Gentoo Linux amd64 Handbook: Working with Portage

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

Contents

Portage files

Configuration directives

Portage comes with a default configuration stored in `/usr/share/portage/config/make.globals`. All portage configuration is handled through variables. What variables portage listens to and what they mean are described later.

Since many configuration directives differ between architectures, portage also has default configuration files which are part of the system profile. This profile is pointed to by the `/etc/portage/make.profile` symlink; portage' configurations are set in the `make.defaults` files of the profile and all parent profiles. We'll explain

more about profiles and the `/etc/portage/make.profile` directory later on.

When changing a configuration variable, don't alter `/usr/share/portage/config/make.globals` or `make.defaults`. Instead use `/etc/portage/make.conf` which has precedence over the previous files. For more information, read the `/usr/share/portage/config/make.conf.example`. As the name implies, this is merely an example file - portage does not read in this file.

It is also possible to define a portage configuration variable as an environment variable, but we don't recommend this.

Profile-specific information

We've already encountered the `/etc/portage/make.profile` directory. Well, this isn't exactly a directory but a symbolic link to a profile, by default one inside `/usr/portage/profiles/` although one can create their own profiles elsewhere and point to them. The profile this symlink points to is the profile to which the system adheres.

A profile contains architecture-specific information for portage, such as a list of packages that belong to the system corresponding with that profile, a list of packages that don't work (or are masked-out) for that profile, etc.

User-specific configuration

When portage's behavior needs to be changed regarding the installation of software, then the right set of files inside `/etc/portage/` will need to be changed. It is highly recommended to use files within `/etc/portage/` and highly discouraged to override the behavior through environment variables!

Within `/etc/portage/` users can create the following files:

- `package.mask` which lists the packages that portage should never try to install
- `package.unmask` which lists the packages portage should be able to install even though the Gentoo developers highly discourage users from emerging them
- `package.accept_keywords` which lists the packages portage should be able to install even though the package hasn't been found suitable for the system or architecture (yet)
- `package.use` which lists the USE flags to use for certain packages without having the entire system use those USE flags

These don't have to be files; they can also be directories that contain one file per package. More information about the `/etc/portage/` directory and a full list of possible files that can be created can be found in the Portage man page:

```
user $ man portage
```

Changing portage file and directory locations

The previously mentioned configuration files cannot be stored elsewhere - portage will always look for those configuration files at those exact locations. However, portage uses many other locations for various purposes: build directory, source code storage, Portage tree location, ...

All these purposes have well-known default locations but can be altered to personal taste through `/etc/portage/make.conf`. The rest of this chapter explains what special-purpose locations Portage uses and how to alter their placement on the filesystem.

This document isn't meant to be used as a reference though. To get 100% coverage, please consult the portage and make.conf man pages:

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

Storing files

Portage tree

The portage tree default location is `/usr/portage`. This is defined by the PORTDIR variable. When storing the portage tree elsewhere (by altering this variable), don't forget to change the `/etc/portage/make.profile` symbolic link accordingly.

After changing the PORTDIR variable, it is recommended to alter the following variables as well since they will not notice the PORTDIR change. This is due to how Portage handles variables: PKGDIR, DISTDIR, RPMDIR.

Prebuilt binaries

Even though Portage doesn't use prebuilt binaries by default, it has extensive support for them. When asking Portage to work with prebuilt packages, it will look for them in `/usr/portage/packages`. This location is defined by the PKGDIR variable.

Source code

Application source code is stored in `/usr/portage/distfiles` by default. This location is defined by the DISTDIR variable.

Portage database

Portage stores the state of the system (what packages are installed, what files belong to which package, ...) in `/var/db/pkg`.

> **Warning**
> Do not alter these files manually! It might break portage's knowledge of the system.

Portage cache

The Portage cache (with modification times, virtuals, dependency tree information, ...) is stored in `/var/cache/edb`. This location really is a cache: users can clean it if they are not running any portage-related application at that moment.

