Tim Berners-Lee, CERN

CVS-RCS

Man pages of CVS and RCS

1. CVS Management

1.1 SYNOPSIS

$CVSROOT/CVSROOT/modules,v
$CVSROOT/CVSROOT/commitinfo,v
$CVSROOT/CVSROOT/loginfo,v
$CVSROOT/CVSROOT/rcsinfo,v
$CVSROOT/CVSROOT/editinfo,v
$CVSROOT/CVSROOT/cvsignore,v
$CVSROOT/CVSROOT/history

1.2 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.

The cvsinit(1) shell script included at the top-level of the cvs distribution can be used to setup an initial $CVSROOT/CVSROOT area, if you don't have one already.

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 '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 '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 entries. 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.

1.3 FILES

1.3.1 modules

The 'modules' file records your definitions of names for collections of source code. cvs will use these definitions if you create a file with the right format in '$CVSROOT/CVSROOT/modules,v'. The mkmodules(1) command should be run whenever the modules file changes, so that the appropriate files can be generated (depending on how you have configured cvs operation).

To allow convenient editing of the 'modules' file itself, the file should include an entry like the following (where localbin represents the directory where your site installs programs like mkmodules(1)):

```
modules -i /localbin/mkmodules CVSROOT modules
```

This defines the name 'modules' as the module name for the file itself, so that you can use

example% cvs checkout modules

to get an editable copy of the file. You should define similar module entries for the other configuration files described here (except 'history'). The cvsinit(1) script will setup a smilar 'modules' file for you automatically.

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.

'-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 'commitinfo', the rest of the line is a commandline template to execute. The template can include 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 wth 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 perdirectory 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) , mkmodules(1).

1.4 Environment Variables

1.4.1 CVSROOT

Should contain the full pathname to the root of the cvs source repository (where the RCS files are kept). This information must be available to cvs for most commands to execute; if CVSROOT is not set, or if you wish to override it for one invocation, you can supply it on the command line:

```
'cvs -d cvsroot cvs_command...'
```

You may not need to set CVSROOT if your cvs binary has the right path compiled in; use 'cvs -v' to display all compiled -in paths.

1.4.2 CVSREAD

If this is set, checkout and update will try hard to make the files in your working directory read -only. When this is not set, the default behavior is to permit modification of your working files.

1.4.3 RCSBIN

Specifies the full pathname where to find RCS programs,such as co(1) and ci(1). If not set, a compiled -invalue is used; see the display from 'cvs -v'.

1.4.4 EDITOR

Specifies the program to use for recording log messages during commit. If not set, the default is /usr/ucb/vi.

1.5 CVS Files

For more detailed information on cvs supporting files, see cvs management (5).

1.5.1 Files in working directories:

CVS A directory of cvs administrative files. Do not delete.
CVS/Entries List and status of files in your working directory.
CVS/Entries.Backup A backup of 'CVS/Entries'.
CVS/Entries.StaticFlag: do not add more entries on 'cvs update'
CVS/Repository Pathname to the corresponding directory in the source repository.
CVS/Tag Contains the per -directory "sticky" tag or date information. This file is created/updated when you specify -r or -D to the checkout or update commands, and no files are specified.
CVS/Checkin.prog Name of program to run on 'cvs commit'.
CVS/Update.prog Name of program to run on 'cvs update'.

1.5.2 Files in source repositories:

$CVSROOT/CVSROOT Directory of global administrative files for repository.
CVSROOT/commitinfo,v Records programs for filtering 'cvs commit' requests.
CVSROOT/history Log file of cvs transactions.
CVSROOT/modules,v Definitions for modules in this repository.
CVSROOT/loginfo,v Records programs for piping 'cvs commit' log entries.
CVSROOT/rcsinfo,v Records pathnames to templates used during a 'cvs commit' operation.
CVSROOT/editinfo,v Records programs for editing/validating 'cvs commit' log entries.
Attic Directory for removed source files.
#cvs.lock A lock directory created by cvs when doing sensitive changes to the RCS source repository.
#cvs.tfl.pid Temporary lock file for repository.
#cvs.rfl.pid A read lock.
#cvs.wfl.pid A write lock.

2. RCS

Introduction to RCS commands

May 11, 1983

2.1 Description

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised fre- quently, for example programs, documentation, graphics, papers, form letters, etc.

The basic user interface is extremely simple. The novice only needs to learn two commands: ci(1L) and co(1L). Ci, short for "check in", depo- sits the contents of a text file into an archival file called an RCS file. An RCS file contains all revisions of a particular text file. Co, short for "check out", retrieves revisions from an RCS file.

2.2 Functions of RCS

+ Storage and retrieval of multiple revisions of text. RCS saves all old revisions in a space efficient way. Changes no longer destroy the original, because the previous revisions remain accessible. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.

