gentoo linux™ Wiki

Gentoo Linux amd64 Handbook: Working with Gentoo

From Gentoo Wiki
Handbook:AMD64 (/wiki/Special:MyLanguage/Handbook:AMD64) | Full (/wiki/Special:MyLanguage/Handbook:AMD64/Full)

Contents

Welcome to portage

Portage is one of Gentoo's most notable innovations in software management. With its high flexibility and enormous amount of features it is frequently seen as the best software management tool available for Linux.

Portage is completely written in Python (http://www.python.org/) and Bash (http://www.gnu.org/software/bash) and therefore fully visible to the users as both are scripting languages.

Most users will work with Portage through the `emerge` tool. This chapter is not meant to duplicate the information available from the emerge man page. For a complete rundown of emerge's options, please consult the man page:

```
user $ man emerge
```

Portage tree

Ebuilds

When Gentoo's documentation talks about packages, it means software titles that are available to the Gentoo users through the portage tree. The portage tree is a collection of ebuilds, files that contain all information portage needs to maintain software (install, search, query, ...). These ebuilds reside in `/usr/portage` by default.

Whenever someone asks portage to perform some action regarding software titles, it will use the ebuilds on the system as a base. It is therefore important to regularly update the ebuilds on the system so portage knows about new software, security updates, etc.

Updating the portage tree

The portage tree is usually updated with `rsync` , a fast incremental file transfer utility. Updating is fairly simple as the `emerge` command provides a front-end for `rsync` :

```
root # emerge --sync
```

Sometimes firewall restrictions apply that prevent `rsync` from contacting the mirrors. In this case, update the portage tree through Gentoo's daily generated portage tree snapshots. The `emerge-webrsync` tool automatically fetches and installs the latest snapshot on the system:

```
root # emerge-webrsync
```

An additional advantage of using `emerge-webrsync` is that it allows the administrator to only pull in portage tree snapshots that are signed by the Gentoo release engineering GPG key. More information on this can be found in the Portage Features section on Fetching Validated Portage Tree Snapshots.

Maintaining software

Searching for software

There are multiple ways to search through the portage tree for software. One way is through `emerge` itself. By default, `emerge --search` returns the names of packages whose title matches (either fully or partially) the given search term.

For instance, to search for all packages who have "pdf" in their name:

```
user $ emerge --search pdf
```

To search through the descriptions as well, use the `--searchdesc` (or `-S`) switch:

```
user $ emerge --searchdesc pdf
```

Notice that the output returns a lot of information. The fields are clearly labelled so we won't go further into their meanings:

CODE **Example output for a search command**

```
*  net-print/cups-pdf
      Latest version available: 1.5.2
      Latest version installed: [ Not Installed ]
      Size of downloaded files: 15 kB
      Homepage:    http://cip.physik.uni-wuerzburg.de/~vrbehr/cups-pdf/
      Description: Provides a virtual printer for CUPS to produce PDF files.
      License:     GPL-2
```

Installing software

When a software title has been found, then the installation is just one `emerge` command away. For instance, to install gnumeric:

```
root # emerge --ask app-office/gnumeric
```

Since many applications depend on each other, any attempt to install a certain software package might result in the installation of several dependencies as well. Don't worry, portage handles dependencies well. To find out what portage would install, add the `--pretend` switch. For instance:

```
root # emerge --pretend gnumeric
```

During the installation of a package, portage will download the necessary source code from the Internet (if necessary) and store it by default in `/usr/portage/distfiles/`. After this it will unpack, compile and install the package. To tell portage to only download the sources without installing them, add the `--fetchonly` option to the emerge command:

```
root # emerge --fetchonly gnumeric
```

Finding installed package documentation

Many packages come with their own documentation. Sometimes, the *doc* USE flag determines whether the package documentation should be installed or not. To see if the *doc* USE flag is used by a package, use `emerge -vp PACKAGENAME` .

```
root # emerge -vp alsa-lib
```

```
...
[ebuild  N    ] media-libs/alsa-lib-1.0.14_rc1  -debug +doc 698 kB
```

The best way of enabling the doc USE flag is doing it on a per-package basis via `/etc/portage/package.use`, so that only the documentation for the wanted packages is installed. For more information, please read the USE flags chapter (/wiki/Handbook:AMD64/Working/USE).

Once the package installed, its documentation is generally found in a subdirectory named after the package under the `/usr/share/doc/` directory. It is also possible to list all installed files with the `equery` tool which is part of the app-portage/gentoolkit (http://packages.gentoo.org/package/app-portage/gentoolkit) package.

```
user $ ls -l /usr/share/doc/alsa-lib-1.0.14_rc1
```

```
total 28
-rw-r--r--  1 root root  669 May 17 21:54 ChangeLog.gz
-rw-r--r--  1 root root 9373 May 17 21:54 COPYING.gz
drwxr-xr-x  2 root root 8560 May 17 21:54 html
-rw-r--r--  1 root root  196 May 17 21:54 TODO.gz
```

user $ `equery files alsa-lib | less`

```
media-libs/alsa-lib-1.0.14_rc1
* Contents of media-libs/alsa-lib-1.0.14_rc1:
/usr
/usr/bin
/usr/bin/alsalisp
...
```

Removing software

To remove software from a system, use `emerge --unmerge`. This will tell Portage to remove all files installed by that package from the system. One exception to this are the configuration files of that application *if* they have been altered by the user. Leaving the configuration files allows users to continue working with the package without the need for reconfiguration if the packages are installed again later on.

Warning
Portage will not check if the package to remove is required by another package. It will however warn when an important package woud be removed that breaks the system if it is unmerged.

root # `emerge --unmerge gnumeric`

When a package is removed from the system, the dependencies of that package that were installed automatically when it was installed are still left on the system. To have Portage locate all dependencies that can now be removed, use emerge's `--depclean` functionality, which is documented later.

Updating the system

To keep the system in perfect shape (and not to mention install the latest security updates) it is necessary to update the system regularly. Since Portage only checks the ebuilds in the portage tree, the first thing to do is to update the Portage tree. When the Portage tree is updated, the system can be updated using `emerge --update @world`. In the next example, the `--ask` switch is also used which will tell Portage to display the list of packages it wants to upgrade and ask for confirmation:

root # `emerge --update --ask @world`

Portage will then search for newer version of the applications that are installed. However, it will only verify the versions for the applications that are explicitly installed (the applications listed in `/var/lib/portage/world`) - it does not thoroughly check their dependencies. To update the dependencies of those packages as well, add the `--deep` argument:

root # `emerge --update --deep @world`

Still, this doesn't mean all packages: some packages on the system are needed during the compile and build process of packages, but once that package is installed, these dependencies are no longer required. Portage calls those *build dependencies*. To include those in an update cycle, add `--with-bdeps=y`:

root # `emerge --update --deep --with-bdeps=y @world`

Since security updates also happen in packages that are not explicitly installed on the system (but that are pulled in as dependencies of other programs), it is recommended to run this command once in a while.

If the USE settings of the system have been altered, it is recommended to add `--newuse` as well. Portage will then verify if the change requires the installation of new packages or recompilation of existing ones:

```
root # emerge --update --deep --with-bdeps=y --newuse @world
```

Metapackages

Some packages in the Portage tree don't have any real content but are used to install a collection of packages. For instance, the kde-base/kde-meta (http://packages.gentoo.org/package/kde-base/kde-meta) package will install a complete KDE environment on the system by pulling in various KDE-related packages as dependencies.

To remove such a package from your system, running `emerge --unmerge` on the package won't have much effect as the dependencies remain on the system.

Portage has the functionality to remove orphaned dependencies as well, but since the availability of software is dynamically dependent it is important to first update the entire system fully, including the new changes applied when changing USE flags. After this one can run `emerge --depclean` to remove the orphaned dependencies. When this is done, it might be necessary to rebuild the applications that were dynamically linked to the now-removed software titles but don't require them anymore, although recently support for this has been added to portage.

