent as per section 12.3.1.

dodir Creates the given directories, by default with file mode 0755, or with the install options set by the most recent diropts call. Failure behaviour is EAPI dependent as per section 12.3.1.

dodoc Installs the given files into a subdirectory under /usr/share/doc/${PF}/ with file mode 0644. The subdirectory is set by the most recent call to docinto. If docinto has not yet been called, instead installs to the directory /usr/share/doc/${PF}/. For EAPIs listed in table 12.10 as supporting -r, if the first argument is -r, any subsequent arguments that are directories are installed recursively to the appropriate location; in any other case, it is an error for a directory to be specified. Any directories that don't already exist are created using install -d with no additional options. Failure behaviour is EAPI dependent as per section 12.3.1. DODOC

doenvd Installs the given environment files into /etc/env.d/, by default with file mode 0644. For EAPIs listed in table 12.17 as respecting insopts in doenvd, the install options set by the most recent insopts call override the default. Failure behaviour is EAPI dependent as per section 12.3.1.

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. Files are installed by default with file mode `0755`, or with the `install` options set by the most recent `exeopts` call. Failure behaviour is EAPI dependent as per section 12.3.1.

dohard Takes two parameters. Creates a hardlink from the second to the first. Both paths are relative to the staging directory including the offset-prefix `ED` in offset-prefix aware EAPIs, or just the staging directory `D` in offset-prefix agnostic EAPIs. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

doheader Installs the given header files into `/usr/include/`, by default with file mode `0644`. For EAPIs listed in table 12.17 as respecting `insopts` in `doheader`, the `install` options set by the most recent `insopts` call override the default. If the first argument is `-r`, then operates recursively, descending into any directories given. Only available in EAPIs listed in table 12.11 as supporting `doheader`. Failure behaviour is EAPI dependent as per section 12.3.1. DOHEADER

dohtml Installs the given HTML files into a subdirectory under `/usr/share/doc/${PF}/`. The subdirectory is `html` by default, but this can be overridden with the `docinto` function. 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, not to be confused with the global offset-prefix.

In EAPIs listed in table 12.3, this command is banned as per section 12.3.2. Failure behaviour is EAPI dependent as per section 12.3.1.

It is undefined whether a failure shall occur if `-r` is not specified and a directory is encountered. Ebuilds must not rely upon any particular behaviour.

doinfo Installs the given GNU Info files into the `/usr/share/info` area with file mode `0644`. Failure behaviour is EAPI dependent as per section 12.3.1.

doinitd Installs the given initscript files into `/etc/init.d`, by default with file mode `0755`. For EAPIs listed in table 12.17 as respecting `insopts` in `doinitd`, the `install` options set by the most recent `exeopts` call override the default. Failure behaviour is EAPI dependent as per section 12.3.1.

doins Takes one or more files as arguments and installs them into `INSDESTTREE`, by default with file mode `0644`, or with the `install` options set by the most recent `insopts` call. If the first argument is `-r`, then operates recursively, descending into any directories given. Any directories are created as if `dodir` was called. For EAPIs listed in table 12.12, `doins` must install symlinks as symlinks; for other EAPIs, behaviour is undefined if any symlink is encountered. Failure behaviour is EAPI dependent as per section 12.3.1. DOINS

dolib.a For each argument, installs it into the appropriate library subdirectory under `DESTTREE`, as determined by algorithm 12.4. Files are installed with file mode `0644`. Any symlinks are installed into the same directory as relative links to their original target. Failure behaviour is EAPI dependent as per section 12.3.1.

dolib.so As for `dolib.a` except each file is installed with mode `0755`.

dolib As for `dolib.a` except that the default install mode can be overriden with the `install` options set by the most recent `libopts` call. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

doman Installs the given man pages into the appropriate subdirectory of `/usr/share/man` depending upon its apparent section suffix (e. g. `foo.1` goes to `/usr/share/man/man1/foo.1`) with file mode `0644`.

DOMAN-LANGS

In EAPIs listed in table 12.13 as supporting language detection by filename, 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. Failure behaviour is EAPI dependent as per section 12.3.1.

With option `-i18n=`*lang*, a man page shall be installed into an appropriate subdirectory of `/usr/share/man/`*lang* (e. g. `/usr/share/man/`*lang*`/man1/foo.pl.1` would be the destination for `foo.pl.1`). The *lang* subdirectory level is skipped if *lang* is the empty string. In EAPIs specified by table 12.13, the `-i18n` option takes precedence over the language code in the filename.

domo Installs the given `.mo` files with file mode `0644` into the appropriate subdirectory of the locale tree, generated by taking the basename of the file, removing the `.*` suffix, and appending `/LC_MESSAGES`. The name of the installed files is the package name with `.mo` appended. Failure behaviour is EAPI dependent as per section 12.3.1. The locale tree location is EAPI dependent as per table 12.15. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

DOMO-PATH

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.

DOSYM-RELATIVE

In EAPIs listed in table 12.16 as supporting creation of relative paths, when called with option `-r`, the first parameter (the link target) is converted from an absolute path to a path relative to the the second parameter (the link name). The algorithm must return a result identical to the one returned by the function in listing 12.2, with `realpath` and `dirname` from GNU coreutils version 8.32. Specifying option `-r` together with a relative path as first (target) parameter is an error.

Failure behaviour is EAPI dependent as per section 12.3.1.

Listing 12.2 Create a relative path for `dosym -r`

```
dosym_relative_path() {
    local link=$(realpath -m -s "/${2#/}")
    local linkdir=$(dirname "${link}")
    realpath -m -s --relative-to="${linkdir}" "$1"
}
```

fowners Acts as for `chown`, but takes paths relative to the image directory. Failure behaviour is EAPI dependent as per section 12.3.1.

fperms Acts as for `chmod`, but takes paths relative to the image directory. Failure behaviour is EAPI dependent as per section 12.3.1.

keepdir For each argument, 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. Failure behaviour is EAPI dependent as per section 12.3.1.

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. In EAPIs specified by table 12.14, standard input is read when the first parameter is `-` (a hyphen). In this case, it is an error if standard input is a terminal.

NEWFOO-STDIN

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

newdoc As above, for `dodoc`.

newenvd As above, for `doenvd`.

Table 12.10: EAPIs supporting `dodoc -r`

EAPI	Supports `dodoc -r`?
0, 1, 2, 3	No
4, 5, 6, 7, 8, 9	Yes

Table 12.11: EAPIs supporting `doheader` and `newheader`

EAPI	Supports `doheader` and `newheader`?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

Table 12.12: EAPIs supporting symlinks for `doins`

EAPI	`doins` supports symlinks?
0, 1, 2, 3	No
4, 5, 6, 7, 8, 9	Yes

Table 12.13: `doman` language support options for EAPIs

EAPI	Language detection by filename?	Option `-i18n` takes precedence?
0, 1	No	Not applicable
2, 3	Yes	No
4, 5, 6, 7, 8, 9	Yes	Yes

Table 12.14: EAPIs supporting stdin for `new*` commands

EAPI	`new*` can read from stdin?
0, 1, 2, 3, 4	No
5, 6, 7, 8, 9	Yes

newexe As above, for `doexe`.

newheader As above, for `doheader`.

newinitd As above, for `doinitd`.

newins As above, for `doins`.

newlib.a As above, for `dolib.a`.

newlib.so As above, for `dolib.so`.

newman As above, for `doman`.

newsbin As above, for `dosbin`.