- Maintenance of a complete history of changes. RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of check-in, and a log message summariz- ing the change. The logging makes it easy to find out what hap- pened to a module, without having to compare source listings or having to track down colleagues.
- Resolution of access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.
- Maintenance of a tree of Revisions. RCS can maintain separate lines of development for each module. It stores a tree structure that represents the ancestral relationships among revisions.
- Merging of revisions and resolution of conflicts. Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged affect the same sections of code, RCS alerts the user about the overlapping changes.
- Release and configuration control. Revisions can be assigned sym- bolic names and marked as released, stable, experimental, etc. With these facilities, configurations of modules can be described simply and directly.
- Automatic identification of each revision with name, revision number, creation time, author, etc. The identification is like a stamp that can be embedded at an appropriate place in the text of a revision. The identification makes it simple to determine which revisions of which modules make up a given configuration.
- Minimization of secondary storage. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

2.3 Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the check-in command

ci f.c

This command creates the RCS file f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later check-in commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ('v' stands for 'versions'), the others are called working files. To get back the working file f.c in the previous example, use the check-out command

co f.c

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it back in by invoking

ci f.c

Ci increments the revision number properly. If ci complains with the message

ci error: no lock set by

then your system administrator has decided to create all RCS files with the locking attribute set to 'strict'. In this case, you should have locked the revision during the previous check-out. Your last check-out should have been

co -l f.c

Of course, it is too late now to do the check-out with locking, because you probably modified f.c already, and a second check-out would overwrite your modifications. Instead, invoke

rcs -l f.c

This command will lock the latest revision for you, unless somebody else got ahead of you already. In this case, you'll have to negotiate with that person.

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Even if a revision is locked, it can still be checked out for reading, compiling, etc. All that locking prevents is a CHECK-IN by anybody but the locker.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner of the RCS file need not have a lock for check-in; all others still do. Turning strict locking off and on is done with the commands

rcs -U f.c and rcs -L f.c

If you don't want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any modification. (Actually, pairs of RCS and working files can be specified in 3 ways: (a) both are given, (b) only the working file is given, (c) only the RCS file is given. Both RCS and working files may have arbitrary path prefixes; RCS commands pair them up intelligently).

To avoid the deletion of the working file during check-in (in case you want to continue editing), invoke

ci -l f.c or ci -u f.c

These commands check in f.c as usual, but perform an implicit check-out. The first form also locks the checked in revision, the second one doesn't. Thus, these options save you one check-out operation. The first form is useful if locking is strict, the second one if not strict. Both update the identification markers in your working file (see below).

You can give ci the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

ci -r2 f.c or ci -r2.1 f.c

assigns the number 2.1 to the new revision. From then on, ci will number the subsequent revisions with 2.2, 2.3, etc. The corresponding co commands

co -r2 f.c and co -r2.1 f.c

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. Co without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

ci -r1.3.1 f.c

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see rcsfile(5L).

2.3.1 Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

$Header$

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

$Header: filename revision_number date time author state $

With such a marker on the first page of each module, you can always see with which revision you are working. RCS keeps the markers up to date automatically. To propagate the markers into your object code, simply put them into literal character strings. In C, this is done as follows:

static char rcsid[]= "$Header$";

The command ident extracts such markers from any file, even object code and dumps. Thus, ident lets you find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker Log into your text, inside a comment. This marker accumulates the log messages that are requested during check-in. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see co(1L) for details. RCS(1L)

rcs - change RCS file attributes

co - check out RCS revisions

ci - check in RCS revisions

ident - identify files

rcsdiff - compare RCS revisions

rcsmerge - merge RCS revisions

rlog - print log messages and other information about RCS files

rcsfile - format of RCS file

2.4 rcs - change RCS file attributes

Synopsis

rcs [options]file ...

Description

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For rcs to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the -i option is present.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, rcs tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in co(1L).

-i creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, rcs tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

-alogins appends the login names appearing in the comma-separated list logins to the access list of the RCS file.

-Aoldfile appends the access list of oldfile to the access list of the RCS file.

-e[logins] erases the login names appearing in the comma-separated list logins from the access list of the RCS file. If logins is omitted, the entire access list is erased.

-b[rev] sets the default branch to rev. If rev is omitted, the default branch is reset to the (dynamically) highest branch on the trunk.

-cstring sets the comment leader to string. The comment leader is printed before every log message line generated by the keyword Log during checkout (see co(1L)). This is useful for programming languages without multi-line comments. During rcs -i or initial ci, the comment leader is guessed from the suffix of the working file.

-l[rev] locks the revision with number rev. If a branch is given, the latest revision on that branch is locked. If rev is omitted, the latest revision on the default branch is locked. Locking prevents overlapping changes. A lock is removed with ci or rcs -u (see below).

-u[rev] unlocks the revision with number rev. If a branch is given, the latest revision on that branch is unlocked. If rev is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

-L sets locking to strict. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

-U sets locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should NOT be used for files that are shared. The default (-L or -U) is determined by your system administrator.

-nname[:rev] associates the symbolic name name with the branch or revision rev. Rcs prints an error message if name is already associated with another number. If rev is omitted, the symbolic name is deleted.

-Nname[:rev]same as -n, except that it overrides a previous assignment of name.

-orange deletes ("outdates") the revisions given by range. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form rev1-rev2 means revisions rev1 to rev2 on the same branch, -rev means from the beginning of the branch containing rev up to and including rev, and rev means from revision rev to the end of the branch containing rev. None of the outdated revisions may have branches or locks.

-q quiet mode; diagnostics are not printed.