All this is handled with the following three commands:

```
root # emerge --update --deep --newuse @world
root # emerge --depclean
root # revdep-rebuild
```

`revdep-rebuild` is provided by the app-portage/gentoolkit (http://packages.gentoo.org/package/app-portage/gentoolkit) package; don't forget to emerge it first:

```
root # emerge --ask app-portage/gentoolkit
```

Licenses

Beginning with Portage version 2.1.7, it is possible to accept or reject software installation based on its license. All packages in the tree contain a LICENSE entry in their ebuilds. Running `emerge --search PACKAGENAME` will show the package's license.

By default, Portage permits all licenses, except *End User License Agreements (EULAs)* that require reading and signing an acceptance agreement.

The variable that controls permitted licenses is called `ACCEPT_LICENSE`, which can be set in `/etc/portage/make.conf`. In the next example, this default value is shown:

FILE **`/etc/portage/make.conf`** **The default ACCEPT_LICENSE setting**

```
ACCEPT_LICENSE="* -@EULA"
```

With this configuration, packages that require interaction during installation to approve their EULA will not be installable. Packages without an EULA will be installable.

It is possible to set `ACCEPT_LICENSE` globally in `/etc/portage/make.conf`, or to specify it on a per-package basis in `/etc/portage/package.license`.

For example, to allow the `truecrypt-2.7` license for app-crypt/truecrypt (http://packages.gentoo.org/package/app-crypt/truecrypt), add the following to `/etc/portage/package.license`:

FILE **`/etc/portage/package.license`** **Accepting the truecrypt-2.7 license for the truecrypt package alone**

```
app-crypt/truecrypt truecrypt-2.7
```

This permits installation of truecrypt versions that have the `truecrypt-2.7` license, but not versions with the `truecrypt-2.8` license.

Important
Licenses are stored in `/usr/portage/licenses/`, and license groups are kept in `/usr/portage/profiles/license_groups`. The first entry of each line in *CAPITAL* letters is the name of the license group, and every entry after that is an individual license.

License groups defined in `ACCEPT_LICENSE` are prefixed with an `@` sign. A commonly requested setting is to only allow the installation of free software and documentation. To accomplish this, remove all currently accepted licenses (using `-*`) and then only allow the licenses in the FREE group as follows:

FILE **`/etc/portage/make.conf`** **Only accept free software and documentation**

```
ACCEPT_LICENSE="-* @FREE"
```

In this case, "free" is mostly defined by the FSF and OSI. Any package whose license does not meet these requirements will not be installable on the system.

When portage is complaining

Terminology

As stated before, portage is extremely powerful and supports many features that other software management tools lack. To understand this, we explain a few aspects of portage without going into too much detail.

With portage different versions of a single package can coexist on a system. While other distributions tend to name their package to those versions (like freetype and freetype2) portage uses a technology called *SLOT*s. An ebuild declares a certain SLOT for its version. Ebuilds with different SLOTs can coexist on the same system. For instance, the freetype package has ebuilds with SLOT="1" and SLOT="2".

There are also packages that provide the same functionality but are implemented differently. For instance, metalogd, sysklogd and syslog-ng are all system loggers. Applications that rely on the availability of "a system logger" cannot depend on, for instance, metalogd, as the other system loggers are as good a choice as any. Portage allows for virtuals: each system logger is listed as an "exclusive" dependency of the logging service in the logger virtual package of the virtual category, so that applications can depend on the virtual/logger (http://packages.gentoo.org/package/virtual/logger) package. When installed, the package will pull in the first logging package mentioned in the package, unless a logging package was already installed (in which case the virtual is satisfied).

Software in the portage tree can reside in different branches. By default the system only accepts packages that Gentoo deems stable. Most new software titles, when committed, are added to the testing branch, meaning more testing needs to be done before it is marked as stable. Although the ebuilds for those software are in the portage tree, portage will not update them before they are placed in the stable branch.

Some softwares are only available for a few architectures. Or the software doesn't work on the other architectures, or it needs more testing, or the developer that committed the software to the portage tree is unable to verify if the package works on different architectures.

Each Gentoo installation also adheres to a certain profile which contains, amongst other information, the list of packages that are required for a system to function normally.

Blocked packages

CODE **Portage warning about blocked packages (with --pretend)**

```
[blocks B     ] mail-mta/ssmtp (is blocking mail-mta/postfix-2.2.2-r1)
```

CODE **Portage warning about blocked packages (without --pretend)**

```
!!! Error: the mail-mta/postfix package conflicts with another package.
!!!        both can't be installed on the same system together.
!!!        Please use 'emerge --pretend' to determine blockers.
```

Ebuilds contain specific fields that inform Portage about its dependencies. There are two possible dependencies: build dependencies, declared in DEPEND and run-time dependencies, declared in RDEPEND. When one of these dependencies explicitly marks a package or virtual as being not compatible, it triggers a blockage.

While recent versions of Portage are smart enough to work around minor blockages without user intervention, occasionally such blockages need to be resolved manually.

To fix a blockage, users can choose to not install the package or unmerge the conflicting package first. In the given example, one can opt not to install postfix or to remove ssmtp first.

Sometimes there are also blocking packages with specific atoms, such as `<media-video/mplayer-1.0_rc1-r2`. In this case, updating to a more recent version of the blocking package could remove the block.

It is also possible that two packages that are yet to be installed are blocking each other. In this rare case, try to find out why both would need to be installed. In most cases it is sufficient to do with one of the packages alone. If not, please file a bug on Gentoo's bugtracking system.

Masked packages

CODE **Portage warning about masked packages**

```
!!! all ebuilds that could satisfy "bootsplash" have been masked.
```

CODE **Portage warning about masked packages - reason**

```
!!! possible candidates are:

- gnome-base/gnome-2.8.0_pre1 (masked by: ~x86 keyword)
- lm-sensors/lm-sensors-2.8.7 (masked by: -sparc keyword)
- sys-libs/glibc-2.3.4.20040808 (masked by: -* keyword)
- dev-util/cvsd-1.0.2 (masked by: missing keyword)
- games-fps/unreal-tournament-451 (masked by: package.mask)
- sys-libs/glibc-2.3.2-r11 (masked by: profile)
- net-im/skype-2.1.0.81 (masked by: skype-eula license(s))
```

When trying to install a package that isn't available for the system, this masking error occurs. Users should try installing a different application that is available for the system or wait until the package is marked as available. There is always a reason why a package is masked:

~arch keyword
the application is not tested sufficiently to be put in the stable branch. Wait a few days or weeks and try again.

-arch keyword or -* keyword
the application does not work on your architecture. If you believe the package does work file a bug at our bugzilla website.

missing keyword
the application has not been tested on your architecture yet. Ask the architecture porting team to test the package or test it for them and report the findings on our bugzilla website.

package.mask
the package has been found corrupt, unstable or worse and has been deliberately marked as do-not-use.

profile
the package has been found not suitable for the current profile. The application might break the system if it is installed or is just not compatible with the profile currently in use.

license
the package's license is not compatible with the ACCEPT_LICENSE setting. Permit its license or the right license group by setting it in `/etc/portage/make.conf` or in `/etc/portage/package.license`

Necessary USE flag changes

CODE **Portage warning about USE flag change requirement**

```
The following USE changes are necessary to proceed:
#required by app-text/happypackage-2.0, required by happypackage (argument)
>=app-text/feelings-1.0.0 test
```

The error message might also be displayed as follows, if `--autounmask` isn't set:

CODE **Portage error about USE flag change requirement**

```
emerge: there are no ebuilds built with USE flags to satisfy "app-text/feelings[tes
t]".
!!! One of the following packages is required to complete your request:
- app-text/feelings-1.0.0 (Change USE: +test)
(dependency required by "app-text/happypackage-2.0" [ebuild])
(dependency required by "happypackage" [argument])
```

Such warning or error occurs when a package is requested for installation which not only depends on another package, but also requires that that package is built with a particular USE flag (or set of USE flags). In the given example, the package app-text/feelings needs to be built with USE="test", but this USE flag is not set on the system.

To resolve this, either add the requested USE flag to the global USE flags in `/etc/portage/make.conf`, or set it for the specific package in `/etc/portage/package.use`.