12.3.10 Commands affecting install destinations

The following commands are used to set the various destination trees and options 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 Takes exactly one argument, and sets the value of `DESTTREE` for future invocations of the above utilities to it. Creates the directory under `${ED}` in offset-prefix aware EAPIs or under `${D}`

Table 12.15: `domo` destination path in EAPIs

EAPI	Destination path
0, 1, 2, 3, 4, 5, 6	`${DESTTREE}/share/locale`
7, 8, 9	`/usr/share/locale`

Table 12.16: EAPIs supporting `dosym -r`

EAPI	`dosym` supports creation of relative paths?
0, 1, 2, 3, 4, 5, 6, 7	No
8, 9	Yes

Table 12.17: Commands respecting `insopts` for EAPIs

EAPI	`doins`?	`doconfd`?	`doenvd`?	`doheader`?
0, 1, 2, 3, 4, 5, 6, 7	Yes	Yes	Yes	Yes
8, 9	Yes	No	No	No

Table 12.18: Commands respecting `exeopts` for EAPIs

EAPI	`doexe`?	`doinitd`?
0, 1, 2, 3, 4, 5, 6, 7	Yes	Yes
8, 9	Yes	No

in offset-prefix agnostic EAPIs, using `install -d` with no additional options, if it does not already exist. Failure behaviour is EAPI dependent as per section 12.3.1.

insinto As `into`, for `INSDESTTREE`.

exeinto As `into`, for install path of `doexe` and `newexe`.

docinto As `into`, for install subdirectory of `dodoc` et al.

insopts Takes one or more arguments, and sets the options passed by `doins` et al. to the `install` command to them. Behaviour upon encountering empty arguments is undefined. Depending on EAPI, affects only those commands that are specified by table 12.17 as respecting `insopts`. INSOPTS

diropts As `insopts`, for `dodir` et al.

exeopts As `insopts`, for `doexe` et al. Depending on EAPI, affects only those commands that are specified by table 12.18 as respecting `exeopts`. EXEOPTS

libopts As `insopts`, for `dolib` et al. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

12.3.11 Commands controlling manipulation of files in the staging area

These commands are used to control optional manipulations that the package manager may perform on files in the staging directory `ED`, like compressing files or stripping symbols from object files.

For each of the operations mentioned below, the package manager shall maintain an inclusion list and an exclusion list, in order to control which directories and files the operation may or may not be performed upon. The initial contents of the two lists is specified below for each of the commands, respectively.

Any of these operations shall be carried out after `src_install` has completed, and before the execution of any subsequent phase function. For each item in the inclusion list, pretend it has the value of the `ED` variable prepended, then:

Table 12.19: Commands controlling manipulation of files in the staging area in EAPIs

EAPI	Supports controllable compression and `docompress`?	Supports controllable stripping and `dostrip`?
0, 1, 2, 3	No	No
4, 5, 6	Yes	No
7, 8, 9	Yes	Yes

- If it is a directory, act as if every file or directory immediately under this directory were in the inclusion list.
- If the item is a file, the operation may be performed on it, unless it has been excluded as described below.
- If the item does not exist, it is ignored.

Whether an item is to be excluded is determined as follows: For each item in the exclusion list, pretend it has the value of the `ED` variable prepended, then:

- If it is a directory, act as if every file or directory immediately under this directory were in the exclusion list.
- If the item is a file, the operation shall not be performed on it.
- If the item does not exist, it is ignored.

The package manager shall take appropriate steps to ensure that any operations that it performs on files in the staging area behave sensibly even if an item is listed in the inclusion list multiple times or if an item is a symlink.

DOCOMPRESS

In EAPIs listed in table 12.19 as supporting controllable compression, the package manager may optionally compress a subset of the files under the `ED` directory. The package manager shall ensure that its compression mechanisms do not compress a file twice if it is already compressed using the same compressed file format. For compression, the initial values of the two lists are as follows:

- The inclusion list contains `/usr/share/doc`, `/usr/share/info` and `/usr/share/man`.
- The exclusion list contains `/usr/share/doc/${PF}/html`.

DOSTRIP

In EAPIs listed in table 12.19 as supporting controllable stripping of symbols, the package manager may strip a subset of the files under the `ED` directory. For stripping of symbols, the initial values of the two lists are as follows:

- If the `RESTRICT` variable described in section 7.3.6 enables a `strip` token, the inclusion list is empty; otherwise it contains `/` (the root path).
- The exclusion list is empty.

The following commands may be used in `src_install` to alter these lists. It is an error to call any of these functions from any other phase.

docompress If the first argument is `-x`, add each of its subsequent arguments to the exclusion list for compression. Otherwise, add each argument to the respective inclusion list. Only available in EAPIs listed in table 12.19 as supporting `docompress`.

dostrip If the first argument is `-x`, add each of its subsequent arguments to the exclusion list for stripping of symbols. Otherwise, add each argument to the respective inclusion list. Only available in EAPIs listed in table 12.19 as supporting `dostrip`.

12.3.12 USE list functions

These functions provide behaviour based upon set or unset use flags. Ebuilds must not run any of these commands once the current phase function has returned.

Unless otherwise noted, if any of these functions is called with a flag value that is not included in `IUSE_EFFECTIVE`, either behaviour is undefined or it is an error as decided by table 12.20.

Table 12.20: EAPI behaviour for use queries not in IUSE_EFFECTIVE

EAPI	Behaviour
0, 1, 2, 3	Undefined
4, 5, 6, 7, 8, 9	Error

Table 12.21: usev, use_with and use_enable arguments for EAPIs

EAPI	usev has optional second argument?	use_with and use_enable support empty third argument?
0, 1, 2, 3	No	No
4, 5, 6, 7	No	Yes
8, 9	Yes	Yes

Table 12.22: EAPIs supporting usex and in_iuse

EAPI	usex?	in_iuse?
0, 1, 2, 3, 4	No	No
5	Yes	No
6, 7, 8, 9	Yes	Yes

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. It is guaranteed that this command is quiet.

usev The same as use, but also prints the flag name if the condition is met. In EAPIs listed in table 12.21 as supporting an optional second argument for usev, prints the second argument instead, if it is specified and if the condition is met. USEV

useq Deprecated synonym for use. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

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}). For EAPIs listed in table 12.21 as not supporting it, an empty third argument is treated as if it weren't provided. 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}. The condition is inverted if the flag name is prefixed with !; this is valid only for the two- and three-argument forms. USE-WITH

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

usex Accepts at least one and at most five arguments. The first argument is a USE flag name, any subsequent arguments (${arg2} to ${arg5}) are string values. If not provided, ${arg2} and ${arg3} default to yes and no, respectively; ${arg4} and ${arg5} default to the empty string. If the USE flag is set, outputs ${arg2}${arg4}. Otherwise, outputs ${arg3}${arg5}. The condition is inverted if the flag name is prefixed with !. Only available in EAPIs listed in table 12.22 as supporting usex. USEX

in_iuse Returns shell true (0) if the first argument (a USE flag name) is included in IUSE_EFFECTIVE, false otherwise. Only available in EAPIs listed in table 12.22 as supporting in_iuse. IN-IUSE

12.3.13 Text list functions

These functions check a list of arguments for a particular value. They must be implemented internally as shell functions and may be called in global scope.

