Gentoo Development Guide

Dependencies

Automatic dependency resolution is one of the most useful features provided by `emerge`.

Build Dependencies

The `DEPEND` ebuild variable should specify any dependencies which are required to unpack, patch, compile or install the package (but see Implicit System Dependency for exemptions).

Runtime Dependencies

The `RDEPEND` ebuild variable should specify any dependencies which are required at runtime. This includes libraries (when dynamically linked), any data packages and (for interpreted languages) the relevant interpreter. In EAPI=3 or older, if this variable is not specified it defaults to the value of `DEPEND`, however the implicit usage is frowned upon. In EAPI=4, the implicit behaviour was removed and the assignment is always explicit.

Note that when installing from a binary package, only `RDEPEND` will be checked. It is therefore necessary to include items even if they are also listed in `DEPEND`.

Items which are in `RDEPEND` but not `DEPEND` could *in theory* be merged *after* the target package. Portage does not currently do this.

Post-Merge Dependencies

The `PDEPEND` variable specifies dependencies which must be merged *after* the package. This is sometimes used for plugins which have a dependency upon the package being merged. Generally `PDEPEND` should be avoided in favour of `RDEPEND` except where this will create circular dependency chains.

Implicit System Dependency

All packages have an implicit compile-time and runtime dependency upon the entire `system` target. It is therefore not necessary, nor advisable, to specify dependencies upon toolchain packages like `gcc`, `libc` and so on, except where specific versions or packages (for example, `glibc` over `uclibc`) are required. Note that this rule also needs consideration for packages like `flex`, `zlib` and `libtool`, which aren't in the `system` target for every profile. For example, the embedded profile doesn't have `zlib` in `system` target, the `libtool` ABI might change and break building order and `flex` might get removed from the `system` target in future.

However, packages which are included in the `system` target, or are dependencies of `system` target packages, should generally include a complete dependency list (excluding bootstrap packages). This makes `emerge -e system` possible when installing from a stage 1 or stage 2 tarball.

Basic Dependency Syntax

A basic `DEPEND` specification might look like the following:

```
DEPEND="dev-lang/ruby
    dev-ruby/ruby-gtk2
    dev-ruby/mysql-ruby"
```

Each atom is the full category and name of a package. Atoms are separated by arbitrary whitespace — convention is to specify one atom per line for readability purposes. When specifying names, the category part should be treated as mandatory.

Version Dependencies

Sometimes a particular version of a package is needed. Where this is known, it should be specified. A simple example:

```
DEPEND=">=dev-libs/openssl-0.9.7d"
```

This states that at least version 0.9.7d of `openssl` is required.

Version Specifiers

Available version specifiers are:

Specifier	Meaning
`>=app-misc/foo-1.23`	Version 1.23 or later is required.

`>app-misc/foo-1.23`	A version strictly later than 1.23 is required.
`~app-misc/foo-1.23`	Version 1.23 (or any `1.23-r*`) is required.
`=app-misc/foo-1.23`	Exactly version 1.23 is required. If at all possible, use the `~` form to simplify revision bumps.
`<=app-misc/foo-1.23`	Version 1.23 or older is required.
`<app-misc/foo-1.23`	A version strictly before 1.23 is required.

Ranged Dependencies

To specify "version 2.x (not 1.x or 3.x)" of a package, it is necessary to use the asterisk postfix. This is most commonly seen in situations like:

```
DEPEND="gtk? ( =x11-libs/gtk+-1.2* )"
```

Note that the equals sign is mandatory, and that there is no dot before the asterisk. Also note that when selecting all versions in a specific `SLOT`, `SLOT` dependencies should be used (see below).

Blockers

Sometimes two packages cannot be installed in parallel. This is handled by blockers. A blocker is specified as follows:

```
RDEPEND="!app-misc/foo"
```

Note that blockers are usually *runtime* rather than buildtime.

Specific versions can also be blocked:

```
RDEPEND="!<app-misc/foo-1.3"
```

Blockers can be optional based upon `USE` flags as per normal dependencies.