Missing dependencies

CODE **Portage warning about missing dependency**

```
emerge: there are no ebuilds to satisfy ">=sys-devel/gcc-3.4.2-r4".

!!! Problem with ebuild sys-devel/gcc-3.4.2-r2
!!! Possibly a DEPEND/*DEPEND problem.
```

The application to install depends on another package that is not available for the system. Please check bugzilla if the issue is known and if not, please report it. Unless the system is configured to mix branches, this should not occur and is therefore a bug.

Ambiguous ebuild name

CODE **Portage warning about ambiguous ebuild names**

```
[ Results for search key : listen ]
[ Applications found : 2 ]

*  dev-tinyos/listen [ Masked ]
      Latest version available: 1.1.15
      Latest version installed: [ Not Installed ]
      Size of files: 10,032 kB
      Homepage:      http://www.tinyos.net/
      Description:   Raw listen for TinyOS
      License:       BSD

*  media-sound/listen [ Masked ]
      Latest version available: 0.6.3
      Latest version installed: [ Not Installed ]
      Size of files: 859 kB
      Homepage:      http://www.listen-project.org
      Description:   A Music player and management for GNOME
      License:       GPL-2

!!! The short ebuild name "listen" is ambiguous. Please specify
!!! one of the above fully-qualified ebuild names instead.
```

The application that is selected for installation has a name that corresponds with more than one package. Supply the category name as well to resolve this. Portage will inform the user about possible matches to choose from.

Circular dependencies

CODE **Portage warning about circular dependencies**

```
!!! Error: circular dependencies:

ebuild / net-print/cups-1.1.15-r2 depends on ebuild / app-text/ghostscript-7.05.3-r
1
ebuild / app-text/ghostscript-7.05.3-r1 depends on ebuild / net-print/cups-1.1.15-r
2
```

Two (or more) packages to install depend on each other and can therefore not be installed. This is most likely a bug in one of the packages in the portage tree. Please resync after a while and try again. It might also be beneficial to check bugzilla to see if the issue is known and if not, report it.

Fetch failed

CODE **Portage warning about fetch failed**

```
!!! Fetch failed for sys-libs/ncurses-5.4-r5, continuing...
(...)
!!! Some fetch errors were encountered.  Please see above for details.
```

Portage was unable to download the sources for the given application and will try to continue installing the other applications (if applicable). This failure can be due to a mirror that has not synchronised correctly or because the ebuild points to an incorrect location. The server where the sources reside can also be down for some reason.

Retry after one hour to see if the issue still persists.

System profile protection

CODE **Portage warning about profile-protected package**

```
!!! Trying to unmerge package(s) in system profile. 'sys-apps/portage'
!!! This could be damaging to your system.
```

The user has asked to remove a package that is part of the system's core packages. It is listed in the profile as required and should therefore not be removed from the system.

Digest verification failure

CODE **Digest verification failure**

```
>>> checking ebuild checksums
!!! Digest verification failed:
```

This is a sign that something is wrong with the portage tree -- often, it is because a developer may have made a mistake when committing a package to the tree.

When the digest verification fails, do not try to re-digest the package personally. Running `ebuild foo manifest` will not fix the problem; it will almost certainly make it worse!

Instead, wait an hour or two for the tree to settle down. It's likely that the error was noticed right away, but it can take a little time for the fix to trickle down the portage tree. Check Bugzilla and see if anyone has reported the problem yet or ask around on #gentoo (IRC). If not, go ahead and file a bug for the broken package.

Once the bug has been fixed, re-sync the portage tree to pick up the fixed digest.

Important
This does not mean that re-synchronization of the tree several times in a short time period! As stated in the rsync policy (as well as when running `emerge --sync`), users who sync too often will be banned! In fact, it's better to just wait until the next scheduled sync, so that resynchronization doesn't overload the rsync servers.

What are USE flags

The idea behind USE flags

When installing Gentoo (or any other distribution, or even operating system for that matter) users make choices depending on the environment they are working with. A setup for a server differs from a setup for a workstation. A gaming workstation differs from a 3D rendering workstation.

This is not only true for choosing what packages to install, but also what features a certain package should support. If there is no need for OpenGL, why would someone bother to install and maintain OpenGL and build OpenGL support in most of the packages? If someone doesn't want to use KDE, why would they bother compiling packages with KDE support if those packages work flawlessly without?

To help users in deciding what to install/activate and what not, Gentoo wanted the user to specify his/her environment in an easy way. This forces the user into deciding what they really want and eases the process for portage to make useful decisions.

Definition of a USE flag

Enter the USE flags. Such a flag is a keyword that embodies support and dependency-information for a certain concept. If someone defines a certain USE flag, Portage will know that they want support for the chosen keyword. Of course this also alters the dependency information for a package.

Take a look at a specific example: the kde keyword. If this keyword is not in the USE variable, all packages that have optional KDE support will be compiled *without* KDE support. All packages that have an optional KDE dependency will be installed *without* installing the KDE libraries (as dependency). When the kde keyword is defined, then those packages will be compiled *with* KDE support, and the KDE libraries *will* be installed as dependency.

By correctly defining the keywords the system will be tailored specifically to the user's needs.

What USE flags exist

There are two types of USE flags: global and local USE flags.

- A *global USE flag* is used by several packages, system-wide. This is what most people see as USE flags.
- A *local USE flag* is used by a single package to make package-specific decisions.

A list of available global USE flags can be found online (https://www.gentoo.org/dyn/use-index.xml#doc_chap1) or locally in `/usr/portage/profiles/use.desc`.

A list of available local USE flags can be found online (https://www.gentoo.org/dyn/use-index.xml#doc_chap2) or locally in `/usr/portage/profiles/use.local.desc`.

Using USE flags

Declare permanent USE flags

As previously mentioned, all USE flags are declared inside the USE variable. To make it easy for users to search and pick USE flags, we already provide a default USE setting. This setting is a collection of USE flags we think are commonly used by the Gentoo users. This default setting is declared in the `make.defaults` files that are part of the selected profile.

The profile the system listens to is pointed to by the `/etc/portage/make.profile` symlink. Each profile works on top of other profiles, and the end result is therefore the sum of all profiles. The top profile is the base profile (`/usr/portage/profiles/base`).

To view the currently active USE flags (completely), use `emerge --info`:

```
root # emerge --info | grep ^USE
```

```
USE="a52 aac acpi alsa branding cairo cdr dbus dts ..."
```

As can be seen, this variable already contains quite a lot of keywords. Do not alter any `make.defaults` file to tailor the USE variable to personal needs though: changes in these file will be undone when the portage tree is updated!

To change this default setting, add or remove keywords to/from the USE variable. This is done globally by defining the USE variable in `/etc/portage/make.conf`. In this variable one can add the extra USE flags required, or remove the USE flags that are no longer needed. This latter is done by prefixing the keyword with the minus-sign (`-`).

For instance, to remove support for KDE and QT but add support for ldap, the following USE can be defined in `/etc/portage/make.conf`:

FILE `/etc/portage/make.conf` **Updating USE in make.conf**

```
USE="-kde -qt4 ldap"
```

Declaring USE flags for individual packages

Sometimes users want to declare a certain USE flag for one (or a couple) of applications but not system-wide. To accomplish this, edit `/etc/portage/package.use`. This is usually a single file, but can also be a directory; see `man portage` for more information. The following examples assume `package.use` is a single file.

For instance, to only have berkdb support for mysql:

FILE `/etc/portage/package.use` **Enabling berkdb support for MySQL**

```
dev-db/mysql berkdb
```

Similarly it is possible to explicitly disable USE flags for a certain application. For instance, to disable java support in PHP (but have it for all other packages through the USE flag declaration in `make.conf`):

FILE `/etc/portage/package.use` **Disable java support in PHP**

```
dev-php/php -java
```

Declaring temporary USE flags

Sometimes users need to set a USE flag for a brief moment. Instead of editing `/etc/portage/make.conf` twice (to do and undo the USE changes) just declare the USE variable as an environment variable. Remember that this setting only applies for the command entered - re-emerging or updating this application (either explicitly or as part of a system update) will undo the changes that were triggered through the (temporary) USE flag definition.