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

hasv The same as `has`, but also prints the first argument if found. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

hasq Deprecated synonym for `has`. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

12.3.14 Version manipulation and comparison commands

VER-COMMANDS

These commands provide utilities for working with version strings. They must all be implemented internally as shell functions, i. e. they are callable in global scope. Availability of these commands per EAPI is listed in table 12.23.

For the purpose of version manipulation commands, the specification provides a method for splitting an arbitrary version string (not necessarily conforming to section 3.2) into a series of version components and version separators.

A version component consists either purely of digits (`[0-9]+`) or purely of upper- and lowercase ASCII letters (`[A-Za-z]+`). A version separator is either a string of any other characters (`[^A-Za-z0-9]+`), or it occurs at the transition between a sequence of digits and a sequence of letters, or vice versa. In the latter case, the version separator is an empty string.

The version string is processed left-to-right, with the successive version components being assigned successive indices starting with 1. The separator following a version component is assigned the index of the preceding version component. If the first version component is preceded by a non-empty string of version separator characters, this separator is assigned the index 0.

The version components are presumed present if not empty. The version separators between version components are always presumed present, even if they are empty. The version separators preceding the first version component and following the last are only presumed present if they are not empty.

Whenever the commands support ranges, the range is specified as an unsigned integer, optionally followed by a hyphen (`-`), which in turn is optionally followed by another unsigned integer.

A single integer specifies a single component or separator index. An integer followed by a hyphen specifies all components or separators starting with the one at the specified index. Two integers separated by a hyphen specify a range starting at the index specified by the first and ending at the second, inclusively.

ver_cut Takes a range as the first argument, and optionally a version string as the second. Prints a substring of the version string starting at the version component specified as start of the range and ending at the version component specified as end of the range. If the version string is not specified, `${PV}` is used.

If the range spans outside the present version components, the missing components and separators are presumed empty. In particular, the range starting at zero includes the zeroth version separator if present, and the range spanning past the last version component includes the suffix following it if present. A range that does not intersect with any present version components yields an empty string.

ver_rs Takes one or more pairs of arguments, optionally followed by a version string. Every argument pair specifies a range and a replacement string. Prints a version string after performing the specified separator substitutions. If the version string is not specified, `${PV}` is used.

For every argument pair specified, each of the version separators present at indices specified by the range is replaced with the replacement string, in order. If the range spans outside the range of present version separators, it is silently truncated.

Table 12.23: EAPIs supporting version manipulation commands

EAPI	ver_cut?	ver_rs?	ver_test?	ver_replacing?
0, 1, 2, 3, 4, 5, 6	No	No	No	No
7, 8	Yes	Yes	Yes	No
9	Yes	Yes	Yes	Yes

ver_test Takes two or three arguments. In the 3-argument form, takes an LHS version string, followed by an operator, followed by an RHS version string. In the 2-argument form, the first version string is omitted and `${PVR}` is used as LHS version string. The operator can be `-eq` (equal to), `-ne` (not equal to), `-gt` (greater than), `-ge` (greater than or equal to), `-lt` (less than) or `-le` (less than or equal to). Returns shell true (0) if the specified relation between the LHS and RHS version strings is fulfilled.

Both version strings must conform to the version specification in section 3.2. Comparison is done using algorithm 3.1.

ver_replacing Takes an operator and a version string as arguments, which follow the same specification as in `ver_test`. Iterates over the elements of `REPLACING_VERSIONS`, using `ver_test` to compare each element against the version string. Returns shell true (0) if the specified relation holds for any element, shell false (1) otherwise. Note that if `REPLACING_VERSIONS` is empty, shell false is returned. VER-REPLACING

Only available in EAPIs listed in table 12.23 as supporting `ver_replacing`. The command is only meaningful in phases where `REPLACING_VERSIONS` is defined.

12.3.15 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 `ED` (offset-prefix aware EAPIs) or `D` (offset-prefix agnostic EAPIs) 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:${ED}::g` in offset-prefix aware EAPIs and `s:${D}::g` in offset-prefix agnostic EAPIs. In EAPIs listed in table 12.3, this command is banned as per section 12.3.2.

unpack Unpacks one or more source archives, in order, into the current directory. For compressed files, creates the target file in the current directory, with the compression suffix removed from its name. 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`.

Arguments to `unpack` are interpreted as follows:

- A filename without path (i. e. not containing any slash) is looked up in `DISTDIR`.
- An argument starting with the string `./` is a path relative to the working directory.
- Otherwise, for EAPIs listed in table 12.24 as supporting absolute and relative paths, the argument is interpreted as a literal path (absolute, or relative to the working directory); for EAPIs listed as *not* supporting such paths, `unpack` shall abort the build process. UNPACK-ABSOLUTE

Any unrecognised file format shall be skipped without raising an error. If unpacking a supported file format fails, `unpack` shall abort the build process.

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

- tar files (`*.tar`). Ebuilds must ensure that GNU tar is installed.

Table 12.24: `unpack` behaviour for EAPIs

EAPI	Supports absolute and relative paths?	Case-insensitive matching?
0, 1, 2, 3, 4, 5	No	No
6, 7, 8, 9	Yes	Yes

Table 12.25: `unpack` extensions for EAPIs

EAPI	`.xz`?	`.tar.xz`?	`.txz`?	`.7z`?	`.rar`?	`.lha`?
0, 1, 2	No	No	No	Yes	Yes	Yes
3, 4, 5	Yes	Yes	No	Yes	Yes	Yes
6, 7	Yes	Yes	Yes	Yes	Yes	Yes
8, 9	Yes	Yes	Yes	No	No	No

- gzip-compressed files (`*.gz, *.z, *.Z`). Ebuilds must ensure that GNU gzip is installed.
- gzip-compressed tar files (`*.tar.gz, *.tgz, *.tar.z, *.tar.Z`). Ebuilds must ensure that GNU gzip and GNU tar are installed.
- bzip2-compressed files (`*.bz2, *.bz`). Ebuilds must ensure that bzip2 is installed.
- bzip2-compressed tar files (`*.tar.bz2, *.tbz2, *.tar.bz, *.tbz`). Ebuilds must ensure that bzip2 and GNU tar are installed.
- zip files (`*.zip, *.ZIP, *.jar`). Ebuilds must ensure that Info-ZIP Unzip is installed.
- 7zip files (`*.7z, *.7Z`). Ebuilds must ensure that P7ZIP is installed. Only for EAPIs listed in table 12.25 as supporting `.7z`.
- rar files (`*.rar, *.RAR`). Ebuilds must ensure that RARLAB's unrar is installed. Only for EAPIs listed in table 12.25 as supporting `.rar`.
- LHA archives (`*.LHA, *.LHa, *.lha, *.lzh`). Ebuilds must ensure that the lha program is installed. Only for EAPIs listed in table 12.25 as supporting `.lha`.
- ar archives (`*.a`). Ebuilds must ensure that GNU binutils is installed.
- deb packages (`*.deb`). Ebuilds must ensure that the deb2targz program is installed on those platforms where the GNU binutils ar program is not available and the installed ar program is incompatible with GNU archives. Otherwise, ebuilds must ensure that GNU binutils is installed.
- lzma-compressed files (`*.lzma`). Ebuilds must ensure that XZ Utils is installed.
- lzma-compressed tar files (`*.tar.lzma`). Ebuilds must ensure that XZ Utils and GNU tar are installed.
- xz-compressed files (`*.xz`). Ebuilds must ensure that XZ Utils is installed. Only for EAPIs listed in table 12.25 as supporting `.xz`.
- xz-compressed tar files (`*.tar.xz, *.txz`). Ebuilds must ensure that XZ Utils and GNU tar are installed. Only for EAPIs listed in table 12.25 as supporting `.tar.xz` or `.txz`.

