NAME

cvs − Concurrent Versions System support files

NOTE

This documentation may no longer be up to date. Please consult the Cederqvist (CVS Manual) as specified in **cvs(1)**.

SYNOPSIS

$CVSROOT/CVSROOT/commitinfo,v

$CVSROOT/CVSROOT/cvsignore,v

$CVSROOT/CVSROOT/cvswrappers,v

$CVSROOT/CVSROOT/editinfo,v

$CVSROOT/CVSROOT/history

$CVSROOT/CVSROOT/loginfo,v

$CVSROOT/CVSROOT/modules,v

$CVSROOT/CVSROOT/rcsinfo,v

$CVSROOT/CVSROOT/taginfo,v

DESCRIPTION

cvs is a system for providing source control to hierarchical collections of source directories. Commands and procedures for using **cvs** are described in **cvs(1)**.

cvs manages *source repositories*, the directories containing master copies of the revision-controlled files, by copying particular revisions of the files to (and modifications back from) developers' private *working directories*. In terms of file structure, each individual source repository is an immediate subdirectory of **$CVSROOT**.

The files described here are supporting files; they do not have to exist for **cvs** to operate, but they allow you to make **cvs** operation more flexible.

You can use the ' modules ' file to define symbolic names for collections of source maintained with **cvs**. If there is no ' modules ' file, developers must specify complete path names (absolute, or relative to **$CVSROOT**) for the files they wish to manage with **cvs** commands.

You can use the ' commitinfo ' file to define programs to execute whenever ' **cvs commit** ' is about to execute. These programs are used for ''pre-commit'' checking to verify that the modified, added, and removed files are really ready to be committed. Some uses for this check might be to turn off a portion (or all) of the source repository from a particular person or group. Or, perhaps, to verify that the changed files conform to the site's standards for coding practice.

You can use the ' cvswrappers ' file to record **cvs** wrapper commands to be used when checking files into and out of the repository. Wrappers allow the file or directory to be processed on the way in and out of CVS. The intended uses are many, one possible use would be to reformat a C file before the file is checked in, so all of the code in the repository looks the same.

You can use the ' loginfo ' file to define programs to execute after any **commit**, which writes a log entry for changes in the repository. These logging programs might be used to append the log message to a file. Or send the log message through electronic mail to a group of developers. Or, perhaps, post the log message to a particular newsgroup.

You can use the ' taginfo ' file to define programs to execute after any **tag**or**rtag** operation. These programs might be used to append a message to a file listing the new tag name and the programmer who created it, or send mail to a group of developers, or, perhaps, post a message to a particular newsgroup.

You can use the ' rcsinfo ' file to define forms for log messages.

You can use the ' editinfo ' file to define a program to execute for editing/validating ' **cvs commit** ' log en-

tries. This is most useful when used with a ‘ rcsinfo ’ forms specification, as it can verify that the proper fields of the form have been filled in by the user committing the change.

You can use the ‘ cvsignore ’ file to specify the default list of files to ignore during **update**.

You can use the ‘ history ’ file to record the **cvs** commands that affect the repository. The creation of this file enables history logging.

FILES

modules

The ‘ modules ’ file records your definitions of names for collections of source code. **cvs** will use these definitions if you use **cvs** to check in a file with the right format to ‘ **$CVSROOT/CVSROOT/modules,v** ’.

The ‘ modules ’ file may contain blank lines and comments (lines beginning with ‘ **#** ’) as well as module definitions. Long lines can be continued on the next line by specifying a backslash (“\”) as the last character on the line.

A *module definition* is a single line of the ‘ modules ’ file, in either of two formats. In both cases, *mname* represents the symbolic module name, and the remainder of the line is its definition.

mname **−a** *aliases . . .*
This represents the simplest way of defining a module *mname*. The ‘ **−a** ’ flags the definition as a simple alias: **cvs** will treat any use of *mname* (as a command argument) as if the list of names *aliases* had been specified instead. *aliases* may contain either other module names or paths. When you use paths in *aliases*, ‘ **cvs checkout** ’ creates all intermediate directories in the working directory, just as if the path had been specified explicitly in the **cvs** arguments.

mname [*options*] *dir* [*files . . .*] [**&***module . . .*]

In the simplest case, this form of module definition reduces to ‘ *mname dir* ’. This defines all the files in directory *dir* as module *mname*. *dir* is a relative path (from **$CVSROOT**) to a directory of source in one of the source repositories. In this case, on **checkout**, a single directory called *mname* is created as a working directory; no intermediate directory levels are used by default, even if *dir* was a path involving several directory levels.

By explicitly specifying *files* in the module definition after *dir*, you can select particular files from directory *dir*. The sample definition for **modules** is an example of a module defined with a single file from a particular directory. Here is another example:

m4test unsupported/gnu/m4 foreach.m4 forloop.m4

With this definition, executing ‘ **cvs checkout m4test** ’ will create a single working directory ‘ m4test ’ containing the two files listed, which both come from a common directory several levels deep in the **cvs** source repository.