The following example temporarily removes java from the USE setting during the installation of seamonkey:

```
root # USE="-java" emerge seamonkey
```

Precedence

Of course there is a certain precedence on what setting has priority over the USE setting. The precedence for the USE setting is, ordered by priority (first has lowest priority):

1. Default USE setting declared in the `make.defaults` files part of your profile
2. User-defined USE setting in `/etc/portage/make.conf`
3. User-defined USE setting in `/etc/portage/package.use`
4. User-defined USE setting as environment variable

To view the final USE setting as seen by Portage, run `emerge --info`. This will list all relevant variables (including the USE variable) with their current definition as known to portage.

```
root # emerge --info
```

Adapting the entire system to the new USE flags

After having altered USE flags, the system should be updated to reflect the necessary changes. To do so, use the `--newuse` option with `emerge`:

```
root # emerge --update --deep --newuse @world
```

Next, run portage's depclean to remove the conditional dependencies that were emerged on the "old" system but that have been obsoleted by the new USE flags.

Warning
Running `emerge --depclean` is a dangerous operation and should be handled with care. Double-check the provided list of "obsoleted" packages to make sure it doesn't remove packages that are needed. In the following example we add the `-p` switch to have depclean only list the packages without removing them.

```
root # emerge -p --depclean
```

When depclean has finished, run `revdep-rebuild` to rebuild the applications that are dynamically linked against shared objects provided by possibly removed packages. `revdep-rebuild` is part of the app-portage/gentoolkit (http://packages.gentoo.org/package/app-portage/gentoolkit) package; don't forget to emerge it first.

```
root # revdep-rebuild
```

When all this is accomplished, the system is using the new USE flag settings.

Package specific USE flags

Viewing available USE flags

Let's take the example of seamonkey: what USE flags does it listen to? To find out, we use `emerge` with the `--pretend` and `--verbose` options:

```
root # emerge --pretend --verbose seamonkey
```

```
These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild   R   ] www-client/seamonkey-1.0.7  USE="crypt gnome java -debug -ipv6
-ldap -mozcalendar -mozdevelop -moznocompose -moznoirc -moznomail -moznopango
-moznoroaming -postgres -xinerama -xprint" 0 kB
```

`emerge` isn't the only tool for this job. In fact, there is a tool dedicated to package information called `equery` which resides in the app-portage/gentoolkit (http://packages.gentoo.org/package/app-portage/gentoolkit) package

```
root # emerge --ask app-portage/gentoolkit
```

Now run equery with the *uses* argument to view the USE flags of a certain package. For instance, for the gnumeric package:

```
user $ equery --nocolor uses =gnumeric-1.6.3 -a
```

```
[ Searching for packages matching =gnumeric-1.6.3... ]
[ Colour Code : set unset ]
[ Legend : Left column  (U) - USE flags from make.conf              ]
[        : Right column (I) - USE flags packages was installed with ]
[ Found these USE variables for app-office/gnumeric-1.6.3 ]
 U I
 - - debug  : Enable extra debug codepaths, like asserts and extra output.
              If you want to get meaningful backtraces see
              http://www.gentoo.org/proj/en/qa/backtraces.xml .
 + + gnome  : Adds GNOME support
 + + python : Adds support/bindings for the Python language
 - - static : !!do not set this during bootstrap!! Causes binaries to be
              statically linked instead of dynamically
```

Satisfying REQUIRED_USE conditions

Some ebuilds require or forbid certain combinations of USE flags in order to work properly. This is expressed via a `REQUIRED_USE` condition. This condition ensures that all features and dependencies are complete, the build will succeed and perform as expected. If any of these are not met, emerge will alert you and ask you to fix the issue.

Some example conditions are given below.

Example	Description
`REQUIRED_USE="foo? ( bar )"`	If `foo` is set, `bar` must be set
`REQUIRED_USE="foo? ( !bar )"`	If `foo` is set, `bar` must not be set
`REQUIRED_USE="foo? ( || ( bar baz ) )"`	If `foo` is set, `bar` or `baz` must be set
`REQUIRED_USE="^^ ( foo bar baz )"`	Exactly one of `foo` `bar` or `baz` must be set
`REQUIRED_USE="|| ( foo bar baz )"`	At least one of `foo` `bar` or `baz` must be set
`REQUIRED_USE="?? ( foo bar baz )"`	No more than one of `foo` `bar` or `baz` may be set

Portage features

Portage has several additional features that makes the Gentoo experience even better. Many of these features rely on certain software tools that improve performance, reliability, security, ...

To enable or disable certain Portage features, edit `/etc/portage/make.conf` and update or set the `FEATURES` variable which contains the various feature keywords, separated by white space. In several cases it will also be necessary to install the additional tool on which the feature relies.

Not all features that Portage supports are listed here. For a full overview, please consult the `make.conf` man page:

```
user $ man make.conf
```

To find out what `FEATURES` are set by default, run `emerge --info` and search for the `FEATURES` variable or grep it out:

```
user $ emerge --info | grep ^FEATURES=
```

Distributed compiling

Using distcc

`distcc` is a program to distribute compilations across several, not necessarily identical, machines on a network. The distcc client sends all necessary information to the available distcc servers (running distccd) so they can compile pieces of source code for the client. The net result is a faster compilation time.

More information about distcc (and how to have it work with Gentoo) can be found in the Distcc (/wiki/Distcc) article.

Installing distcc

Distcc ships with a graphical monitor to monitor tasks that the computer is sending away for compilation. This tool is automatically installed if USE=gnome or USE=gtk is set.

```
root # emerge --ask sys-devel/distcc
```

Activating portage distcc support

Add distcc to the FEATURES variable inside `/etc/portage/make.conf`. Next, edit the MAKEOPTS variable and increase the number of parallel build jobs that the system allows. A known guideline is to fill in "-jX" with X the number of CPUs that run distccd (including the current host) plus one, but that is just a guideline.

Now run `distcc-config` and enter the list of available distcc servers. For a simple example assume that the available DistCC servers are 192.168.1.102 (the current host), 192.168.1.103 and 192.168.1.104 (two "remote" hosts):

```
root # distcc-config --set-hosts "192.168.1.102 192.168.1.103 192.168.1.104"
```

Don't forget to run the distccd daemon as well:

```
root # rc-update add distccd default
root # /etc/init.d/distccd start
```

Caching compilation objects

About ccache

`ccache` is a fast compiler cache. Whenever an application is compiled, it will cache intermediate results so that, whenever the same program is recompiled, the compilation time is greatly reduced. The first time ccache is run, it will be much slower than a normal compilation. Subsequent recompiles however should be faster. ccache is only helpful if the same application will be recompiled many times (or upgrades of the same application are happening frequently); thus it's mostly only useful for software developers.

For more information about ccache, please visit its homepage (http://ccache.samba.org/).

Warning
ccache is known to cause numerous compilation failures. Sometimes ccache will retain stale code objects or corrupted files, which can lead to packages that cannot be emerged. If this happens (errors like "File not recognized: File truncated" come up in build logs), try recompiling the application with ccache disabled (FEATURES="-ccache" in `/etc/portage/make.conf`) before reporting a bug.

Installing ccache

To install ccache, run `emerge ccache` :

```
root # emerge --ask dev-util/ccache
```

Activating portage ccache support

Open `/etc/portage/make.conf` and add *ccache* to the FEATURES variable. Next, add a new variable called CCACHE_SIZE and set it to "2G":

FILE `/etc/portage/make.conf` **Enabling portage ccache support**

```
FEATURES="... ccache ..."
CCACHE_SIZE="2G"
```

To check if ccache functions, ask ccache to provide its statistics. Because Portage uses a different ccache home directory, it is necessary to temporarily set the CCACHE_DIR variable:

```
root # CCACHE_DIR="/var/tmp/ccache" ccache -s
```

The `/var/tmp/ccache/` location is Portage' default ccache home directory; it can be changed by setting the CCACHE_DIR variable in `/etc/portage/make.conf`.

When running `ccache` standalone, it would use the default location of `${HOME}/.ccache/`, which is why the CCACHE_DIR variable needs to be set when asking for the (Portage) ccache statistics.

Using ccache outside portage

To use ccache for non-Portage compilations, add `/usr/lib/ccache/bin/` to the beginning of the PATH variable (before `/usr/bin`). This can be accomplished by editing `~/.bash_profile` in the user's home directory. Using `~/.bash_profile` is one way to define PATH variables.