-sstate[:rev] sets the state attribute of the revision rev to state. If rev is a branch number, the latest revision on that branch is assumed. If rev is omitted, the latest revision on the default branch is assumed. Any identifier is acceptable for state. A useful set of states is Exp (for experimental), Stab (for stable), and Rel (for released). By default, ci(1L) sets the state of a revision to Exp.

-t[txtfile] writes descriptive text into the RCS file (deletes the existing text). If txtfile is omitted, rcs prompts the user for text supplied from the standard input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file txtfile. If the -i option is present, descriptive text is requested even if -t is not given. The prompt is suppressed if the standard input is not a terminal.

Diagnostics

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

FILES The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. Rcs creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, rcs always creates a new file. On successful completion, rcs deletes the old one and renames the new one. This strategy makes links to RCS files useless. CO(1L)

2.5 co - check out RCS revisions

Synopsis

co [options]file ...

Description

Co retrieves a revision from each RCS file and stores it into the corresponding working file. Each file name ending in ',v' is taken to be an RCS file; all other files are assumed to be working files. If only a working file is given, co tries to find the corresponding RCS file in the directory ./RCS and then in the current directory. For more details, see the file naming section below.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Co with locking fails if the revision to be checked out is currently locked by another user. (A lock may be broken with the rcs(1L) command.) Co with locking also requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. Co without locking is not subject to accesslist restrictions, and is not affected by the presence of locks.

A revision is selected by options for revision or branch number, checkin date/time, author, or state. When the selection options are applied in combination, co retrieves the latest revision that satisfies all of them. If none of the selection options is specified, co retrieves the latest revision on the default branch (normally the trunk, see the -b option of rcs(1L)). A revision or branch number may be attached to any of the options -f, -l, -p, -q, -r, or -u. The options -d (date), -s (state), and -w (author) retrieve from a single branch, the selected branch, which is either specified by one of -f,..., -u, or the default branch.

A co command applied to an RCS file with no revisions creates a zero-length working file. Co always performs keyword substitution (see below).

-r[rev] retrieves the latest revision whose number is less than or equal to rev. If rev indicates a branch rather than a revision, the latest revision on that branch is retrieved. If rev is omitted, the latest revision on the default branch (see the -b option of rcs(1L)) is retrieved. Rev is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the -n option of the commands ci(1L) and rcs(1L).

-l[rev] same as -r, except that it also locks the retrieved revision for the caller. See option -r for handling of the revision number rev .

-u[rev] same as -r, except that it unlocks the retrieved revision (if it was locked by the caller). If rev is omitted, -u retrieves the latest revision locked by the caller; if no such lock exists, it retrieves the latest revision on the default branch.

-f[rev] forces the overwriting of the working file; useful in connection with -q. See also the section on file modes below.

-p[rev] prints the retrieved revision on the standard output rather than storing it in the working file. This option is useful when co is part of a pipe.

-q[rev] quiet mode; diagnostics are not printed.

-ddate retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to date. The date and time may be given in free format and are converted to local time. Examples of formats for date:
22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21 (free format),
Fri, April 16 15:52:25 EST 1982 (output of ctime).
Most fields in the date and time may be defaulted. Co determines the defaults in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-sstate retrieves the latest revision on the selected branch whose state is set to state.

-w[login] retrieves the latest revision on the selected branch which was checked in by the user with login name login. If the argument login is omitted, the caller's login is assumed.

-jjoinlist generates a new revision which is the join of the revisions on joinlist. Joinlist is a comma-separated list of pairs of the form rev2:rev3, where rev2 and rev3 are (symbolic or numeric) revision numbers. For the initial such pair, rev1 denotes the revision selected by the above options -r, ..., -w. For all other pairs, rev1 denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, co joins revisions rev1 and rev3 with respect to rev2. This means that all changes that transform rev2 into rev1 are applied to a copy of rev3. This is particularly useful if rev1 and rev3 are the ends of two branches that have rev2 as a common ancestor. If rev1 < rev2 < rev3 on the same branch, joining generates a new revision which is like rev3, but with all changes that lead from rev1 to rev2 undone. If changes from rev2 to rev1 overlap with changes from rev2 to rev3, co prints a warning and includes the overlapping sections, delimited by the lines <<<<<<< rev1, =======, and >>>>>>> rev3.

For the initial pair, rev2 may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. The options -l and -u lock or unlock rev1.

2.5.1 Keyword Substitution

Strings of the form $keyword$ and $keyword:...$ embedded in the text are replaced with strings of the form $keyword: value $, where keyword and value are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form $keyword$. On checkout, co replaces these strings with strings of the form $keyword: value $. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values:

$Author$ The login name of the user who checked in the revision.

$Date$ The date and time the revision was checked in.

$Header$ A standard header containing the full pathname of the RCS file, the revision number, the date, the author, the state, and the locker (if locked).

Id Same as $Header$, except that the RCS file name is without a path.

$Locker$ The login name of the user who locked the revision (empty if not locked).

Log The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after $Log:...$. This is useful for accumulating a complete change log in a source file.

$RCSfile$ The name of the RCS file without path.

$Revision$ The revision number assigned to the revision.

$Source$ The full pathname of the RCS file.

$State$ The state assigned to the revision with the -s option of rcs(1L) or ci(1L).