A module definition can refer to other modules by including ‘ **&***module* ’ in its definition. **checkout** creates a subdirectory for each such *module*, in your working directory.
New in **cvs** *1.3;* avoid this feature if sharing module definitions with older versions of **cvs**.

Finally, you can use one or more of the following *options* in module definitions:

‘ **−d** *name* ’, to name the working directory something other than the module name.
New in **cvs** *1.3;* avoid this feature if sharing module definitions with older versions of **cvs**.

‘ **−i** *prog* ’ allows you to specify a program *prog* to run whenever files in a module are committed. *prog* runs with a single argument, the full pathname of the affected directory in a source repository. The ‘ commitinfo ’, ‘ loginfo ’, and ‘ editinfo ’ files provide other ways to call a program on **commit**.

‘ **−o** *prog* ’ allows you to specify a program *prog* to run whenever files in a module are checked out. *prog* runs with a single argument, the module name.

‘ **−e** *prog* ’ allows you to specify a program *prog* to run whenever files in a module are exported. *prog* runs with a single argument, the module name.

‘ **−t** *prog* ’ allows you to specify a program *prog* to run whenever files in a module are tagged. *prog* runs with two arguments: the module name and the symbolic tag specified to **rtag**.

‘ **−u** *prog* ’ allows you to specify a program *prog* to run whenever ‘ **cvs update** ’ is executed from the top-level directory of the checked-out module. *prog* runs with a single argument, the full path to the source repository for this module.

commitinfo, **loginfo**, **rcsinfo**, **editinfo**

These files all specify programs to call at different points in the ‘ **cvs commit** ’ process. They have a common structure. Each line is a pair of fields: a regular expression, separated by whitespace from a filename or command-line template. Whenever one of the regular expression matches a directory name in the repository, the rest of the line is used. If the line begins with a **#** character, the entire line is considered a comment and is ignored. Whitespace between the fields is also ignored.

For ‘ loginfo ’, the rest of the line is a command-line template to execute. The templates can include not only a program name, but whatever list of arguments you wish. If you write ‘ **%s** ’ somewhere on the argument list, **cvs** supplies, at that point, the list of files affected by the **commit**. The first entry in the list is the relative path within the source repository where the change is being made. The remaining arguments list the files that are being modified, added, or removed by this **commit** invocation.

For ‘ taginfo ’, the rest of the line is a command-line template to execute. The arguments passed to the command are, in order, the *tagname* , *operation* (i.e. **add** for ‘tag’, **mov** for ‘tag -F’, and **del** for ‘tag -d‘), *repository* , and any remaining are pairs of **filename revision .** A non-zero exit of the filter program will cause the tag to be aborted.

For ‘ commitinfo ’, the rest of the line is a command-line template to execute. The template can include not only a program name, but whatever list of arguments you wish. The full path to the current source repository is appended to the template, followed by the file names of any files involved in the commit (added, removed, and modified files).

For ‘ rcsinfo ’, the rest of the line is the full path to a file that should be loaded into the log message template.

For ‘ editinfo ’, the rest of the line is a command-line template to execute. The template can include not only a program name, but whatever list of arguments you wish. The full path to the current log message template file is appended to the template.

You can use one of two special strings instead of a regular expression: ‘ **ALL** ’ specifies a command line template that must always be executed, and ‘ **DEFAULT** ’ specifies a command line template to use if no regular expression is a match.

The ‘ commitinfo ’ file contains commands to execute *before* any other **commit** activity, to allow you to check any conditions that must be satisfied before **commit** can proceed. The rest of the **commit** will execute only if all selected commands from this file exit with exit status **0**.

The ‘ rcsinfo ’ file allows you to specify *log templates* for the **commit** logging session; you can use this to provide a form to edit when filling out the **commit** log. The field after the regular expression, in this file, contains filenames (of files containing the logging forms) rather than command templates.

The ‘ editinfo ’ file allows you to execute a script *before the commit starts*, but after the log information is recorded. These "edit" scripts can verify information recorded in the log file. If the edit script exits with a non-zero exit status, the commit is aborted.

The ‘ loginfo ’ file contains commands to execute *at the end* of a commit. The text specified as a commit log message is piped through the command; typical uses include sending mail, filing an article in a newsgroup, or appending to a central file.

cvsignore, **.cvsignore**
: The default list of files (or **sh**(**1**) file name patterns) to ignore during ‘ **cvs update** ’. At startup time, **cvs** loads the compiled in default list of file name patterns (see **cvs**(**1**)). Then the per-repository list included in **$CVSROOT/CVSROOT/cvsignore** is loaded, if it exists. Then the per-user list is loaded from ‘ $HOME/.cvsignore ’. Finally, as **cvs** traverses through your directories, it will load any per-directory ‘ .cvsignore ’ files whenever it finds one. These per-directory files are only valid for exactly the directory that contains them, not for any sub-directories.

history
: Create this file in **$CVSROOT/CVSROOT** to enable history logging (see the description of ‘ **cvs history** ’).

SEE ALSO

cvs(**1**),

COPYING