FILE `~/.bash_profile` **Setting the ccache location before any other PATH**

```
PATH="/usr/lib/ccache/bin:/opt/bin:${PATH}"
```

Binary package support

Creating prebuilt packages

Portage supports the installation of prebuilt packages. Even though Gentoo does not provide prebuilt packages by itself Portage can be made fully aware of prebuilt packages.

To create a prebuilt package use `quickpkg` if the package is already installed on the system, or emerge with the `--buildpkg` or `--buildpkgonly` options.

To have portage create prebuilt packages of every single package that gets installed, add *buildpkg* to the FEATURES variable.

More extended support for creating prebuilt package sets can be obtained with catalyst. For more information on catalyst please read the Catalyst Frequently Asked Questions (https://www.gentoo.org/proj/en/releng/catalyst/faq.xml).

Installing prebuilt packages

Although Gentoo doesn't provide one, it is possible to create a central repository where prebuilt packages are stored. In order to use this repository, it is necessary to make Portage aware of it by having the PORTAGE_BINHOST variable point to it. For instance, if the prebuilt packages are on ftp://buildhost/gentoo (ftp://buildhost/gentoo):

FILE `/etc/portage/make.conf` **Add PORTAGE_BINHOST location**

```
PORTAGE_BINHOST="ftp://buildhost/gentoo"
```

To install a prebuilt package, add the `--getbinpkg` option to the emerge command alongside of the `--usepkg` option. The former tells emerge to download the prebuilt package from the previously defined server while the latter asks emerge to try to install the prebuilt package first before fetching the sources and compiling it.

For instance, to install gnumeric with prebuilt packages:

```
root # emerge --usepkg --getbinpkg gnumeric
```

More information about emerge's prebuilt package options can be found in the emerge man page:

```
user $ man emerge
```

Distributing prebuilt packages to others

If prebuilt packages are to be distributed to others, then make sure that this is permitted. Check the distribution terms of the upstream package for this. For example, for a package released under the GNU GPL, sources must be made available along with the binaries.

Ebuilds may define a `bindist` restriction in their `RESTRICT` variable if built binaries are not distributable. Sometimes this restriction is conditional on one or more USE flags.

By default, Portage will not mask any packages because of restrictions. This can be changed globally by setting the `ACCEPT_RESTRICT` variable in `/etc/portage/make.conf`. For example, to mask packages that have a `bindist` restriction, add the following line to `make.conf`:

FILE `/etc/portage/make.conf` **Only accept binary distributable packages**

```
ACCEPT_RESTRICT="* -bindist"
```

It is also possible to override the `ACCEPT_RESTRICT` variable by passing the `--accept-restrict` option to the `emerge` command. For example, `--accept-restrict=-bindist` will temporarily mask packages with a `bindist` restriction.

Also consider setting the `ACCEPT_LICENSE` variable when distributing packages. See the Licenses section (/wiki/Handbook:AMD64/Working/Portage#Licenses) for this.

Important
It is entirely the responsibility of the user to comply with packages' license terms and with his country's laws. Metadata variables defined by ebuilds like `RESTRICT` or `LICENSE` can only give some guidance when distribution of binaries would not be allowed. They are *not* a legal statement, so don't rely on them.

Fetching files

Userfetch

When Portage is run as root, FEATURES="userfetch" will allow Portage to drop root privileges while fetching package sources. This is a small security improvement.

Pulling validated Gentoo ebuild tree snapshots

Administrators can opt to update the local Gentoo ebuild tree with a cryptographically validated tree snapshot as released by the Gentoo infrastructure. This ensures that no rogue rsync mirror is adding unwanted code or packages to the tree the system is downloading.

Note
The following is an updated method for setting up and using the emerge-webrsync sync method.

The Gentoo release media OpenPGP keys are now available as a binary keyring, installed via the app-crypt/gentoo-keys (http://packages.gentoo.org/package/app-crypt/gentoo-keys) package.

```
root # emerge app-crypt/gentoo-keys
```

This will install the keyring to the `/var/lib/gentoo/gkeys/keyrings/gentoo/release` location.

FILE `/etc/portage/make.conf` **Enabling GPG support in Portage**

```
FEATURES="webrsync-gpg"
PORTAGE_GPG_DIR="/var/lib/gentoo/gkeys/keyrings/gentoo/release"
```

FILE `/etc/portage/repos.conf/gentoo.conf` **Clear the sync-uri variable**

```
[DEFAULT]
main-repo = gentoo

[gentoo]
# Disable synchronization by clearing the values or setting auto-sync = no
# Do not set value of the variables in this configuration file using quotes ('' or
"")!
# For portage-2.2.18 use 'websync'
# For portage-2.2.19 and greater use 'webrsync' (websync was renamed to webrsync)
sync-type = websync
sync-uri =
auto-sync = yes
```

Make sure that app-crypt/gnupg (http://packages.gentoo.org/package/app-crypt/gnupg) package is installed:

```
root # emerge --ask app-crypt/gnupg
```

Use gpg to verify that the keys in the keyring are the correct keys:

```
root # gpg --homedir /var/lib/gentoo/gkeys/keyrings/gentoo/release --with-fingerprint --list-keys
```

Verify the fingerprints of the key(s) against those listed here: Project:RelEng#Release_security_and_signing (/wiki/Project:RelEng#Release_security_and_signing)
Repeat the following command for each key you wish to trust. (Substitute the keyid '0x...' for the desired key you wish to trust.)

```
root # gpg --homedir /var/lib/gentoo/gkeys/keyrings/gentoo/release --edit-key 0xDB6B8C1F96D8BF6D trust
```

The system is now set-up to sync using only OpenPGP/gpg verified snapshots.
Several command options are available to perform the sync.

Note
Only one of the following commands is needed to sync. See the Portage's sync wiki article (/wiki/Project:Portage/Sync) for more details.

```
root # emerge --sync
root # emaint sync -a
root # emaint sync --repo gentoo
root # emerge-webrsync
```

Original install and configuration instructions

To configure Portage to validate the snapshots, first create a truststore in which the keys of the Gentoo Infrastructure responsible for signing the Portage tree snapshots are stored. Of course, this OpenPGP key can be validated as per the proper guidelines (/wiki/GnuPG#Importing_keys) (like checking the key fingerprint). The list of OpenPGP keys used by the release engineering team is available on their project page (https://www.gentoo.org/proj/en/releng/index.xml).

```
root # mkdir -p /etc/portage/gpg
root # chmod 0700 /etc/portage/gpg
```

Make sure that app-crypt/gnupg (http://packages.gentoo.org/package/app-crypt/gnupg) is installed:

```
root # emerge --ask app-crypt/gnupg
```

In the next set of commands, substitute the keys with those mentioned on the release engineering site:

```
root # gpg --homedir /etc/portage/gpg --keyserver subkeys.pgp.net --recv-keys 0xDB6B8C1F96D8BF6D
root # gpg --homedir /etc/portage/gpg --edit-key 0xDB6B8C1F96D8BF6D trust
```

Next, edit `/etc/portage/make.conf` and enable support for validating the signed portage tree snapshots (using FEATURES="webrsync-gpg") and disabling updating the Portage tree using the regular `emerge --sync` method.

FILE `/etc/portage/make.conf` **Enabling GPG support in portage**

```
FEATURES="webrsync-gpg"
PORTAGE_GPG_DIR="/etc/portage/gpg"
```

In the `/etc/portage/repos.conf/gentoo.conf` file, clear the `sync-type` and `sync-uri` variables so that `emerge --sync` no longer works (and thus no longer pulls in ebuilds that come from a potentially non-validated source):

FILE `/etc/portage/repos.conf/gentoo.conf` **Clear the sync-type and sync-uri variables**

```
[DEFAULT]
main-repo = gentoo

[gentoo]
# Disable synchronization by clearing the values
# Do not set value of the variables in this configuration file using quotes ('' or
"")!
sync-type =
sync-uri =
```

That is it. Next time `emerge-webrsync` is ran, only the snapshots with a valid signature will be expanded on the filesystem.