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file name is of the form path1/workfile,v and the working file name is of the form path2/workfile, where path1/ and path2/ are (possibly different or empty) paths and workfile is a file name.

2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing path1/ and the suffix ,v.

3) Only the working file is given. Then co looks for an RCS file of the form path2/RCS/workfile,v or path2/workfile,v (in this order).

If the RCS file is specified without a path in 1) and 2), then co looks for the RCS file first in the directory ./RCS and then in the current directory.

Examples

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

co io.c; co RCS/io.c,v; co io.c,v;

co io.c RCS/io.c,v; co io.c io.c,v;

co RCS/io.c,v io.c; co io.c,v io.c;

2.5.2 File Modes

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to strict (see rcs(1L)).

If a file with the name of the working file exists already and has write permission, co aborts the checkout if -q is given, or asks whether to abort if -q is not given. If the existing working file is not writable or -f is given, the working file is deleted without asking.

2.5.3 Files

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

Diagnostics

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

Limitations

The option -d gets confused in some circumstances, and accepts no date before 1970. Links to the RCS and working files are not preserved. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character '\&' into the keyword.

Bugs

The option -j does not work for files that contain lines with a single

2.6 ci - check in RCS revisions

Synopsis

ci [options]file ...

Description

Ci stores new revisions into RCS files. Each file name ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. Ci deposits the contents of each working file into the corresponding RCS file. If only a working file is given, ci tries to find the corresponding RCS file in the directory ./RCS and then in the current directory. For more details, see the file naming section below.

For ci to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to strict (see rcs(1L)). A lock held by someone else may be broken with the rcs command.

Normally, ci checks whether the revision to be deposited is different from the preceding one. If it is not different, ci either aborts the deposit (if -q is given) or asks whether to abort (if -q is omitted). A deposit can be forced with the -f option.

For each revision deposited, ci prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. If several files are checked in, ci asks whether to reuse the previous log message. If the standard input is not a terminal, ci suppresses the prompt and uses the same log message for all files. See also -m.

The number of the deposited revision can be given by any of the options

-r, -f, -k, -l, -u, or -q.

If the RCS file does not exist, ci creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, ci requests descriptive text (see -t below).

-r[rev] assigns the revision number rev to the checked-in revision, releases the corresponding lock, and deletes the working file. This is the default. Rev may be symbolic, numeric, or mixed.

If rev is a revision number, it must be higher than the latest one on the branch to which rev belongs, or must start a new branch.

If rev is a branch rather than a revision number, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If rev indicates a non-existing branch, that branch is created with the initial revision numbered rev.1.

If rev is omitted, ci tries to derive the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1.

If rev is omitted and the caller has no lock, but he is the owner of the file and locking is not set to strict, then the revision is appended to the default branch (normally the trunk; see the -b option of rcs(1L)).

Exception: On the trunk, revisions can be appended to the end, but not inserted.

-f[rev] forces a deposit; the new revision is deposited even it is not different from the preceding one.

-k[rev] searches the working file for keyword values to determine its revision number, creation date, state, and author (see co(1)), and assigns these values to the deposited revision, rather than computing them locally. It also generates a default login message noting the login of the caller and the actual checkin date. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the -k option at these sites to preserve the original number, date, author, and state. The extracted keyword values and the default log message may be overridden with the options -r, -d, -s, -w, and -m.

-l[rev] works like -r, except it performs an additional co -l for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

-u[rev] works like -l, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

-q[rev] quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless -f is given.

-ddate uses date for the checkin date and time. Date may be specified in free format as explained in co(1). Useful for lying about the checkin date, and for -k if no date is available.

-mmsg uses the string msg as the log message for all revisions checked in.

-nname assigns the symbolic name name to the number of the checked-in revision. Ci prints an error message if name is already assigned to another number.

-Nname same as -n, except that it overrides a previous assignment of name.

-sstate sets the state of the checked-in revision to the identifier state. The default is Exp.

-t[txtfile] writes descriptive text into the RCS file (deletes the existing text). If txtfile is omitted, ci prompts the user for text supplied from the standard input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file txtfile. During initialization, descriptive text is requested even if -t is not given. The prompt is suppressed if standard input is not a terminal.

-wlogin uses login for the author field of the deposited revision. Useful for lying about the author, and for -k if no author is available.

2.6.1 File Naming

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of co(1)).

1) Both the RCS file and the working file are given. The RCS file name is of the form path1/workfile,v and the working file name is of the form path2/workfile, where path1/ and path2/ are (possibly different or empty) paths and workfile is a file name.

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing path1/ and the suffix ,v.

3) Only the working file is given. Then ci looks for an RCS file of the form path2/RCS/workfile,v or path2/workfile,v (in this order).

If the RCS file is specified without a path in 1) and 2), then ci looks for the RCS file first in the directory ./RCS and then in the current directory.

File Modes

An RCS file created by ci inherits the read and execute permissions from the working file. If the RCS file exists already, ci preserves its read and execute permissions. Ci always turns off all write permissions of RCS files.

Files

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. Ci always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

Diagnostics

For each revision, ci prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the

2.7 ident - identify files

Synopsis

ident [-q][file ...]