Building software

Temporary portage files

Portage's temporary files are stored in `/var/tmp/` by default. This is defined by the PORTAGE_TMPDIR variable.

When altering the PORTAGE_TMPDIR variable, it is recommended to also change the following variables as well since they will not notice the PORTAGE_TMPDIR change. This is due to how Portage handles variables: BUILD_PREFIX.

Building directory

Portage creates specific build directories for each package it emerges inside `/var/tmp/portage/`. This location is defined by the BUILD_PREFIX variable.

Live filesystem location

By default Portage installs all files on the current filesystem (/), but this can be changed by setting the ROOT environment variable. This is useful when creating new build images.

Logging features

Ebuild logging

Portage can create per-ebuild log files, but only when the PORT_LOGDIR variable is set to a location that is writable by portage (through the portage user). By default this variable is unset. If PORT_LOGDIR is not set, then there will not be any build logs with the current logging system, though users may receive some logs from the new `elog` support.

If PORT_LOGDIR is not defined and `elog` is used, then build logs and any other logs saved by elog will be made available, as explained below.

Portage offers fine-grained control over logging through the use of elog:

- PORTAGE_ELOG_CLASSES: This is where users can define what kinds of messages to be logged. This can be any space-separated combination of info, warn, error, log, and qa.
  - info: Logs "einfo" messages printed by an ebuild
  - warn: Logs "ewarn" messages printed by an ebuild
  - error: Logs "eerror" messages printed by an ebuild
  - log: Logs the "elog" messages found in some ebuilds
  - qa: Logs the "QA Notice" messages printed by an ebuild
- PORTAGE_ELOG_SYSTEM: This selects the module(s) to process the log messages. If left empty, logging is disabled. Any space-separated combination of save, custom, syslog, mail, save_summary, and mail_summary can be used. At least one module must be used in order to use elog.
  - save: This saves one log per package in `$PORT_LOGDIR/elog`, or `/var/log/portage/elog` if $PORT_LOGDIR is not defined.
  - custom: Passes all messages to a user-defined command in $PORTAGE_ELOG_COMMAND; this will be discussed later.
  - syslog: Sends all messages to the installed system logger.
  - mail: Passes all messages to a user-defined mailserver in $PORTAGE_ELOG_MAILURI; this will be discussed later. The mail features of elog require >=portage-2.1.1.
  - save_summary: Similar to save, but it merges all messages in `$PORT_LOGDIR/elog/summary.log`, or `/var/log/portage/elog/summary.log` if $PORT_LOGDIR is not defined.
  - mail_summary: Similar to mail, but it sends all messages in a single mail when emerge exits.
- PORTAGE_ELOG_COMMAND: This is only used when the custom module is enabled. Users can specify a command to process log messages. Note that the command can make use of two variables: ${PACKAGE} is the package name and version, while ${LOGFILE} is the absolute path to the logfile. For instance:

CODE **Example PORTAGE_ELOG_COMMAND definition**

```
PORTAGE_ELOG_COMMAND="/path/to/logger -p '\${PACKAGE}' -f '\${LOGFILE}'"
```

- PORTAGE_ELOG_MAILURI: This contains settings for the mail module such as address, user, password, mailserver, and port number. The default setting is "root@localhost localhost". The following is an example for an smtp server that requires username and password-based authentication on a particular port (the default is port 25):

CODE **Example PORTAGE_ELOG_MAILURI definition**

```
PORTAGE_ELOG_MAILURI="user@some.domain username:password@smtp.some.domain:995"
```

- PORTAGE_ELOG_MAILFROM: Allows the user to set the "from" address of log mails; defaults to "portage" if unset.
- PORTAGE_ELOG_MAILSUBJECT: Allows the user to create a subject line for log mails. Note that it can make

use of two variables: ${PACKAGE} will display the package name and version, while ${HOST} is the fully qualified domain name of the host Portage is running on. For instance:

CODE **Example PORTAGE_ELOG_MAILSUBJECT definition**

```
PORTAGE_ELOG_MAILSUBJECT="package \${PACKAGE} was merged on \${HOST} with some mess
ages"
```

Important
Users who used enotice with Portage-2.0.* must completely remove enotice, as it is incompatible with elog.