Runlevels

Booting the system

When the system is booted, lots of text floats by. When paying close attention, one will notice this text is (usually) the same every time the system is rebooted. The sequence of all these actions is called the boot sequence and is (more or less) statically defined.

First, the boot loader will load the kernel image that is defined in the boot loader configuration. Then, the boot loader instructs the CPU to run execute kernel. When the kernel is loaded and run, it initializes all kernel-specific structures and tasks and starts the init process.

This process then makes sure that all filesystems (defined in `/etc/fstab`) are mounted and ready to be used. Then it executes several scripts located in `/etc/init.d/`, which will start the services needed in order to have a successfully booted system.

Finally, when all scripts are executed, init activates the terminals (in most cases just the virtual consoles which are hidden beneath `Alt+F1`, `Alt+F2`, etc.) attaching a special process called `agetty` to it. This process will then make sure users are able to log on through these terminals by running login.

Initscripts

Now init doesn't just execute the scripts in `/etc/init.d/` randomly. Even more, it doesn't run all scripts in `/etc/init.d/`, only the scripts it is told to execute. It decides which scripts to execute by looking into `/etc/runlevels/`.

First, init runs all scripts from `/etc/init.d/` that have symbolic links inside `/etc/runlevels/boot/`. Usually, it will start the scripts in alphabetical order, but some scripts have dependency information in them, telling the system that another script must be run before they can be started.

When all `/etc/runlevels/boot/` referenced scripts are executed, init continues with running the scripts that have a symbolic link to them in `/etc/runlevels/default/`. Again, it will use the alphabetical order to decide what script to run first, unless a script has dependency information in it, in which case the order is changed to provide a valid start-up sequence. The latter is also the reason why commands used during the installation of Gentoo Linux used *default*, as in `rc-update add sshd default` .

How init works

Of course init doesn't decide all that by itself. It needs a configuration file that specifies what actions need to be taken. This configuration file is `/etc/inittab`.

Remember the boot sequence that was just described -- init's first action is to mount all filesystems. This is defined in the following line from `/etc/inittab`:

FILE `/etc/inittab` **Initialization command**

```
si::sysinit:/sbin/rc sysinit
```

This line tells init that it must run `/sbin/rc sysinit` to initialize the system. The `/sbin/rc` script takes care of the initialization, so one might say that init doesn't do much -- it delegates the task of initializing the system to another process.

Second, init executed all scripts that had symbolic links in `/etc/runlevels/boot/`. This is defined in the following line:

FILE `/etc/inittab` **Boot command invocation**

```
rc::bootwait:/sbin/rc boot
```

Again the rc script performs the necessary tasks. Note that the option given to rc (boot) is the same as the subdirectory of `/etc/runlevels/` that is used.

Now init checks its configuration file to see what runlevel it should run. To decide this, it reads the following line from `/etc/inittab`:

FILE `/etc/inittab` **Default runlevel selection**

```
id:3:initdefault:
```

In this case (which the majority of Gentoo users will use), the runlevel id is 3. Using this information, init checks what it must run to start runlevel 3:

FILE `/etc/inittab` **Runlevel definitions**

```
l0:0:wait:/sbin/rc shutdown
l1:S1:wait:/sbin/rc single
l2:2:wait:/sbin/rc nonetwork
l3:3:wait:/sbin/rc default
l4:4:wait:/sbin/rc default
l5:5:wait:/sbin/rc default
l6:6:wait:/sbin/rc reboot
```

The line that defines level 3, again, uses the rc script to start the services (now with argument *default*). Again note that the argument of rc is the same as the subdirectory from `/etc/runlevels/`.

When rc has finished, init decides what virtual consoles it should activate and what commands need to be run at each console:

FILE `/etc/inittab` **Terminal definitions**

```
c1:12345:respawn:/sbin/agetty 38400 tty1 linux
c2:12345:respawn:/sbin/agetty 38400 tty2 linux
c3:12345:respawn:/sbin/agetty 38400 tty3 linux
c4:12345:respawn:/sbin/agetty 38400 tty4 linux
c5:12345:respawn:/sbin/agetty 38400 tty5 linux
c6:12345:respawn:/sbin/agetty 38400 tty6 linux
```

Available runlevels

In a previous section, we saw that init uses a numbering scheme to decide what runlevel it should activate. A runlevel is a state in which the system is running and contains a collection of scripts (runlevel scripts or initscripts) that must be executed when entering or leaving a runlevel.

In Gentoo, there are seven runlevels defined: three internal runlevels, and four user-defined runlevels. The internal runlevels are called *sysinit*, *shutdown* and *reboot* and do exactly what their names imply: initialize the system, powering off the system and rebooting the system.

The user-defined runlevels are those with an accompanying `/etc/runlevels/` subdirectory: *boot*, *default*, *nonetwork* and *single*. The boot runlevel starts all system-necessary services which all other runlevels use. The remaining three runlevels differ in what services they start: default is used for day-to-day operations, nonetwork is used in case no network connectivity is required, and single is used when the system needs to be fixed.

Working with initscripts

The scripts that the rc process starts are called init scripts. Each script in `/etc/init.d/` can be executed with the arguments *start*, *stop*, *restart*, *zap*, *status*, *ineed*, *iuse*, *needsme*, *usesme* or *broken*.

To start, stop or restart a service (and all depending services), start, stop and restart should be used:

```
root # /etc/init.d/postfix start
```

Note
Only the services that need the given service are stopped or restarted. The other depending services (those that use the service but don't need it) are left untouched.

To stop a service, but not the services that depend on it, use the `--nodeps` argument together with the stop command:

```
root # /etc/init.d/postfix --nodeps stop
```

To see what status a service has (started, stopped, ...) use the status argument:

```
root # /etc/init.d/postfix status
```

If the status information shows that the service is running, but in reality it is not, then reset the status information to "stopped" with the zap argument:

```
root # /etc/init.d/postfix zap
```

To also ask what dependencies the service has, use *iuse* or *ineed*. With *ineed* it is possible to see the services that are really necessary for the correct functioning of the service. *iuse* on the other hand shows the services that can be used by the service, but are not necessary for the correct functioning.

```
root # /etc/init.d/postfix ineed
```

Similarly, it is possible to ask what services require the service (*needsme*) or can use it (*usesme*):

```
root # /etc/init.d/postfix needsme
```

Finally, to ask what dependencies the service requires that are missing:

```
root # /etc/init.d/postfix broken
```

Updating runlevels

rc-update

Gentoo's init system uses a dependency-tree to decide what service needs to be started first. As this is a tedious task that we wouldn't want our users to have to do manually, we have created tools that ease the administration of the runlevels and init scripts.

With `rc-update` it is possible to add and remove init scripts to a runlevel. The `rc-update` tool will then automatically ask the depscan.sh script to rebuild the dependency tree.

Adding and removing services

In earlier instructions, init scripts have already been added to the "default" runlevel. What "default" means has been explained earlier in this document. Next to the runlevel, the rc-update script requires a second argument that defines the action: *add*, *del* or *show*.

To add or remove an init script, just give `rc-update` the *add* or *del* argument, followed by the init script and the runlevel. For instance:

```
root # rc-update del postfix default
```

The `rc-update -v show` command will show all the available init scripts and list at which runlevels they will execute:

```
root # rc-update -v show
```

It is also possible to run `rc-update show` (without `-v`) to just view enabled init scripts and their runlevels.

Configuring services

Why additional configuration is needed

Init scripts can be quite complex. It is therefore not really desirable to have the users edit the init script directly, as it would make it more error-prone. It is however important to be able to configure such a service. For instance, users might want to give more options to the service itself.

A second reason to have this configuration outside the init script is to be able to update the init scripts without the fear that the user's configuration changes will be undone.

conf.d directory

Gentoo provides an easy way to configure such a service: every init script that can be configured has a file in `/etc/conf.d/`. For instance, the apache2 initscript (called `/etc/init.d/apache2`) has a configuration file called `/etc/conf.d/apache2`, which can contain the options to give to the Apache 2 server when it is started:

FILE **`/etc/conf.d/apache2`** **Example options for apache2 init script**

```
APACHE2_OPTS="-D PHP5"
```

Such a configuration file contains *only* variables (just like `/etc/portage/make.conf` does), making it very easy to configure services. It also allows us to provide more information about the variables (as comments).