Blockers added to older ebuilds should not be expected to be retroactive. If the user already has the ebuild installed, any changes to the ebuild should not be expected to make any difference. This means that you should add the blockers to whichever ebuild is the newest (even if it means that logically it would seem backwards). For example, certain versions of portage don't like some versions of bash, but the blocker was put into bash because that was the newer package that caused the issues.

SLOT Dependencies

In order to depend on a package in a specific `SLOT` you must specify at least `EAPI="1"`.

To depend on a specific `SLOT`, `:SLOT` should be appended to the package name, where 'SLOT' is the `SLOT` of the package wanted:

```
DEPEND="qt3? ( x11-libs/qt:3 )
    gtk? ( x11-libs/gtk+:2 )
```

USE-Conditional Dependencies

To depend upon a certain package if and only if a given `USE` flag is set:

```
DEPEND="perl? ( dev-lang/perl )
    ruby? ( >=dev-lang/ruby-1.8 )
    python? ( dev-lang/python )"
```

It is also possible to depend upon a certain package if a given `USE` flag is *not* set:

```
RDEPEND="!crypt? ( net-misc/netkit-rsh )"
```

This should **not** be used for disabling a certain `USE` flag on a given architecture. In order to do this, the architecture team should add the `USE` flag to their `use.mask` file in the `profiles/arch` directory of the Portage tree.

This can be nested:

```
DEPEND="!build? (
    gcj? (
        gtk? (
            x11-libs/libXt
            x11-libs/libX11
            x11-libs/libXtst
            x11-proto/xproto
            x11-proto/xextproto
            >=x11-libs/gtk+-2.2
            x11-libs/pango
        )
        >=media-libs/libart_lgpl-2.1
    )
    >=sys-libs/ncurses-5.2-r2
    nls? ( sys-devel/gettext )
)"
```

Any of Many Dependencies

To depend on either `foo` or `bar`:

```
DEPEND="|| ( app-misc/foo app-misc/bar )"
```

To depend on either `foo` or `bar` if the `baz` `USE` flag is set:

```
DEPEND="baz? ( || ( app-misc/foo app-misc/bar ) )"
```

Any of Many Versus USE

Say `fnord` can be built against either `foo` or `bar`. Then a `USE` flag is not necessary if and only if all of the following hold:

- `fnord` is merged on a system which has `foo` and not `bar` installed. `foo` is then unmerged, and `bar` is installed. `fnord` must continue to work correctly.
- A binary package of `fnord` made on a system with `foo` and not `bar` can be taken and installed on a system with `bar` and not `foo`.

Built with USE Dependencies

In order to use built with use dependencies you must specify `EAPI="2"`.

Available specifiers are:

Specifier	Meaning
`app-misc/foo[bar]`	foo must have bar enabled.
`app-misc/foo[bar,baz]`	foo must have both bar and baz enabled.
`app-misc/foo[-bar,baz]`	foo must have bar disabled and baz enabled.

There are also shortcuts for conditional situations:

Compact form	Equivalent expanded form
`app-misc/foo[bar?]`	`bar? ( app-misc/foo[bar] ) !bar? ( app-misc/foo )`
`app-misc/foo[!bar?]`	`bar? ( app-misc/foo ) !bar? ( app-misc/foo[-bar] )`
`app-misc/foo[bar=]`	`bar? ( app-misc/foo[bar] ) !bar? ( app-misc/foo[-bar] )`
`app-misc/foo[!bar=]`	`bar? ( app-misc/foo[-bar] ) !bar? ( app-misc/foo[bar] )`

Legacy Inverse USE-Conditional Dependency Syntax

Once upon a time the `:` conditional operator was allowed in `*DEPEND`:

```
DEPEND="use-flag? ( app-misc/foo ) : ( app-misc/bar )"
```

This syntax is no longer permitted. It is exactly equivalent to the following, which should be used instead:

```
DEPEND="use-flag?  ( app-misc/foo )
    !use-flag? ( app-misc/bar )"
```

It is useful to recognise the legacy syntax and to know that it is no longer valid.

← CVS to RSYNC | ↑ General Concepts ↑ | Manifest →