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.

`unpack` matches filename extensions in a case-insensitive manner, for EAPIs listed such in table 12.24. UNPACK-IGNORE-CASE

inherit See section 10.1.

default Calls the `default_` function for the current phase (see section 9.1.17). Must not be called if the `default_` function does not exist for the current phase in the current EAPI. Only available in EAPIs listed in table 12.26 as supporting `default`. DEFAULT-FUNC

einstalldocs Takes no arguments. Installs the files specified by the `DOCS` and `HTML_DOCS` variables or a default set of files, according to algorithm 12.3. If called using `nonfatal` and any of the called commands returns a non-zero exit status, returns immediately with the same exit status. Only available in EAPIs listed in table 12.26 as supporting `einstalldocs`. EINSTALLDOCS

Algorithm 12.3 `einstalldocs` logic

1: save the value of the install directory for `dodoc`
2: set the install directory for `dodoc` to `/usr/share/doc/${PF}`
3: **if** the DOCS variable is a non-empty array **then**
4: call `dodoc -r "${DOCS[@]}"`
5: **else if** the DOCS variable is a non-empty scalar **then**
6: call `dodoc -r ${DOCS}`
7: **else if** the DOCS variable is unset **then**
8: **for all** *d* matching the filename expansion of `README* ChangeLog AUTHORS NEWS TODO CHANGES THANKS BUGS FAQ CREDITS CHANGELOG` **do**
9: **if** file *d* exists and has a size greater than zero **then**
10: call `dodoc` with *d* as argument
11: **end if**
12: **end for**
13: **end if**
14: set the install directory for `dodoc` to `/usr/share/doc/${PF}/html`
15: **if** the HTML_DOCS variable is a non-empty array **then**
16: call `dodoc -r "${HTML_DOCS[@]}"`
17: **else if** the HTML_DOCS variable is a non-empty scalar **then**
18: call `dodoc -r ${HTML_DOCS}`
19: **end if**
20: restore the value of the install directory for `dodoc`
21: **return** shell true (0)

get_libdir Prints the libdir name obtained according to algorithm 12.4. Must be implemented internally as a shell function and may be called in global scope. Only available in EAPIs listed in table 12.26 as supporting `get_libdir`. GET-LIBDIR

Algorithm 12.4 Library directory logic

1: let libdir=`lib`
2: **if** the ABI environment variable is set **then**
3: let libvar=LIBDIR_$ABI
4: **if** the environment variable named by libvar is set **then**
5: let libdir=the value of the variable named by libvar
6: **end if**
7: **end if**
8: **return** the value of libdir

edo Takes one or more arguments. The entire argument list is output as an informational message to stderr; individual tokens may be reformatted to avoid ambiguity. The first argument is then executed as a command, with the remaining arguments passed to it. If the command fails, `edo` aborts the build process using `die`, unless it was called under `nonfatal`, in which case it returns a non-zero exit status. EDO

`edo` must be implemented internally as a shell function. Only available in EAPIs listed in table 12.26 as supporting `edo`.

Table 12.26: Misc commands for EAPIs

EAPI	`default?`	`einstalldocs?`	`get_libdir?`	`edo?`
0, 1	No	No	No	No
2, 3, 4, 5	Yes	No	No	No
6, 7, 8	Yes	Yes	Yes	No
9	Yes	Yes	Yes	Yes

12.3.16 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. These commands must be implemented internally as shell functions and may be called in global scope. 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 $*`.

12.3.17 Reserved commands and variables

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

- `__` (two underscores)
- `abort`
- `dyn`
- `prep`

The same applies to functions and variables that contain any of the following strings (ignoring case):

- `ebuild` (unless immediately preceded by another letter)
- `hook`
- `paludis`
- `portage`

Chapter 13

Merging and unmerging

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

13.1 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.

13.2 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.

13.2.1 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 superuser.
- Any directory whose group is the primary group of the user used to perform the build must have its group be that of the superuser.

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

13.2.2 Empty directories

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

13.3 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`.

Table 13.1: Preservation of file modification times (mtimes)

EAPI	mtimes preserved?
0, 1, 2	Undefined
3, 4, 5, 6, 7, 8, 9	Yes

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.

13.3.1 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 superuser.
- Any file whose group is the primary group of the user used to perform the build must have its group be that of the superuser.
- 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 may be discarded.

13.3.2 File modification times

MTIME-PRESERVE

In EAPIs listed in table 13.1, the package manager must preserve modification times of regular files. This includes files being compressed before merging. Exceptions to this are files newly created by the package manager and binary object files being stripped of symbols.

When preserving, the seconds part of every regular file's mtime must be preserved exactly. The sub-second part must either be set to zero, or set to the greatest value supported by the operating system and filesystem that is not greater than the sub-second part of the original time.

For any given destination filesystem, the package manager must ensure that for any two preserved files a, b in that filesystem the relation $\text{mtime}(a) \leq \text{mtime}(b)$ still holds, if it held under the original image directory.

In other EAPIs, the behaviour with respect to file modification times is undefined.

13.3.3 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 5.3) 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.

Table 13.2: Rewriting of absolute symlinks in EAPIs

EAPI	Rewrite symlinks?
0, 1, 2, 3, 4, 5, 6, 7, 8	Yes
9	No

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.

13.4 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.

13.4.1 Rewriting

SYMLINK-REWRITE

In EAPIs listed in table 13.2 as rewriting symlinks, any absolute symlink whose link target starts with `D` must be rewritten with the leading `D` removed. The package manager should issue a notice when encountering such a symlink. In all other EAPIs, symlinks must be merged with their targets unmodified.

13.5 Hard links

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

13.6 Other files

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

Chapter 14

Metadata cache

14.1 Directory contents

The `metadata/cache` or `metadata/md5-cache` directories, if either of them exists, contain directories whose names are the same as categories in the repository. Each subdirectory may optionally contain one file per package version in that category, named `<package>-<version>`, in one of the formats described below.

The metadata cache may be incomplete or non-existent, and may contain additional bogus entries.

14.2 Legacy cache file format

The legacy cache file format is used in the `metadata/cache` directory. 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. Use flags that this package requires (`REQUIRED_USE`). Blank in some EAPIs; see table 7.2.
13. Post dependencies (`PDEPEND`)
14. Build-time dependencies for `CBUILD` host (`BDEPEND`). Blank in some EAPIs; see table 8.4.
15. The ebuild API version to which this package conforms (`EAPI`)
16. Properties (`PROPERTIES`). In some EAPIs, may optionally be blank, regardless of ebuild metadata; see table 7.2.
17. Defined phases (`DEFINED_PHASES`). In some EAPIs, may optionally be blank, regardless of ebuild metadata; see table 7.5.
18. Install-time dependencies (`IDEPEND`). Blank in some EAPIs; see table 8.4.
19. 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.