Writing initscripts

Is it necessary

No, writing an init script is usually not necessary as Gentoo provides ready-to-use init scripts for all provided services. However, some users might have installed a service without using portage, in which case they will most likely have to create an init script.

Do not use the init script provided by the service if it isn't explicitly written for Gentoo: Gentoo's init scripts are not compatible with the init scripts used by other distributions!

Layout

The basic layout of an init script is shown below.

CODE **Example initscript layout**

```
#!/sbin/runscript

depend() {
  (Dependency information)
}

start() {
  (Commands necessary to start the service)
}

stop() {
  (Commands necessary to stop the service)
}
```

Every init script requires the *start()* function to be defined. All other sections are optional.

Dependencies

There are two dependency-alike settings that can be defined which influence the start-up or sequencing of init scripts: *use* and *need*. Next to these two, there are also two order-influencing methods called *before* and *after*. These last two are no dependencies per se - they do not make the original init script fail if the selected one isn't scheduled to start (or fails to start).

- The *use* settings informs the init system that this script uses functionality offered by the selected script, but does not directly depend on it. A good example would be *use logger* or *use dns*. If those services are available, they will be put in good use, but if the system does not have a logger or DNS server the services will still work. If the services exist, then they are started before the script that uses them.
- The *need* setting is a hard dependency. It means that the script that is needing another script will not start before the other script is launched successfully. Also, if that other script is restarted, then this one will be restarted as well.
- When using *before*, then the given script is launched before the selected one if the selected one is part of the init level. So an init script xdm that defines *before alsasound* will start before the alsasound script, but only if alsasound is scheduled to start as well in the same init level. If alsasound is not scheduled to start too, then this particular setting has no effect and xdm will be started when the init system deems it most appropriate.
- Similarly, *after* informs the init system that the given script should be launched after the selected one if the selected one is part of the init level. If not, then the setting has no effect and the script will be launched by the init system when it deems it most appropriate.

It should be clear from the above that *need* is the only "true" dependency setting as it affects if the script will be started or not. All the others are merely pointers towards the init system to clarify in which order scripts can be (or should be) launched.

Now, look at many of Gentoo's available init scripts and notice that some have dependencies on things that are no init scripts. These "things" we call virtuals.

A *virtual dependency* is a dependency that a service provides, but that is not provided solely by that service. An init script can depend on a system logger, but there are many system loggers available (metalogd, syslog-ng, sysklogd, ...). As the script cannot need every single one of them (no sensible system has all these system loggers installed and running) we made sure that all these services provide a virtual dependency.

For instance, take a look at the postfix dependency information:

FILE `/etc/init.d/postfix` **Dependency information of the postfix service**

```
depend() {
  need net
  use logger dns
  provide mta
}
```

As can be seen, the postfix service:

- requires the (virtual) net dependency (which is provided by, for instance, `/etc/init.d/net.eth0`)
- uses the (virtual) logger dependency (which is provided by, for instance, `/etc/init.d/syslog-ng`)
- uses the (virtual) dns dependency (which is provided by, for instance, `/etc/init.d/named`)
- provides the (virtual) mta dependency (which is common for all mail servers)

Controlling the order

As described in the previous section, it is possible to tell the init system what order it should use for starting (or stopping) scripts. This ordering is handled both through the dependency settings use and need, but also through the order settings before and after. As we have described these earlier already, let's take a look at the portmap service as an example of such init script.

FILE `/etc/init.d/portmap` **Dependency information of the portmap service**

```
depend() {
  need net
  before inetd
  before xinetd
}
```

It is possible to use the "*" glob to catch all services in the same runlevel, although this isn't advisable.

CODE **Using the * glob**

```
depend() {
  before *
}
```

If the service must write to local disks, it should need localmount. If it places anything in `/var/run/` such as a pidfile, then it should start after bootmisc:

CODE **Dependency setting with needing localmount and after bootmisc**

```
depend() {
  need localmount
  after bootmisc
}
```

Standard functions

Next to the *depend()* function, it is also necessary to define the *start()* function. This one contains all th

Next to the *depend()* functionality, it is also necessary to define the *start()* function. This one contains all the commands necessary to initialize the service. It is advisable to use the *ebegin* and *eend* functions to inform the user about what is happening:

CODE **Example start() function**

```
start() {
  if [ "${RC_CMD}" = "restart" ];
  then
    # Do something in case a restart requires more than stop, start
  fi

  ebegin "Starting my_service"
  start-stop-daemon --start --exec /path/to/my_service \
    --pidfile /path/to/my_pidfile
  eend $?
}
```

Both --exec and --pidfile should be used in start and stop functions. If the service does not create a pidfile, then use --make-pidfile if possible, though it is recommended to test this to be sure. Otherwise, don't use pidfiles. It is also possible to add --quiet to the start-stop-daemon options, but this is not recommended unless the service is extremely verbose. Using --quiet may hinder debugging if the service fails to start.

Another notable setting used in the above example is to check the contents of the RC_CMD variable. Unlike the previous init script system, the newer openrc system does not support script-specific restart functionality. Instead, the script needs to check the contents of the RC_CMD variable to see if a function (be it start() or stop()) is called as part of a restart or not.

Note
Make sure that --exec actually calls a service and not just a shell script that launches services and exits -- that's what the init script is supposed to do.

For more examples of the start() function, please read the source code of the available init scripts in the `/etc/init.d/` directory.

Another function that can (but does not have to) be defined is stop(). The init system is intelligent enough to fill in this function by itself if start-stop-daemon is used.

CODE **Example stop() function**

```
stop() {
  ebegin "Stopping my_service"
  start-stop-daemon --stop --exec /path/to/my_service \
    --pidfile /path/to/my_pidfile
  eend $?
}
```

If the service runs some other script (for example, bash, python, or perl), and this script later changes names (for example, foo.py to foo), then it is necessary to add --name to start-stop-daemon. This must specify the name that the script will be changed to. In this example, a service starts foo.py, which changes names to foo:

CODE **Example definition for a service that starts the foo script**

```
start() {
  ebegin "Starting my_script"
  start-stop-daemon --start --exec /path/to/my_script \
    --pidfile /path/to/my_pidfile --name foo
  eend $?
}
```

start-stop-daemon has an excellent man page available if more information is needed:

```
user $ man start-stop-daemon
```

Gentoo's init script syntax is based on the POSIX Shell so people are free to use sh-compatible constructs inside their init scripts. Keep other constructs, like bash-specific ones, out of the init scripts to ensure that the scripts remain functional regardless of the change Gentoo might do on its init system.

Adding custom options

If the initscript needs to support more options than the ones we have already encountered, then add the option to the extra_commands variable, and create a function with the same name as the option. For instance, to support an option called restartdelay:

CODE **Example definition of restartdelay method**

```
extra_commands="restartdelay"

restartdelay() {
  stop
  sleep 3    # Wait 3 seconds before starting again
  start
}
```

Important
The function restart() cannot be overridden in openrc!

Service configuration variables

In order to support configuration files in `/etc/conf.d/`, no specifics need to be impleneted: when the init script is executed, the following files are automatically sourced (i.e. the variables are available to use):

- `/etc/conf.d/YOUR_INIT_SCRIPT`
- `/etc/conf.d/basic`
- `/etc/rc.conf`

Also, if the init script provides a virtual dependency (such as net), the file associated with that dependency (such as `/etc/conf.d/net`) will be sourced too.

Changing runlevel behavior

Who might benefit from this

Many laptop users know the situation: at home they need to start net.eth0, but they don't want to start net.eth0 while on the road (as there is no network available). With Gentoo the runlevel behaviour can be altered at will.

For instance, a second "default" runlevel can be created which can be booted that has other init scripts assigned to it. At boottime, the user can then select what default runlevel to use.