Portage configuration

As noted previously, Portage is configurable through many variables which should be defined in `/etc/portage/make.conf` or one of the subdirectories of `/etc/portage/`. Please refer to the `make.conf` and portage man pages for more and complete information:

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

Build-specific options

Configure and compiler options

When Portage builds applications, it passes the contents of the following variables to the compiler and configure script:

- CFLAGS & CXXFLAGS define the desired compiler flags for C and C++ compiling.
- CHOST defines the build host information for the application's configure script
- MAKEOPTS is passed to the make command and is usually set to define the amount of parallelism used during the compilation. More information about the make options can be found in the make man page.

The USE variable is also used during configure and compilations but has been explained in great detail in previous chapters.

Merge options

When portage has merged a newer version of a certain software title, it will remove the obsoleted files of the older version from the system. Portage gives the user a 5 second delay before unmerging the older version. These 5 seconds are defined by the CLEAN_DELAY variable.

It is possible to tell emerge to use certain options every time it is run by setting EMERGE_DEFAULT_OPTS. Some useful options would be --ask, --verbose, --tree, and so on.

Configuration file protection

Portage protected locations

Portage overwrites files provided by newer versions of a software title if the files aren't stored in a protected location. These protected locations are defined by the CONFIG_PROTECT variable and are generally configuration file locations. The directory listing is space-delimited.

A file that would be written in such a protected location is renamed and the user is warned about the presence of a newer version of the (presumable) configuration file.

To find out about the current CONFIG_PROTECT setting, use the emerge --info output:

```
user $ emerge --info | grep 'CONFIG_PROTECT='
```

More information about portage's Configuration File Protection is available in the CONFIGURATION FILES section of the emerge manpage:

```
user $ man emerge
```

Excluding directories

To 'unprotect' certain subdirectories of protected locations users can use the CONFIG_PROTECT_MASK variable.

Download options

Server locations

When the requested information or data is not available on the system, portage will retrieve it from the Internet. The server locations for the various information and data channels are defined by the following variables:

- GENTOO_MIRRORS defines a list of server locations which contain source code (distfiles)
- PORTAGE_BINHOST defines a particular server location containing prebuilt packages for the system

A third setting involves the location of the rsync server which users use to update their portage tree. This is defined in the `/etc/portage/repos.conf` file (or a file inside that directory if it is defined as a directory):

- sync-type defines the type of server and defaults to "rsync"
- sync-uri defines a particular server which Portage uses to fetch the Portage tree from

The GENTOO_MIRRORS, sync-type and sync-uri variables can be set automatically through the mirrorselect application. Of course, app-portage/mirrorselect (http://packages.gentoo.org/package/app-portage/mirrorselect) needs to be installed first before it can be used. For more information, see mirrorselect's online help:

```
user $ mirrorselect --help
```

If the environment requires the use of a proxy server, then the http_proxy, ftp_proxy and RSYNC_PROXY variables can be declared.

Fetch commands

When portage needs to fetch source code, it uses wget by default. This can be changed through the FETCHCOMMAND variable.

Portage is able to resume partially downloaded source code. It uses wget by default, but this can be altered through the RESUMECOMMAND variable.

Make sure that the FETCHCOMMAND and RESUMECOMMAND store the source code in the correct location. Inside the variables the \${URI} and \${DISTDIR} variables can be used to point to the source code location and distfiles location respectively.

It is also possible to define protocol-specific handlers with FETCHCOMMAND_HTTP, FETCHCOMMAND_FTP, RESUMECOMMAND_HTTP, RESUMECOMMAND_FTP, and so on.