14.3 md5-dict cache file format

The "md5-dict" cache file format is used in the `metadata/md5-cache` directory. Each cache file contains `<key>=<value>` pairs, one per line, in arbitrary order. The keys are the same as those listed in section 14.2 except the `INHERITED` key. In addition, keys `_md5_` and `_eclasses_` contain values as defined below.

_md5_ The MD5 checksum of the ebuild for the package version.

eclasses A list of *name-checksum* pairs for all eclasses directly or indirectly inherited by the ebuild, in arbitrary order, where *name* is the eclass name and *checksum* is the MD5 checksum of the eclass. Pairs are separated from each other by single tab characters, as are *name* and *checksum* in each pair.

All MD5 checksums are computed and formatted as described in RFC 1321 [2].

Other keys may be present; their meanings are not defined here. Lines with an empty value can be omitted.

Glossary

This glossary 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.

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 *non-stand-alone repository* below.

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.

Bibliography

[1] Michał Górny, Robin Hugh Johnson, and Ulrich Müller. Full-tree verification using Manifest files. GLEP 74, Gentoo Linux, October 2022. URL: `https://www.gentoo.org/glep/glep-0074.html`.

[2] Ronald L. Rivest. The MD5 message-digest algorithm. RFC 1321, RFC Editor, April 1992. URL: `https://www.rfc-editor.org/rfc/rfc1321`.

[3] Michał Górny. Package and category metadata. GLEP 68, Gentoo Linux, October 2022. URL: `https://www.gentoo.org/glep/glep-0068.html`.

[4] Jason Stubbs. Virtuals deprecation. GLEP 37, Gentoo Linux, September 2006. URL: `https://www.gentoo.org/glep/glep-0037.html`.

[5] Piotr Jaroszyński. Use EAPI-suffixed ebuilds. GLEP 55, Gentoo Linux, May 2009. URL: `https://www.gentoo.org/glep/glep-0055.html`.

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 GLEP 68 [3].

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 `EMERGE_` variable and any `PORTAGE_` variable.
- 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

C.1 Long-obsolete features

The items described in this section are included for information only. Unless otherwise noted, 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.

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.

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.

use.defaults The `use.defaults` file in the profile directory was used to implement 'autouse'—switching USE flags on or off depending upon which packages are installed. It was deprecated long ago and finally removed in 2009.

C.2 Retroactive changes

In some exceptional cases, changes to the specification have been approved by the Gentoo Council without introducing a new EAPI. This section lists such retroactive changes.

Bash version EAPIs 0, 1 and 2 originally specified GNU Bash version 3.0. This was retroactively updated to version 3.2 (see table 6.1) in November 2009.

Old-style virtuals Historically, virtuals were special packages rather than regular ebuilds. An ebuild could specify in the `PROVIDE` metadata that it supplied certain virtuals, and the package manager had to bear this in mind when handling dependencies.

Old-style virtuals were supported by EAPIs 0, 1, 2, 3 and 4. They were phased out via GLEP 37 [4] and finally removed in 2011.

Listing C.1 If-else use blocks

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

Note: A 'new-style virtual' is a normal package that installs no files and uses its dependency requirements to pull in a 'provider'. This does not require any special handling from the package manager.

EAPI parsing The method to specify the EAPI of an ebuild used to be a shell variable assignment, and the package manager had to source the ebuild in order to determine the EAPI. Therefore any ebuild using a future EAPI would still have to be sourceable by old package managers, which imposed restrictions e. g. on updating the Bash version or on possible changes of global scope functions. Several approaches to overcome this limitation were discussed, notably GLEP 55 [5], which was ultimately rejected.

The current syntax of the `EAPI` assignment statement (see section 7.3.1), allowing the package manager to obtain the EAPI from the ebuild by a regular expression match and without sourcing it, was introduced in May 2012.

Package names Previously, package names were only required not to end in a hyphen followed by one or more digits. In October 2012 this was tightened to the specification in section 3.1.2, namely that they must not end in a hyphen followed by anything resembling a package version.

Asterisk in dependency specification In the = dependency operator specified in section 8.3.1, an asterisk used to induce string prefix comparison instead of the normal version comparison logic. That could lead to surprising results, e. g. `=dev-lang/perl-5.2*` matching `dev-lang/perl-5.22.0`. Moreover, implementation in package managers deviated from what was specified.

String prefix matching was effective in EAPIs 0, 1, 2, 3, 4 and 5. It was retroactively dropped in favour of the current behaviour in October 2015.

Empty dependency groups The dependency specification format (see section 8.2) originally permitted all-of, any-of, exactly-one-of, at-most-one-of and use-conditional groups with zero subelements. However, such empty groups were neither supported by all package managers nor used in ebuilds. They were dropped from the specification in October 2017.

econf --disable-static option The `--disable-static` option in `econf` (see section 12.3.8) was intended to disable only static Libtool archive building. The original check for either `--disable-static` or `--enable-static` occuring in `configure --help` output produced false positives.

The test mentioned above was effective in EAPI 8. It was updated in November 2022 to require both `--enable-static` and `--enable-shared`, and in addition checks for a proper end of these option strings.

econf matches configure --help output better The simple string matching used for `configure --help` output caused false positives for options like `--with-sysroot`. It was effective in EAPIs 4, 5, 6, 7 and 8, and was updated in April 2023 to check for a proper end of string for all option names beginning with `with-`, `disable-` or `enable-`.

Appendix D

Feature availability by EAPI

Note: This chapter is informative and for convenience only. Refer to the main text for specifics. For lack of space, EAPIs 0 to 5 have been omitted from the table below, as well as items that would have identical entries for all listed EAPIs.

Table D.1: Features in EAPIs

Feature	Ref.	EAPIs			
		6	7	8	9
`package.mask` directory	p. 19	No	Yes	Yes	Yes
Less strict `updates` syntax	p. 21	No	No	Yes	Yes
Default EAPI for profiles	p. 23	0	0	0	Top-level
Profile files as directories	p. 24	No	Yes	Yes	Yes
`package.provided`	p. 25	Optional	No	No	No
`use.stable`	p. 25	No	No	No	Yes
`package.use.stable`	p. 25	No	No	No	Yes
Bash version	p. 30	4.2	4.2	5.0	5.2
Selective URI restrictions	p. 33	No	No	Yes	Yes
`BDEPEND`	p. 36	No	Yes	Yes	Yes
`IDEPEND`	p. 36	No	No	Yes	Yes
Empty `||`, `^^` groups match	p. 38	Yes	No	No	No
Working dir in `pkg_*` phases	p. 42	Any	Any	Empty	Empty
`src_prepare` style	p. 44	6	6	8	8
Accumulate `PROPERTIES`	p. 50	No	No	Yes	Yes
Accumulate `RESTRICT`	p. 50	No	No	Yes	Yes
Export variables	p. 52	Yes	Yes	Yes	No
`PORTDIR`	p. 54	Yes	No	No	No
`ECLASSDIR`	p. 54	Yes	No	No	No
`SYSROOT`, `ESYSROOT`	p. 54	No	Yes	Yes	Yes
`BROOT`	p. 54	No	Yes	Yes	Yes
`DESTTREE`	p. 55	Yes	No	No	No
`INSDESTTREE`	p. 55	Yes	No	No	No
`ENV_UNSET`	p. 57	No	Yes	Yes	Yes
Trailing slash in `D` etc.	p. 59	Yes	No	No	No
GNU `patch` version	p. 61	Any	2.7	2.7	2.7
`nonfatal` function / external	p. 62	No	Yes	Yes	Yes
`dohtml`	p. 62	Yes	Banned	Banned	Banned
`dolib`	p. 62	Yes	Banned	Banned	Banned
`libopts`	p. 62	Yes	Banned	Banned	Banned
`useq`	p. 62	Yes	Yes	Banned	Banned
`hasv`	p. 62	Yes	Yes	Banned	Banned
`hasq`	p. 62	Yes	Yes	Banned	Banned