Using softlevel

First of all, create the runlevel directory for the second "default" runlevel. As an example we create the *offline* runlevel:

```
root # mkdir /etc/runlevels/offline
```

Add the necessary init scripts to the newly created runlevel. For instance, to have an exact copy of the current default runlevel but without net.eth0:

```
root # cd /etc/runlevels/default
root # for service in *; do rc-update add $service offline; done
root # rc-update del net.eth0 offline
root # rc-update show offline
```

```
(Partial sample Output)
               acpid | offline
          domainname | offline
               local | offline
            net.eth0 |
```

Even though net.eth0 has been removed from the offline runlevel, udev might want to attempt to start any devices it detects and launch the appropriate services, a functionality that is called hotplugging. By default, Gentoo does not enable hotplugging.

To enable hotplugging, but only for a selected set of scripts, use the `rc_hotplug` variable in `/etc/rc.conf`:

FILE `/etc/rc.conf` **Enable hotplugging of the WLAN interface**

```
rc_hotplug="net.wlan !net.*"
```

Note
For more information on device initiated services, please see the comments inside `/etc/rc.conf`.

Edit the bootloader configuration and add a new entry for the offline runlevel. In that entry, add `softlevel=offline` as a boot parameter.

Using bootlevel

Using bootlevel is completely analogous to softlevel. The only difference here is that a second "boot" runlevel is defined instead of a second "default" runlevel.

Environment variables

Introduction

An environment variable is a named object that contains information used by one or more applications. Many users (and especially those new to Linux) find this a bit weird or unmanageable. However, this is a mistake: by using environment variables one can easily change a configuration setting for one or more applications.

Important examples

The following table lists a number of variables used by a Linux system and describes their use. Example values are presented after the table.

Variable	Description
PATH	This variable contains a colon-separated list of directories in which the system looks for executable files. If a name is entered of an executable (such as ls, rc-update or emerge) but this executable is not located in a listed directory, then the system will not execute it (unless the full path is entered as the command, such as /bin/ls).
ROOTPATH	This variable has the same function as PATH, but this one only lists the directories that should be checked when the root-user enters a command.
LDPATH	This variable contains a colon-separated list of directories in which the dynamical linker searches through to find a library.
MANPATH	This variable contains a colon-separated list of directories in which the man command searches for the man pages.
INFODIR	This variable contains a colon-separated list of directories in which the info command searches for the info pages.
PAGER	This variable contains the path to the program used to list the contents of files through (such as less or more).
EDITOR	This variable contains the path to the program used to change the contents of files with (such as nano or vi).
KDEDIRS	This variable contains a colon-separated list of directories which contain KDE-specific material.
CONFIG_PROTECT	This variable contains a space-delimited list of directories which should be protected by Portage during updates.
CONFIG_PROTECT_MASK	This variable contains a space-delimited list of directories which should not be protected by Portage during updates.

Below is an example definition of all these variables:

CODE **Example settings for the mentioned variables**

```
PATH="/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/games/bin"
ROOTPATH="/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin"
LDPATH="/lib:/usr/lib:/usr/local/lib:/usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.3"
MANPATH="/usr/share/man:/usr/local/share/man"
INFODIR="/usr/share/info:/usr/local/share/info"
PAGER="/usr/bin/less"
EDITOR="/usr/bin/vim"
KDEDIRS="/usr"
CONFIG_PROTECT="/usr/X11R6/lib/X11/xkb /opt/tomcat/conf \
                /usr/kde/3.1/share/config /usr/share/texmf/tex/generic/config/ \
                /usr/share/texmf/tex/platex/config/ /usr/share/config"
CONFIG_PROTECT_MASK="/etc/gconf"
```

Defining variables globally

The env.d directory

To centralize the definitions of these variables, Gentoo introduced the `/etc/env.d/` directory. Inside this directory a number of files are available, such as 00basic, 05gcc, etc. which contain the variables needed by the application mentioned in their name.

For instance, when gcc is installed, a file called 05gcc was created by the ebuild which contains the definitions of the following variables:

FILE `/etc/env.d/05gcc` **Default gcc enabled environment variables**

```
PATH="/usr/i686-pc-linux-gnu/gcc-bin/3.2"
ROOTPATH="/usr/i686-pc-linux-gnu/gcc-bin/3.2"
MANPATH="/usr/share/gcc-data/i686-pc-linux-gnu/3.2/man"
INFOPATH="/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info"
CC="gcc"
CXX="g++"
LDPATH="/usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.3"
```

Other distributions might tell their users to change or add such environment variable definitions in `/etc/profile` or other locations. Gentoo on the other hand makes it easy for the user (and for portage) to maintain and manage the environment variables without having to pay attention to the numerous files that can contain environment variables.

For instance, when gcc is updated, the `/etc/env.d/05gcc` file is updated too without requesting any user-interaction.

This not only benefits portage, but also the user. Occasionally users might be asked to set a certain environment variable system-wide. As an example we take the http_proxy variable. Instead of messing about with `/etc/profile`, users can now just create a file (say `/etc/env.d/99local`) and enter the definition(s) in it:

FILE `/etc/env.d/99local` **Setting a global variable**

```
http_proxy="proxy.server.com:8080"
```

By using the same file for all self-managed variables, users have a quick overview on the variables they have defined themselves.

env-update

Several files in `/etc/env.d/` define the PATH variable. This is not a mistake: when `env-update` is executed, it will append the several definitions before it updates the environment variables, thereby making it easy for packages (or users) to add their own environment variable settings without interfering with the already existing values.

The env-update script will append the values in the alphabetical order of the `/etc/env.d/` files. The file names must begin with two decimal digits.

CODE **Update order used by env-update**

```
00basic        99kde-env       99local
     +-------------+----------------+-------------+
PATH="/bin:/usr/bin:/usr/kde/3.2/bin:/usr/local/bin"
```

The concatenation of variables does not always happen, only with the following variables: ADA_INCLUDE_PATH, ADA_OBJECTS_PATH, CLASSPATH, KDEDIRS, PATH, LDPATH, MANPATH, INFODIR, INFOPATH, ROOTPATH, CONFIG_PROTECT, CONFIG_PROTECT_MASK, PRELINK_PATH, PRELINK_PATH_MASK, PKG_CONFIG_PATH and PYTHONPATH. For all other variables the latest defined value (in alphabetical order of the files in `/etc/env.d/`) is used.

It is possible to add more variables into this list of concatenate-variables by adding the variable name to either COLON_SEPARATED or SPACE_SEPARATED variables (also inside an `/etc/env.d/` file).

When executing `env-update` , the script will create all environment variables and place them in `/etc/profile.env` (which is used by `/etc/profile`). It will also extract the information from the LDPATH variable and use that to create `/etc/ld.so.conf`. After this, it will run `ldconfig` to recreate the `/etc/ld.so.cache` file used by the dynamical linker.

To notice the effect of env-update immediately after running it, execute the following command to update the environment. Users who have installed Gentoo themselves will probably remember this from the installation instructions:

```
root # env-update && source /etc/profile
```

Note
The above command only updates the variables in the current terminal, new consoles, and their children. Thus, if the user is working in X11, he needs to either type `source /etc/profile` in every new terminal opened or restart X so that all new terminals source the new variables. If a login manager is used, it is necessary to become root and restart the `/etc/init.d/xdm` service.

Important
It is not possible to use shell variables when defining other variables. This means things like FOO="$BAR" (where $BAR is another variable) are forbidden.

Defining variables locally

User specific

It might not be necessary to define an environment variable globally. For instance, one might want to add `/home/my_user/bin` and the current working directory (the directory the user is in) to the PATH variable but don't want all other users on the system to have that in their PATH too. To define an environment variable locally, use `~/.bashrc` or `~/.bash_profile`:

FILE `~/.bashrc` **Extending PATH for local usage**

```
# A colon followed by no directory is treated as the current working directory
PATH="${PATH}:/home/my_user/bin:"
```

After logout/login, the PATH variable will be updated.

Session specific

Sometimes even stricter definitions are requested. For instance, a user might want to be able to use binaries from a temporary directory created without using the path to the binaries themselves or editing `~/.bashrc` for the short time necessary.

In this case, just define the PATH variable in the current session by using the `export` command. As long as the user does not log out, the PATH variable will be using the temporary settings.

```
root # export PATH="${PATH}:/home/my_user/tmp/usr/bin"
```

(http://wiki.gentoo.org/index.php?title=Handbook:AMD64/Full/Working&oldid=181110)"

- This page was last modified on 13 December 2014, at 18:45.