Rsync settings

It is not possible to alter the rsync command used by Portage to update the Portage tree, but it is possible to set some variables related to the rsync command:

- PORTAGE_RSYNC_OPTS sets a number of default variables used during sync, each space-separated. These shouldn't be changed unless you know exactly what you're doing. Note that certain absolutely required options will always be used even if PORTAGE_RSYNC_OPTS is empty.
- PORTAGE_RSYNC_EXTRA_OPTS can be used to set additional options when syncing. Each option should be space separated.
  - --timeout=<number>: This defines the number of seconds an rsync connection can idle before rsync sees the connection as timed-out. This variable defaults to 180 but dialup users or individuals with slow computers might want to set this to 300 or higher.
  - --exclude-from=/etc/portage/rsync_excludes: This points to a file listing the packages and/or categories rsync should ignore during the update process. In this case, it points to `/etc/portage/rsync_excludes`.
  - --quiet: Reduces output to the screen
  - --verbose: Prints a complete filelist
  - --progress: Displays a progress meter for each file
- PORTAGE_RSYNC_RETRIES defines how many times rsync should try connecting to the mirror pointed to by the SYNC variable before bailing out. This variable defaults to 3.

For more information on these options and others, please read man rsync.

Gentoo configuration

Branch selection

It is possible to change the default branch with the ACCEPT_KEYWORDS variable. It defaults to the architecture's stable branch. More information on Gentoo's branches can be found in the next chapter.

Portage features

It is possible to activate certain portage features through the FEATURES variable. The portage features have been discussed in previous chapters.

Portage behavior

Resource management

With the PORTAGE_NICENESS variable users can augment or reduce the nice value portage runs with. The PORTAGE_NICENESS value is added to the current nice value.

For more information about nice values, see the nice man page:

```
user $ man nice
```

Output behavior

The NOCOLOR variable, which defaults to "false", defines if Portage should disable the use of colored output.

Using one branch

Stable

The ACCEPT_KEYWORDS variable defines what software branch to use on the system. It defaults to the stable software branch for the system's architecture, for instance amd64.

It is recommended to stick with the stable branch. However, if stability is not that much important and/or the administrator wants to help out Gentoo by submitting bugreports to https://bugs.gentoo.org (https://bugs.gentoo.org), then the testing branch can be used instead.

Testing

To use more recent software, users can consider using the testing branch instead. To have Portage use the testing branch, add a ~ in front of the architecture.

The testing branch is exactly what it says - Testing. If a package is in testing, it means that the developers feel that it is functional but has not been thoroughly tested. Users using the testing branch might very well be the first to discover a bug in the package in which case they should file a bugreport to let the developers know about it.

Beware though; using the testing branch might incur stability issues, imperfect package handling (for instance wrong/missing dependencies), too frequent updates (resulting in lots of building) or broken packages. Users that do not know how Gentoo works and how to solve problems, we recommend to stick with the stable and tested branch.

For example, to select the testing branch for the amd64 architecture, edit `/etc/portage/make.conf` and set:

FILE `/etc/portage/make.conf` **Using the testing branch**

```
ACCEPT_KEYWORDS="~amd64"
```

When changing from stable to testing, users will find out that lots of packages will be updated. Keep in mind that, after moving to the testing branch, it might be challenging to go back to the stable branch.

Mixing stable with testing

package.accept_keywords

It is possible to ask portage to allow the testing branch for particular packages but use the stable branch for the rest of the system. To achieve this, add the package category and name in `/etc/portage/package.accept_keywords`. It is also possible to create a directory (with the same name) and list the package in the files under that directory.

For instance, to use the testing branch for gnumeric:

FILE `/etc/portage/package.accept_keywords` **Use the testing branch for just the gnumeric application**

```
app-office/gnumeric
```

Testing particular versions

To use a specific software version from the testing branch but don't want portage to use the testing branch for subsequent versions, add in the version in the `package.accept_keywords` location. In this case use the = operator. It is also possible to enter a version range using the <=, <, > or >= operators.