Feature	Ref.	EAPIs			
		6	7	8	9
`assert`	p. 62	Yes	Yes	Yes	Banned
`domo`	p. 62	Yes	Yes	Yes	Banned
Query command options	p. 63	`--host-root`	`-b`, `-d`, `-r`	`-b`, `-d`, `-r`	`-b`, `-d`, `-r`
Output commands use stdout	p. 63	Yes	No	No	No
`eqawarn`	p. 64	No	Yes	Yes	Yes
`die` in subshell	p. 64	No	Yes	Yes	Yes
`pipestatus`	p. 64	No	No	No	Yes
`econf --datarootdir`	p. 66	No	No	Yes	Yes
`econf --with-sysroot`	p. 66	No	Yes	Yes	Yes
`econf --disable-static`	p. 66	No	No	Yes	Yes
`domo` destination path	p. 70	`${DESTTREE}`	`/usr`	`/usr`	`/usr`
`dosym -r`	p. 70	No	No	Yes	Yes
`insopts` affects misc. cmds	p. 72	Yes	Yes	No	No
`exeopts` affects `doinitd`	p. 72	Yes	Yes	No	No
Controllable stripping	p. 73	No	Yes	Yes	Yes
`dostrip`	p. 73	No	Yes	Yes	Yes
`usev` second arg	p. 74	No	No	Yes	Yes
`ver_*` commands	p. 75	No	Yes	Yes	Yes
`ver_replacing`	p. 76	No	No	No	Yes
`unpack` support for `7z`	p. 76	Yes	Yes	No	No
`unpack` support for `lha`	p. 76	Yes	Yes	No	No
`unpack` support for `rar`	p. 76	Yes	Yes	No	No
`edo`	p. 78	No	No	No	Yes
Absolute symlink rewriting	p. 82	Yes	Yes	Yes	No

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, IUSE-DEFAULTS on page 32.
- Slot dependencies, SLOT-DEPS on page 40.
- Different `src_compile` implementation, SRC-COMPILE on page 45.

EAPI 2

EAPI 2 is EAPI 1 with the following changes:

- `SRC_URI` arrows, SRC-URI-ARROWS on page 33.
- Use dependencies, USE-DEPS on page 39.
- `!` and `!!` blockers, BANG-STRENGTH on page 40.
- `src_prepare`, SRC-PREPARE on page 44.
- `src_configure`, SRC-CONFIGURE on page 44.
- Different `src_compile` implementation, SRC-COMPILE on page 45.
- `default_` phase functions for phases `pkg_nofetch`, `src_unpack`, `src_prepare`, `src_configure`, `src_compile` and `src_test`; DEFAULT-PHASE-FUNCS on page 48.
- `doman` language detection by filename, DOMAN-LANGS on page 70.
- `default` function, DEFAULT-FUNC on page 78.

EAPI 3

EAPI 3 is EAPI 2 with the following changes:

- Offset-prefix support by definition of `EPREFIX`, `ED` and `EROOT`, OFFSET-PREFIX-VARS on page 58.
- `unpack` supports `.xz` and `.tar.xz`, UNPACK-EXTENSIONS on page 76.
- File modification times are preserved, MTIME-PRESERVE on page 81.

EAPI 4

EAPI 4 is EAPI 3 with the following changes:

- `REQUIRED_USE`, REQUIRED-USE on page 32.
- `PROPERTIES` support is mandatory, PROPERTIES on page 32.
- `RDEPEND=DEPEND` no longer done, RDEPEND-DEPEND on page 34.
- `DEFINED_PHASES` support is mandatory, DEFINED-PHASES on page 35.
- Use dependency defaults, USE-DEP-DEFAULTS on page 41.
- `S` to `WORKDIR` fallback restricted, S-WORKDIR-FALLBACK on page 42.
- `pkg_pretend`, PKG-PRETEND on page 42.
- Default `src_install` no longer a no-op, SRC-INSTALL on page 46.
- `pkg_info` can run on non-installed packages, PKG-INFO on page 47.
- `AA` is gone, AA on page 53.
- `KV` is gone, KV on page 56.
- `MERGE_TYPE`, MERGE-TYPE on page 56.
- `REPLACING_VERSIONS` and `REPLACED_BY_VERSION`, REPLACE-VERSION-VARS on page 58.
- Utilities now die on failure, DIE-ON-FAILURE on page 62, unless called under `nonfatal`, NONFATAL on page 62
- `dohard`, `dosed` banned, BANNED-COMMANDS on page 62.
- `econf` adds `--disable-dependency-tracking`, ECONF-OPTIONS on page 66.
- `dodoc -r` support, DODOC on page 68.
- `doins` supports symlinks, DOINS on page 69.
- `doman -i18n` option takes precedence, DOMAN-LANGS on page 70.
- Controllable compression and `docompress`, DOCOMPRESS on page 73.
- `use_with` and `use_enable` support empty third argument, USE-WITH on page 74.

EAPI 5

EAPI 5 is EAPI 4 with the following changes:

- Stable use masking and forcing, STABLEMASK on page 26.
- `REQUIRED_USE` now supports `??` groups, AT-MOST-ONE-OF on page 38.
- Slot operator dependencies, SLOT-OPERATOR-DEPS on page 40.
- `SLOT` now supports an optional sub-slot part, SUB-SLOT on page 40.
- `src_test` supports parallel tests, PARALLEL-TESTS on page 46.
- `EBUILD_PHASE_FUNC`, EBUILD-PHASE-FUNC on page 56.
- `USE` is calculated differently, PROFILE-IUSE-INJECT on page 58.
- `find` is guaranteed to be GNU, GNU-FIND on page 61.
- `best_version` and `has_version` support the `--host-root` option, PM-QUERY-OPTIONS on page 63.
- `econf` adds `--disable-silent-rules`, ECONF-OPTIONS on page 66.
- `doheader` and `newheader` support, DOHEADER on page 69.
- `new*` can read from standard input, NEWFOO-STDIN on page 70.
- `usex` support, USEX on page 74.