Description

Ident searches the named files or, if no file name appears, the standard input for all occurrences of the pattern $keyword:...$, where keyword is one of

- Author
- Date
- Header

- Id
- Locker
- Log
- Revision
- RCSfile
- Source
- State

These patterns are normally inserted automatically by the RCS command co(1L), but can also be inserted manually. The option -q suppresses the warning given if there are no patterns in a file.

Ident works on text files as well as object files and dumps. For example, if the C program in file f.c contains

char rcsid[]= "$Header: Header information $";

and f.c is compiled into f.o, then the command

ident f.c f.o

will print

f.c:

$Header: Header information $

f.o:

$Header: Header information $

Identification

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907. Revision Number: 1.2 ; Release Date: 89/05/02 . Copyright c 1982, 1988, 1989 by Walter F. Tichy. RCSDIFF(1L)

2.8 rcsdiff - compare RCS revisions

Synopsis

rcsdiff [-biwt][-cefhn][-q][-rrev1][-rrev2]file ...

Description

Rcsdiff runs diff(1) to compare two revisions of each RCS file given. A file name ending in ',v' is an RCS file name, otherwise a working file name. Rcsdiff derives the working file name from the RCS file name and vice versa, as explained in co(1L). Pairs consisting of both an RCS and a working file name may also be specified.

The options -b, -i, -w, -t, -c, -e, -f, and -h, have the same effect as described in diff(1); option -n generates an edit script of the format used by RCS. The option -q suppresses diagnostic output.

If both rev1 and rev2 are omitted, rcsdiff compares the latest revision on the default branch (normally the highest branch on the trunk) with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If rev1 is given, but rev2 is omitted, rcsdiff compares revision rev1 of the RCS file with the contents of the corresponding working file.

If both rev1 and rev2 are given, rcsdiff compares revisions rev1 and rev2 of the RCS file.

Both rev1 and rev2 may be given numerically or symbolically, and may actually be attached to any of the options.

Examples

The command rcsdiff f.c

runs diff on the latest revision on the default branch of RCS file f.c,v and the working file f.c.

Diagnostics

The exit status is 0 if there were no differences during the last comparison, 1 if there were differences, and 2 if there were errors.

Identification

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907. Revision Number: 1.3 ; Release Date: 89/05/02 . Copyright c 1982, 1988, 1989 by Walter F. Tichy.

2.9 rcsmerge - merge RCS revisions

Synopsis

rcsmerge -rrev1 [-rrev2][-p]file

Description

Rcsmerge incorporates the changes between rev1 and rev2 of an RCS file into the corresponding working file. If -p is given, the result is printed on the standard output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. Merge derives the working file name from the RCS file name and vice versa, as explained in co(1L). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If rev2 is omitted, the latest revision on the default branch (normally the highest branch on the trunk) is assumed. Both rev1 and rev2 may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in co -j. The command is useful for incorporating changes into a checked-out revision.

Examples

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when you receive updates to release 2.8 from someone else. To combine the updates to 2.8 and your changes between 2.8 and 3.4, put the updates to 2.8 into file f.c and execute

rcsmerge -p -r2.8 -r3.4 f.c >f.merged.c

Then examine f.merged.c. Alternatively, if you want to save the updates to 2.8 in the RCS file, check them in as revision 2.8.1.1 and execute co -j:

ci -r2.8.1.1 f.c

co -r3.4 -j2.8:2.8.1.1 f.c

As another example, the following command undoes the changes between revision 2.4 and 2.8 in your currently checked out revision in f.c.

rcsmerge -r2.8 -r2.4 f.c

Note the order of the arguments, and that f.c will be overwritten.

Bugs

Rcsmerge does not work on files that contain lines with a single '.'.

rlog - print log messages and other information about RCS files

2.10 rlog - print log messages and other information about RCS files

Synopsis

rlog [options]file ...

Description

Rlog prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, rlog tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in co(1L).

Rlog prints the following information for each RCS file: RCS file name, working file name, head (i.e., the number of the latest revision on the trunk), default branch, access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for the selected revisions in reverse chronological order for each branch. For each revision, rlog prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. Without options, rlog prints complete information. The options below restrict this output.

-L ignores files with no locks set; convenient in combination with -R,-h, or -l.

-R only prints the name of the RCS file; convenient for translating a working file name into an RCS file name.

-h prints only RCS file name, working file name, head, default branch, access list, locks, symbolic names, and suffix.

-t prints the same as -h, plus the descriptive text.

-b prints information about the revisions on the default branch (normally the highest branch on the trunk).

-ddates prints information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of dates. A range d1<d2 or d2>d1 selects the revisions that were deposited between d1 and d2, (inclusive). A range <d or d> selects all revisions dated d or earlier. A range d< or >d selects all revisions dated d or later. A range d selects the single, latest revision dated d or earlier. The date/time strings d, d1, and d2 are in the free format explained in co(1L). Quoting is normally necessary, especially for < and >. The separator is ';'.

-l[lockers] prints information about locked revisions. If the comma-separated list lockers of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

-rrevisions prints information about revisions given in the comma-separated list revisions of revisions and ranges. A range rev1-rev2 means revisions rev1 to rev2 on the same branch, -rev means revisions from the beginning of the branch up to and including rev, and revmeans revisions starting with rev to the end of the branch containing rev. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

-sstates prints information about revisions whose state attributes match one of the states given in the comma-separated list states.