In any case, if version information is added, an operator *must* be used. Without version information, an operator cannot be used.

In the following example we ask portage to allow installing gnumeric-1.2.13 even when it is in the testing branch:

FILE `/etc/portage/package.accept_keywords` **Allow specific version selection**

```
=app-office/gnumeric-1.2.13
```

Masked packages

package.unmask

Important
The Gentoo developers do not support the use of unmasking packages. Please exercise due caution when doing so. Support requests related to package.unmask and/or package.mask might not be answered.

When a package has been masked by the Gentoo developers, yet despite the reason mentioned in the package.mask file (situated in `/usr/portage/profiles/` by default) a user still wants to use this package, then add the desired version (usually this will be the exact same line from the `package.mask` file in the profile) to the `/etc/portage/package.unmask` file (or in a file in that directory if it is a directory).

For instance, if =net-mail/hotwayd-0.8 is masked, then it can be unmasked by adding the exact same line in the `package.unmask` location:

FILE `/etc/portage/package.unmask` **Unmasking a particular package/version**

```
=net-mail/hotwayd-0.8
```

Note
If an entry in `/usr/portage/profiles/package.mask` contains a range of package versions, then it is necessary to unmask only the version(s) that are actually needed. Please read the previous section to learn how to specify versions.

package.mask

It is also possible to ask portage not to take a certain package or a specific version of a package into account. To do so, mask the package by adding an appropriate line to the `/etc/portage/package.mask` location (either in that file or in a file in this directory).

For instance, to prevent portage from installing kernel sources newer than gentoo-sources-2.6.8.1, add the following line at the `package.mask` location:

FILE `/etc/portage/package.mask` **Mask gentoo-sources with a version greater than 2.6.8.1**

```
>sys-kernel/gentoo-sources-2.6.8.1
```

dispatch-conf

`dispatch-conf` is a tool that aids in merging the `._cfg0000_<name>` files. `._cfg0000_<name>` files are generated by portage when it wants to overwrite a file in a directory protected by the CONFIG_PROTECT variable.

With `dispatch-conf`, users are able to merge updates to their configuration files while keeping track of all changes. `dispatch-conf` stores the differences between the configuration files as patches or by using the RCS revision system. This means that if someone makes a mistake when updating a config file, the administrator can revert the file to the previous version at any time.

When using `dispatch-conf`, users can ask to keep the configuration file as-is, use the new configuration file, edit the current one or merge the changes interactively. `dispatch-conf` also has some nice additional features:

- Automatically merge configuration file updates that only contain updates to comments
- Automatically merge configuration files which only differ in the amount of whitespace

Edit `/etc/dispatch-conf.conf` first and create the directory referenced by the archive-dir variable. Then, execute `dispatch-conf`:

```
root # dispatch-conf
```

When running `dispatch-conf`, each changed config file will pass the revue one at a time. Press `u` to update (replace) the current config file with the new one and continue to the next file. Press `w` to zap (delete) the new config file and continue to the next file. Once all config files have been taken care of, `dispatch-conf` will exit. At any time, `q` can be used to exit the application as well.

For more information, check out the dispatch-conf man page. It describes how to interactively merge current and new config files, edit new config files, examine differences between files, and more.

```
user $ man dispatch-conf
```

etc-update

Another tool to merge configuration files is `etc-update`. It's not as simple to use as `dispatch-conf`, nor as featureful, but it does provide an interactive merging setup and can also auto-merge trivial changes.