EAPI 6

EAPI 6 is EAPI 5 with the following changes:

- Bash version is 4.2, BASH-VERSION on page 30.
- `failglob` is enabled in global scope, FAILGLOB on page 30.
- Default `src_prepare` no longer a no-op, SRC-PREPARE on page 44.
- Different `src_install` implementation, SRC-INSTALL on page 46.
- `LC_CTYPE` and `LC_COLLATE` compatible with POSIX locale, LOCALE-SETTINGS on page 57.
- `einstall` banned, BANNED-COMMANDS on page 62.
- `die` and `assert` called with `-n` respect `nonfatal`, NONFATAL-DIE on page 64.

- `eapply` support, EAPPLY on page 65.
- `eapply_user` support, EAPPLY-USER on page 65.
- `econf` adds `--docdir` and `--htmldir`, ECONF-OPTIONS on page 66.
- `in_iuse` support, IN-IUSE on page 74.
- `unpack` supports absolute and relative paths, UNPACK-ABSOLUTE on page 76.
- `unpack` supports `.txz`, UNPACK-EXTENSIONS on page 76.
- `unpack` matches filename extensions case-insensitively, UNPACK-IGNORE-CASE on page 77.
- `einstalldocs` support, EINSTALLDOCS on page 78.
- `get_libdir` support, GET-LIBDIR on page 78.

EAPI 7

EAPI 7 is EAPI 6 with the following changes:

- `profiles/package.mask` can be a directory, PACKAGE-MASK-DIR on page 19.
- `package.mask`, `package.use`, `use.*` and `package.use.*` in a profile can be directories, PROFILE-FILE-DIRS on page 24.
- `package.provided` in profiles banned, PACKAGE-PROVIDED on page 25.
- Empty `||` and `^^` dependency groups no longer count as being matched, EMPTY-DEP-GROUPS on page 38.
- `PORTDIR` is gone, PORTDIR on page 54.
- `ECLASSDIR` is gone, ECLASSDIR on page 54.
- `DESTTREE` is gone, DESTTREE on page 55.
- `INSDESTTREE` is gone, INSDESTTREE on page 55.
- `ROOT`, `EROOT`, `D`, `ED` no longer end with a trailing slash, TRAILING-SLASH on page 59.
- `SYSROOT` and `ESYSROOT`, SYSROOT on page 54.
- `econf` adds `--with-sysroot`, ECONF-OPTIONS on page 66.
- `BDEPEND`, BDEPEND on page 36.
- `BROOT`, BROOT on page 54.
- `best_version` and `has_version` support `-b`, `-d` and `-r` options instead of `--host-root`, PM-QUERY-OPTIONS on page 63.
- `ENV_UNSET`, ENV-UNSET on page 57.
- `patch` is compatible with GNU patch 2.7, GNU-PATCH on page 61.
- `nonfatal` is both a shell function and an external command, NONFATAL on page 62.
- `dohtml` banned, BANNED-COMMANDS on page 62.
- `dolib` and `libopts` banned, BANNED-COMMANDS on page 62.
- Output commands no longer use stdout, OUTPUT-NO-STDOUT on page 63.
- `eqawarn`, EQAWARN on page 64.
- `die` is guaranteed to work in a subshell, SUBSHELL-DIE on page 64.
- `domo` installs to `/usr`, DOMO-PATH on page 70.
- Controllable stripping and `dostrip`, DOSTRIP on page 73.
- Version manipulation and comparison commands, VER-COMMANDS on page 75.

EAPI 8

EAPI 8 is EAPI 7 with the following changes:

- Less strict naming rules for files in `updates` directory, UPDATES-FILENAMES on page 21.
- Bash version is 5.0, BASH-VERSION on page 30.
- Selective fetch/mirror restriction, URI-RESTRICT on page 33.
- `IDEPEND`, IDEPEND on page 36.
- Empty working directory in `pkg_*` phase functions, PHASE-FUNCTION-DIR on page 42.
- Different `src_prepare` implementation, SRC-PREPARE on page 44.
- `PROPERTIES` and `RESTRICT` accumulated across eclasses, ACCUMULATE-VARS on page 50.
- `useq` banned, BANNED-COMMANDS on page 62.
- `hasv` and `hasq` banned, BANNED-COMMANDS on page 62.

- `econf` adds `--datarootdir`, ECONF-OPTIONS on page 66.
- `econf` adds `--disable-static`, ECONF-OPTIONS on page 66.
- `dosym` can create relative paths, DOSYM-RELATIVE on page 70.
- `insopts` no longer affects `doconfd`, `doenvd` and `doheader`, INSOPTS on page 72.
- `exeopts` no longer affects `doinitd`, EXEOPTS on page 72.
- `usev` supports an optional second argument, USEV on page 74.
- `unpack` no longer supports `.7z`, `.rar`, `.lha`, UNPACK-EXTENSIONS on page 76.

EAPI 9

EAPI 9 is EAPI 8 with the following changes:

- Different default EAPI for profiles, PROFILE-EAPI-DEFAULT on page 23.
- `use.stable` and `package.use.stable`, USE-STABLE on page 25.
- Bash version is 5.2, BASH-VERSION on page 30.
- Variables no longer exported, EXPORT-VARS on page 52.
- `assert` banned, BANNED-COMMANDS on page 62.
- `domo` banned, BANNED-COMMANDS on page 62.
- `pipestatus`, PIPESTATUS on page 64.
- `ver_replacing`, VER-REPLACING on page 76.
- `edo`, EDO on page 78.
- Absolute symlinks no longer rewritten, SYMLINK-REWRITE on page 82.

usev This helper has an optional second argument now: `usev <flag> [true]`. If the flag is set, outputs *[true]*, or the flag's name if called with only one argument. Otherwise outputs nothing. See USEV on page 74.

Removals/bans

useq No longer allowed. Use regular `use` as a drop-in replacement. See BANNED-COMMANDS on page 62.

hasv and hasq No longer allowed. Regular `has` should be used instead. See BANNED-COMMANDS on page 62.

unpack No longer supports unpacking of 7-Zip, RAR, and LHA archives. See UNPACK-EXTENSIONS on page 76.

EAPI 9

Additions/changes

Default EAPI for profiles Profile directories without their own `eapi` file no longer default to EAPI 0, but to the EAPI specified in the top-level `profiles` directory. See PROFILE-EAPI-DEFAULT on page 23.

use.stable and package.use.stable Profile dirs may contain two new files. They can be used to override the USE flags specified by `make.defaults`, but act only on packages that would be merged due to a stable keyword. See USE-STABLE on page 25.

Bash version Ebuilds can use features of Bash version 5.2 (was 5.0 before). See BASH-VERSION on page 30.

Variables are no longer exported The package manager no longer exports its defined shell variables (exceptions are `TMPDIR` and `HOME`) to the environment. The same applies to variables defined in `make.defaults` that have specific meanings. See EXPORT-VARS on page 52.

pipestatus Checks if all commands in the last executed pipeline have returned an exit status of zero. When the `-v` option is specified, also prints the shell's pipe status array. See PIPESTATUS on page 64.

ver_replacing *op v2* Checks if the relation *v1 op v2* is true for any element *v1* of `REPLACING_VERSIONS`. *op* can be any operator that is accepted by `ver_test`. See VER-REPLACING on page 76.

edo Outputs its entire argument list as an informational message, then executes it as a simple shell command, with standard failure behaviour. See EDO on page 78.