However, unlike `dispatch-conf`, `etc-update` does not preserve the old versions of the config files. Once a file is updated, the old version is gone forever! So be very careful, as using `etc-update` is significantly less safe than using `dispatch-conf` in that regard.

```
root # etc-update
```

After merging the straightforward changes, a list of protected files will be provided that have an update waiting. At the bottom the possible options are shown:

CODE **Options presented by etc-update**

```
Please select a file to edit by entering the corresponding number.
              (-1 to exit) (-3 to auto merge all remaining files)
                           (-5 to auto-merge AND not use 'mv -i'):
```

With -1, `etc-update` will exit and discontinue any other changes. With -3 or -5, all listed configuration files will be overwritten with the newer versions. It is therefore very important to first select the configuration files that should not be automatically updated. This is simply a matter of entering the number listed to the left of that configuration file.

As an example, we select the configuration file `/etc/pear.conf`:

CODE **Updating a specific configuration file**

```
Beginning of differences between /etc/pear.conf and /etc/._cfg0000_pear.conf
[...]
End of differences between /etc/pear.conf and /etc/._cfg0000_pear.conf
1) Replace original with update
2) Delete update, keeping original as is
3) Interactively merge original with update
4) Show differences again
```

The differences between the two files are shown. If the updated configuration file can be used without problems, enter 1. If the updated configuration file isn't necessary, or doesn't provide any new or useful information, enter 2. If the current configuration file has to be interactively updated, enter 3.

There is no point in further elaborating the interactive merging here. For completeness sake, we will list the possible commands that can be used while interactively merging the two files. Users are greeted with two lines (the original one, and the proposed new one) and a prompt at which the user can enter one of the following commands:

CODE **Commands available for the interactive merging**

```
ed:     Edit then use both versions, each decorated with a header.
eb:     Edit then use both versions.
el:     Edit then use the left version.
er:     Edit then use the right version.
e:      Edit a new version.
l:      Use the left version.
r:      Use the right version.
s:      Silently include common lines.
v:      Verbosely include common lines.
q:      Quit.
```

After having finished updating the important configuration files, users can then automatically update all the other configuration files. `etc-update` will exit if it doesn't find any more updateable configuration files.

quickpkg

With `quickpkg` users can create archives of the packages that are already merged on the system. These archives can be used as prebuilt packages. Running `quickpkg` is straightforward: just add the names of the packages to archive.

For instance, to archive curl, orage, and procps:

```
root # quickpkg curl orage procps
```

The prebuilt packages will be stored in $PKGDIR (`/usr/portage/packages/` by default). These packages are placed in `$PKGDIR/CATEGORY`.

Using a subset of the portage tree

Excluding packages and categories

It is possible to selectively update certain categories/packages and ignore the other categories/packages. This can be achieved by having rsync exclude categories/packages during the `emerge --sync` step.

Define the name of the file that contains the exclude patterns in the PORTAGE_RSYNC_EXTRA_OPTS variable in `/etc/portage/make.conf`:

FILE `/etc/portage/make.conf` **Defining the exclude file**

```
PORTAGE_RSYNC_EXTRA_OPTS="--exclude-from=/etc/portage/rsync_excludes"
```

FILE `/etc/portage/rsync_excludes` **Excluding all games**

```
games-*/*
```

Note however that this may lead to dependency issues since new, allowed packages might depend on new but excluded packages.

Adding unofficial ebuilds

Defining a portage overlay directory

It is possible to ask portage to use ebuilds that are not officially available through the portage tree. Create a new directory (for instance `/usr/local/portage`) in which to store the 3rd-party ebuilds. Use the same directory structure as the official Portage tree!

Then define PORTDIR_OVERLAY in `/etc/portage/make.conf` and have it point to the previously defined directory. When using portage now, it will take those ebuilds into account as well without removing/overwriting those ebuilds the next time `emerge --sync` is ran.

Working with several overlays

For the powerusers who develop on several overlays, test packages before they hit the portage tree or just want to use unofficial ebuilds from various sources, the app-portage/layman (http://packages.gentoo.org/package/app-portage/layman) package brings `layman` , a tool to help users keep the overlay repositories up to date.

First install and configure `layman` as shown in the Overlays Users' Guide (https://www.gentoo.org/proj/en/overlays/userguide.xml), and add the desired repositories with `layman -a` .

For instance, to enable the hardened-development overlay:

```
root # layman -a hardened-development
```

Regardless of how many repositories are used through `layman` , all the repositories can be updated with the following command:

```
root # layman -S
```

For more information on working with overlays, please read `man layman` and the previously linked layman/overlay users' guide.

Non-portage maintained software

Using portage with self-maintained software

Sometimes users want to configure, install and maintain software individually without having Portage automate the process, even though Portage can provide the software titles. Known cases are kernel sources and nvidia drivers. It is possible to configure Portage so it knows that a certain package is manually installed on the system (and thus take this information into account when calculating dependencies). This process is called *injecting* and is supported by portage through the `/etc/portage/profile/package.provided` file.

For instance, to inform portage about gentoo-sources-2.6.11.6 which has been installed manually, add the following line to `/etc/portage/profile/package.provided`:

FILE `/etc/portage/package.provided` **Marking gentoo-sources-2.6.11.6 as manually installed**

```
sys-kernel/gentoo-sources-2.6.11.6
```

Note
This is a file that uses versions *without* an operator.

Introduction

For most users, the information received thus far is sufficient for all their Linux operations. But Portage is capable of much more; many of its features are for advanced users or only applicable in specific corner cases. Still, that would not be an excuse not to document them.

Of course, with lots of flexibility comes a huge list of potential cases. It will not be possible to document them all here. Instead, we hope to focus on some generic issues which can then be bended to fit personal needs. More tweaks and tips can be found all over the Gentoo Wiki.

Most, if not all of these additional features can be easily found by digging through the manual pages that portage provides:

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

Finally, know that these are advanced features which, if not worked with correctly, can make debugging and troubleshooting very difficult. Make sure to mention these when hitting a bug and opening a bugreport.

Per-package environment variables

Using /etc/portage/env

By default, package builds will use the environment variables defined in `/etc/portage/make.conf`, such as CFLAGS, MAKEOPTS and more. In some cases though, it might be beneficial to provide different variables for specific packages. To do so, Portage supports the use of `/etc/portage/env` and `/etc/portage/package.env`.

The `/etc/portage/package.env` file contains the list of packages for which deviating environment variables are needed as well as a specific identifier that tells portage which changes to make. The identifier name is free format, and portage will look for the variables in the `/etc/portage/env/IDENTIFIER` file.

Example: Using debugging for specific packages

As an example, we enable debugging for the media-video/mplayer (http://packages.gentoo.org/package/media-video/mplayer) package.

First of all, set the debugging variables in a file called `/etc/portage/env/debug-cflags`. The name is arbitrarily chosen, but of course reflects the reason of the deviation to make it more obvious later why a deviation was put in.

FILE `/etc/portage/env/debug-cflags` **Specific variables for debugging**

```
CFLAGS="-O2 -ggdb -pipe"
FEATURES="${FEATURES} nostrip"
```

Next, we tag the media-video/mplayer (http://packages.gentoo.org/package/media-video/mplayer) package to use this content:

FILE `/etc/portage/package.env` **Using debug-cflags for the mplayer package**

```
media-video/mplayer debug-cflags
```

Hooking in the emerge process

Using /etc/portage/bashrc and affiliated files

When portage works with ebuilds, it uses a bash environment in which it calls the various build functions (like src_prepare, src_configure, pkg_postinst, etc.). But portage also allows users to set up a specific bash environment.

The advantage of using a specific bash environment is that it allows users to hook in the emerge process during each step it performs. This can be done for every emerge (through `/etc/portage/bashrc`) or by using per-package environments (through `/etc/portage/env` as discussed earlier).