Merging of symlinks When merging `D` to `ROOT`, absolute symlinks are now merged as-is. The package manager will no longer strip a leading `D` from their link targets. See SYMLINK-REWRITE on page 82.

Removals/bans

assert No longer allowed. Use `pipestatus` instead. See BANNED-COMMANDS on page 62.

domo No longer allowed. Use `insinto` and `newins` as replacement. See BANNED-COMMANDS on page 62.

EAPI Cheat Sheet

Christian Faulhammer
fauli@gentoo.org

Ulrich Müller
ulm@gentoo.org

Version 9.0
21st June 2025

Abstract

An overview of the main EAPI changes in Gentoo, for ebuild authors. For full details, consult the Package Manager Specification found on the project page;[1] this is an incomplete summary only.

Official Gentoo EAPIs are consecutively numbered integers (0, 1, 2, ...). Except where otherwise noted, an EAPI is the same as the previous EAPI. All labels refer to the Package Manager Specification itself, built from the same checkout as this document.

EAPIs 0 to 6

Omitted for lack of space. See previous versions of this document for differences between these EAPIs.

EAPI 7 (2018-04-30)

Additions/changes

package.* and use.* These profile files can be directories instead of regular files. This is intended to be used in overlays only. See PACKAGE-MASK-DIR on page 19 and PROFILE-FILE-DIRS on page 24.

[1] https://wiki.gentoo.org/wiki/Project:Package_Manager_Specification
[2] https://creativecommons.org/licenses/by-sa/4.0/

|| and ^^ dependency groups These groups now evaluate to false when they are empty (for example, if there are only unmatched use dependencies inside of them). See EMPTY-DEP-GROUPS on page 38.

No trailing slash The paths specified by `ROOT`, `EROOT`, `D`, and `ED` no longer end with a slash. Thus, default `ROOT` is empty now. See TRAILING-SLASH on page 59.

Cross compilation support Several variables have been added and some commands have been extended for better cross compilation support:

BDEPEND Build dependencies are divided into two classes: `BDEPEND` for native build tools (`CBUILD`); `DEPEND` for dependencies compatible with the system being built (`CHOST`). See BDEPEND on page 36.

SYSROOT The path to the root directory for `DEPEND` type dependencies. See SYSROOT on page 54.

ESYSROOT The concatenation of `SYSROOT` and the applicable offset-prefix. See SYSROOT on page 54.

BROOT The prefixed root directory path for `BDEPEND` type dependencies, typically executable build tools. See BROOT on page 54.

econf If supported, configure will be called with the `--with-sysroot=${ESYSROOT:-/}` option. See ECONF-OPTIONS on page 66.

has_version and best_version These helpers support `-b`, `-d` or `-r` options, causing the query to apply to `BDEPEND`, `DEPEND` or `RDEPEND` (the default). This replaces the `--host-root` option. See PM-QUERY-OPTIONS on page 63.

Environment blacklist Any environment variable listed in the profile-defined `ENV_UNSET` variable will be unset by the package manager. See ENV-UNSET on page 57.

patch All inputs valid for GNU patch version 2.7 are supported. Especially, this includes support for git-formatted patches. See GNU-PATCH on page 61.

nonfatal In addition to its definition as a shell function, the `nonfatal` wrapper has now a fallback implementation as an external command. Thus, it can be called from other commands. See NONFATAL on page 62.

Output commands `einfo` and friends no longer use stdout, so inside of command substitution their output won't be caught. See OUTPUT-NO-STDOUT on page 63.

eqawarn The `eqawarn` output command is supported in the package manager itself. See EQAWARN on page 64.

die in subshell The `die` command is guaranteed to work in a subshell context. See SUBSHELL-DIE on page 64.

domo destination `domo` installs the specified files under `/usr/share/locale` instead of `${DESTTREE}/share/locale`. See DOMO-PATH on page 70.

Controllable stripping The `dostrip -x` command can be used to add paths to an exclusion list for stripping of debug symbols, to allow more fine-grained control than with `RESTRICT="strip"`. See DOSTRIP on page 73.

Version manipulation and comparison commands

ver_cut *range* [*version*] Print the version substring specified by *range*. *version* defaults to `PV`.

ver_rs *range repl* ... [*version*] Replace all version separators in *range* by string *repl*. Multiple *range repl* pairs are allowed. *version* defaults to `PV`.

ver_test [*v1*] *op v2* Check if the relation *v1 op v2* is true. *v1* defaults to `PVR`; *op* can be `-eq`, `-ne`, `-gt`, `-ge`, `-lt` or `-le`.

See VER-COMMANDS on page 75.

Removals/bans

package.provided Deprecated since a long time and finally dropped. See PACKAGE-PROVIDED on page 25.

PORTDIR and ECLASSDIR No longer defined, because ebuilds should not directly access files in the repository. See PORTDIR on page 54 and ECLASSDIR on page 54.

DESTTREE and INSDESTTREE Not defined any more. Use the `into` and `insinto` commands instead. See DESTTREE on page 55 and INSDESTTREE on page 55.

dohtml No longer allowed. `doins -r` can be used as a replacement. See BANNED-COMMANDS on page 62.

dolib and libopts No longer allowed. The specific `dolib.a` or `dolib.so` commands should be used as replacement. See BANNED-COMMANDS on page 62.

EAPI 8 (2021-06-13)

Additions/changes

profiles/updates directory Arbitrary filenames are now allowed, instead of strict naming by quarters (like `2Q-2021`). See UPDATES-FILENAMES on page 21.

Bash version Ebuilds can use features of Bash version 5.0 (was 4.2 before). See BASH-VERSION on page 30.

Selective fetch/mirror restriction In `SRC_URI`, adding a `fetch+` or `mirror+` prefix to an individual URI means that the file may be fetched or mirrored. This overrides the corresponding global settings in the `RESTRICT` variable. See URI-RESTRICT on page 33.

IDEPEND This variable specifies install-time dependencies on packages used in (e. g.) `pkg_postinst`. In a cross-compilation environment, these are dependencies for native tools (`CBUILD`). See IDEPEND on page 36.

pkg_* phases The initial working directory is guaranteed to be empty. See PHASE-FUNCTION-DIR on page 42.

src_prepare Items in the `PATCHES` variable are interpreted as files, even if their name begins with a hyphen. See SRC-PREPARE on page 44.

PROPERTIES and RESTRICT These variables are accumulated across the ebuild and inherited eclasses, like `IUSE`, `REQUIRED_USE`, and `*DEPEND` were before. See ACCUMULATE-VARS on page 50.

econf If supported, options `--disable-static` and `--datarootdir=${EPREFIX}/usr/share` are passed to configure, respectively. See ECONF-OPTIONS on page 66.

dosym With the new option `-r`, an absolute path specified for the link target will be converted to a path relative to the link location. See DOSYM-RELATIVE on page 70.

insopts Commands `doconfd`, `doenvd`, `doheader` install files with fixed mode 0644, i. e. they are no longer affected by `insopts`. See INSOPTS on page 72.

exeopts Command `doinitd` installs files with fixed mode 0755, i. e. it is no longer affected by `exeopts`. See EXEOPTS on page 72.