To hook in the process, the bash environment can listen to the variables EBUILD_PHASE, CATEGORY as well as the variables that are always available during ebuild development (such as P, PF, ...). Based on the values of these variables, additional steps/functions can then be executed.

Example: Updating the file database

In this example, we'll use `/etc/portage/bashrc` to call some file database applications to ensure their databases are up to date with the system. The applications used in the example are `aide` (an intrusion detection tool) and `updatedb` (to use with `mlocate`), but these are meant as examples. Do *not* consider this as a HOWTO for AIDE ;-)

To use `/etc/portage/bashrc` for this case, we need to "hook" in the postrm (after removal of files) and postinst (after installation of files) functions, because that is when the files on the file system have been changed.

FILE `/etc/portage/bashrc` **Hooking into the postinst and postrm phases**

```
if [ "${EBUILD_PHASE}" == "postinst" ] || [ "${EBUILD_PHASE}" == "postrm" ];
then
  echo ":: Calling aide --update to update its database";
  aide --update;
  echo ":: Calling updatedb to update its database";
  updatedb;
fi
```

Executing tasks after --sync

Using /etc/portage/postsync.d location

Until now we've talked about hooking into the ebuild processes. However, portage also has another important function: updating the portage tree. In order to run tasks after updating the portage tree, put a script inside `/etc/portage/postsync.d` and make sure it is marked as executable.

Example: Running eix-update

If `eix-sync` was not used to update the tree, then it is still possible to update the eix database after running `emerge --sync` (or `emerge-webrsync`) by putting a symlink to `/usr/bin/eix` called `eix-update` in `/etc/portage/postsync.d`.

```
root # ln -s /usr/bin/eix /etc/portage/postsync.d/eix-update
```

Note
If a different name would be chosen, then it needs to be a script that calls `/usr/bin/eix-update` instead.
The eix binary looks at how it has been called to find out which function it has to execute. If a symlink would be created that points to eix yet isn't called eix-update, it will not run correctly.

Overriding profile settings

Using /etc/portage/profile

By default, Gentoo uses the settings contained in the profile pointed to by `/etc/portage/make.profile` (a symbolic link to the right profile directory). These profiles define both specific settings as well as inherit settings from other profiles (through their parent file).

By using `/etc/portage/profile`, users can override profile settings such as packages (what packages are considered to be part of the system set), forced use flags and more.

Example: Adding nfs-utils to the system set

When using an NFS-based file systems for rather critical file systems, it might be necessary to mark net-fs/nfs-utils (http://packages.gentoo.org/package/net-fs/nfs-utils) as a system package, causing portage to heavily warn administrators if they would attempt to unmerge it.

To accomplish that, we add the package to `/etc/portage/profile/packages`, prepended with a *:

FILE `/etc/portage/profile/packages` **Marking nfs-utils as a system package**

```
*net-fs/nfs-utils
```

Applying non-standard patches

Using epatch_user

To manage several ebuilds in a similar manner, ebuild developers use eclasses (sort-of shell libraries) that define commonly used functions. One of these eclasses is {Path|eutils.eclass}} which offers an interesting function called epatch_user.

The epatch_user function applies source code patches that are found in `/etc/portage/patches/<category>/<package>[-<version>[-<revision>]]`, whatever directory is found first. Sadly, not all ebuilds automatically call this function so just putting a patch in this location might not always work.

Luckily, with the information provided earlier in this chapter, users can call this function by hooking into, for instance, the prepare phase. The function can be called as many times as necessary - it will only apply the patches once.

Example: Applying patches to firefox

The www-client/firefox (http://packages.gentoo.org/package/www-client/firefox) package is one of the many that already call epatch_user from within the ebuild, so there is no need to override anything specific.

If for some reason (for instance because a developer provided a patch and asked to check if it fixes the bug reported) patching firefox is wanted, all that is needed is to put the patch in `/etc/portage/patches/www-client/firefox` (probably best to use the full name and version so that the patch does not interfere with later versions) and rebuild firefox.