-w[logins] prints information about revisions checked in by users with login names appearing in the comma-separated list logins. If logins is omitted, the user's login is assumed.

Rlog prints the intersection of the revisions selected with the options -d, -l, -s, -w, intersected with the union of the revisions selected by -b and -r.

Examples

rlog -L -R RCS/*,v

rlog -L -h RCS/*,v

rlog -L -l RCS/*,v

rlog RCS/*,v

The first command prints the names of all RCS files in the subdirectory 'RCS' which have locks. The second command prints the headers of those files, and the third prints the headers plus the log messages of the locked revisions. The last command prints complete information.

Diagnostics

The exit status always refers to the last RCS file operated upon,

2.11 rcsfile - format of RCS file

Description

An RCS file is an ASCII file. Its contents are described by the grammar below. The text is free format, i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by ’@’. If a string contains a ’@’, it must be doubled.

The meta syntax uses the following conventions: ’|’ (bar) separates alternatives; ’{’ and ’}’ enclose optional phrases; ’{’ and ’}*’ enclose phrases that may be repeated zero or more times; ’{’ and ’}+’ enclose phrases that must appear at least once and may be repeated; ’ ’ enclose nonterminals.

```
<rcstext>      ::=  <admin> {<delta>}* <desc> {<deltatext>}*

<admin>        ::=  head      {<num>};
                    branch    {<num>};
                    access    {<id>}*;
                    symbols   {<id> : <num>}*;
                    locks     {<id> : <num>}*;
                    comment   {<string>};

<delta>        ::=  <num>
                    date      <num>;
                    author    <id>;
                    state     {<id>};
                    branches  {<num>}*;
                    next      {<num>};

<desc>         ::=  desc      <string>

<deltatext>    ::=  <num>
                    log       <string>
                    text      <string>

<num>          ::=  {<digit>{.}}+

<digit>        ::=  0 | 1 | ... | 9

<id>           ::=  <letter>{<idchar>}*

<letter>       ::=  A | B | ... | Z | a | b | ... | z

<idchar>       ::=  Any printing ASCII character except space,
                    tab, carriage return, new line, and <special>.

<special>      ::=  ; | : | , | @

<string>       ::=  @{any ASCII character, with ‘@’ doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

The <delta> nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the "trunk", and are linked through the next field in order of decreasing numbers. The head field in the <admin> node points to the head of that sequence (i.e., contains the highest pair). The branch node in the admin node indicates the default branch (or revision) for most RCS operations. If empty, the default branch is the highest branch on the trunk.

All <delta> nodes whose numbers consist of 2n fields (n>2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first (2n)-1 number fields are identical are linked through the next field in order of increasing numbers. For each such sequence, the <delta> node whose number is identical to the first 2(n-1) number fields of the deltas on that sequence is called the branchpoint. The branches field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

Example:

```
                           Head
                             |
                             |
                             v
                         ---------
   / \          / \      |       |      / \          / \
  /   \        /   \     |  2.1  |     /   \        /   \
 /     \      /     \    |       |    /     \      /       \
/1.2.1.3\    /1.3.1.1\   |       |   /1.2.2.2\   /1.2.2.1.1.1\
---------    ---------   ---------   ---------   -------------
    ^            ^           |           ^             ^
    |            |           |           |             |
    |            |           v           |             |
   / \           |       ---------      / \            |
  /   \          |       \  1.3  /     /   \           |
 /     \         ---------\     /     /     \-----------
/1.2.1.1\                  \   /     /1.2.2.1\
---------                   \ /      ---------
    ^                        |           ^
    |                        |           |
    |                        v           |
    |                    ---------       |
    |                    \  1.2  /       |
    ----------------------\     /---------
                           \   /
                            \ /
                             |
                             |
                             v
                         ---------
                         \  1.1  /
                          \     /
                           \   /
                            \ /
```

Fig. 1: A revision tree

3. CVS −− Concurrent Versions System

See also the gnu Info on CVS.

3.1 SYNOPSIS

cvs [cvs_options]

cvs_command [command_options][command_args]

3.2 Contents:

- Introduction
- Commands
- Files
- Environment variables
- Management
- RCS

3.3 Authors

Dick Grune −− Original author of the cvs shell script version posted to comp.sources.unix in the volume6 release of December, 1986. Credited with much of the cvs conflict resolution algorithms.

Brian Berliner −− Coder and designer of the cvs program itself in April,1989, based on the original work done by Dick.

Jeff Polk −− Helped Brian with the design of the cvs module and vendor branch support and author of the checkin(1) shellscript (the ancestor of 'cvs import').

3.4 SEE ALSO

ci(1), co(1), cvs(5), diff(1), grep(1), mkmodules(1),patch(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1), rm(1),sort(1).

3.5 COPYING

3.6 Introduction to CVS

cvs is a front end to the rcs (1) revision control system which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories consisting of revision controlled files. These directories and files can be combined together to form a software release. cvs provides the functions necessary to manage these software releases and to control the concurrent editing of source files among multiple software developers.

cvs keeps a single copy of the master sources. This copy is called the source "repository"; it contains all the information to permit extracting previous software releases at any time based on either a symbolic revision tag, or a date in the past.

3.6.1 ESSENTIAL COMMANDS

cvs provides a rich variety of commands (cvs_command in the Synopsis), each of which often has a wealth of options, to satisfy the many needs of source management in distributed environments. However, you don't have to master every detail to do useful work with cvs; in fact, five commands are sufficient to use (and contribute to) the source repository.

3.6.2 cvs checkout modules...

A necessary preliminary for most cvs work: creates your private copy of the source for modules (named collections of source; you can also use a path relative to the source repository here). You can work with this copy without interfering with others' work. At least one subdirectory level is always created.

cvs update

Execute this command from within your private source directory when you wish to update your copies of source files from changes that other developers have made to the source in the repository.

cvs add file...

Use this command to enroll new files in cvs records of your working directory. The files will be added to the repository the next time you run 'cvs commit'. Note: You should use the 'cvs import' command to bootstrap new sources into the source repository. 'cvs add' is only used for new files to an already checked -out module.

cvs remove file...

Use this command (after erasing any files listed) to declare that you wish to eliminate files from the repository. The removal does not affect others until you run 'cvs commit'.

cvs commit file...

Use this command when you wish to "publish" your changes to other developers, by incorporating them in the source repository.

3.6.3 OPTIONS

The cvs command line can include cvs_options, which apply to the overall cvs program; a cvs_command, which specifies aparticular action on the source repository; and command_options and command_arguments to fully specify what the cvs_command will do.

Warning: you must be careful of precisely where you place options relative to the cvs_command. The same option can mean different things depending on whether it is in the cvs_options position (to the left of a cvs command) or in the command_options position (to the right of a cvs command).

There are only two situations where you may omit cvs_command: 'cvs -H' elicits a list of available commands and 'cvs -v ' displays version information on cvs itself.

3.6.4 CVS OPTIONS

Use these options for overall control of the cvs program:

H Display usage information about the specified cvs_command (but do not actually execute the command). If you don't specify a command name, 'cvs -H' displays a summary of all the commands available.

Q Causes the command to be really quiet; the command will generate output only for serious problems.

q Causes the command to be somewhat quiet; informational messages, such as reports of recursion through subdirectories, are suppressed.

b bindir Use bindir as the directory where RCS programs are located. Overrides the setting of the RCSBIN environment variable. This value should be specified as an absolute pathname.

d CVS_root_directory Use CVS_root_directory as the root directory pathname of the master RCS source repository. Overrides the setting of the CVSROOT environment variable. This value should be specified as an absolute pathname.

e editor Use editor to enter revision log information. Overrides the setting of the EDITOR environment variable.

l Do not log the cvs_command in the command history (but execute it anyway). See the description of the history command for information on command history.

n Do not change any files. Attempt to execute the cvs_command, but only to issue reports; do not remove, update, or merge any existing files, or create any new files.

t Trace program execution; display messages showing the steps of cvs activity. Particularly useful with -n to explore the potential impact of an unfamiliar command.

r Makes new working files files read -only. Same effect as if the CVSREAD environment variable is set.

v Displays version and copyright information for cvs.

w Makes new working files read -write (default). Overrides the setting of the CVSREAD environment variable.

3.6.5 USAGE

Except when requesting general help with 'cvs -H', you mustspecify a cvs_command to cvs to performa specific release control function. Each cvs command accepts its own collection of options and arguments. However, many options are available across several commands. You can display a usage summary for each command by specifying the -H option with the command.

3.6.6 CVS COMMAND SUMMARY

Here are brief descriptions of all the cvs commands :

add Add a new file or directory to the repository, pending a 'cvs commit' on the same file. Can only be done from within sources created by a previous 'cvs checkout' invocation. Use 'cvs import' to place whole new hierarchies of sources under cvs control. (Does not directly affect repository; changes working directory.)

admin Execute RCS control functions on the source repository. (Changes repository directly; uses working directory without changing it.)

checkout Make a working directory of source files for editing. (Creates or changes working directory.)

commit Apply to the source repository changes, additions, and deletions from your working directory. (Changes repository.)

diff Show differences between files in working directory and source repository, or between two revisions in source repository. (Does not change either repository or working directory.)

export Prepare copies of a set of source files for shipment off-site. Differs from 'cvs checkout' in that no cvs administrative directories are created (and therefore 'cvs commit' cannot be executed from a directory prepared with 'cvsíexport'), and a symbolic tag must be specified. (Does not change repository; creates directory similar to working directories).

history Show reports on cvs commands that you or others have executed on a particular file or directory in the source repository. (Does not change repository or working directory.) History logs are kept only if enabled by creation of the '$CVSROOT/CVSROOT/history' file; see cvs(5).

import Incorporate a set of updates from off -site into the source repository, as a "vendor branch". (Changes repository.)

log Display RCS log information. (Does not change repository or working directory.)

rdiff Prepare a collection of diffs as a patch file between two releases in the repository. (Does not change repository or working directory.)

release Cancel a 'cvsícheckout' abandoning any changes. (Can delete working directory; no effect on repository.)

remove Remove files from the source repository, pending a 'cvs-commit' on the same files. (Does not directly affect repository; changes working directory.)

rtag Explicitly specify a symbolic tag for particular revisions of files in the source repository. See also 'cvs tag'. (Changes repository directly; does not require or affect working directory.)

status Show current status of files: latest version, versioníin working directory, whether working version has been edited and, optionally, symbolic tags in the RCS file. (Does not change repository or working directory.)

tag Specify a symbolic tag for files in the repository. Tags the revisions that were last synchronized with your working directory. (Changes repository directly; uses working directory without changing it.)

update Bring your working directory up to date with changesfrom the repository. Merges are performed automatically when possible; a warning is issued if manualresolution is required for conflicting changes.(Changes working directory; does not change repository.)

3.6.7 COMMON COMMAND OPTIONS

This section describes the command_options that are available across several cvs commands. Not all commands support all of these options; each option is only supported for commands where it makes sense. However, when a command has one of these options you can count on the same meaning for the option as in other commands. (Other command options, which are listed with the individual commands, may have different meanings from one cvs command to another.) Warning: the history command is an exception; it supports many options that conflict even with these standard options.

D date_spec Use the most recent revision no later than date_spec (a single argument, date description specifying a date in the past). A wide variety of date formats are supported by the underlying RCS facilities, similar to those described in co(1), but not exactly the same. The date_spec is interpreted as being in the local time zone, unless a specific time zone is specified. The specification is "sticky" when you use it to make a private copy of a source file; that is, when you get a working file using -D, cvs records the date you specified, so that further updates in the same directory will use the same date (unless you explicitly override it; see the description of the update command). -D is available with the checkout, diff, history, rdiff, rtag, and update commands. Examples of valid date specifications include:

```
1 month ago
2 hours ago
```

```
400000 seconds ago
last year
last Monday
yesterday
a fortnight ago
3/31/92 10:00:07 PST
January 23, 1987 10:05pm
22:00 GMT
```

f When you specify a particular date or tag to cvs commands, they normally ignore files that do not contain the tag (or did not exist on the date) that you specified. Use the -f option if you want files retrieved even when there is no match for the tag or date. (The most recent version is used in this situation.) -f is available with these commands: checkout, export, rdiff, rtag, and update.

-H Help; describe the options available for this command. This is the only option supported for all cvs commands.

-k kflag Alter the default RCS processing of keywords; all the -k options described in rcs(1) are available. The -k option is available with the add, checkout, diff, rdiff, and update commands. Your kflag specification is "sticky" when you use it to create a private copy of a source file; that is, when you use this option with the checkout or update commands, cvs associatesyour selected kflag with the file, and continues to use it with future update commands on the same file until you specify otherwise.

-l Local; run only in current working directory, rather than recurring through subdirectories. Available with the following commands: checkout, commit, diff, export,remove, rdiff, status, tag, and update. Warning: this is not the same as the overall 'cvs -l' option, which you can specify to the left of a cvs command!

-n Do not run any checkout/commit/tag/ program. (A program can be specified to run on each of these activities, in the modules database; this option bypasses it.) Available with the checkout, commit, export, and rtag commands. Warning: this is not the same as theoverall 'cvs -n' option, which you can specify to the left of a cvs command!

-P Prune (remove) directories that are empty after being updated, on checkout, or update. Normally, an empty directory (one that is void of revision -controlled files) is left alone. Specifying -P will cause these directories to be silently removed from your checked -out sources. This does not remove the directory from the repository, only from your checked out copy. Note that this option is implied by the -r or -D options of checkout and export.

-p Pipe the files retrieved from the repository to standard output, rather than writing them in the current directory. Available with the checkout and update commands.

-Q Causes the command to be really quiet; the command will generate output only for serious problems. Available with the following commands: checkout, import, export, rdiff, rtag, tag, and update.

-q Causes the command to be somewhat quiet; informational messages, such as reports of recursion through subdirectories, are suppressed. Available with the following commands: checkout, import, export, rtag, tag and update.

-r tag Use the revision specified by the tag argument instead of the default "head" revision. As well as arbitrary tags defined with the tag or rtag command, two special tags are always available: 'HEAD' refers to the most recent version available in the repository, and 'BASE' refers to the revision you last checked out into thecurrent working directory.

The tag specification is "sticky" when you use this option with 'cvs checkout' or 'cvs update' to make your own copy of a file: cvs remembers the tag and continues to use it on future update commands, until you specify otherwise. tag can be either a symbolic or numeric tag, in RCS fashion. Specifying the -q option alongwith the -r option is often useful, to suppress the warning messages when the RCS file does not contain the specified tag. -r is available with the checkout, commit, diff, history, export, rdiff, rtag, and update commands. Warning: this is not the same as the overall 'cvs -r' option, which you can specify to the left of a cvs command!

4. CVS Commands

Here are details on all the cvs commands and the options each accepts. The summary lines at the top of each command's description highlight three kinds of things:

Command Options and Argument Special options are described in detail below; common command options may appear only in the summary line.

Working Directory, or Repository? Some cvs commands require a working directory to operate; some require a repository. Also, some commands change the repository, some change the working directory, and some change nothing.

Synonyms Many commands have synonyms, which you may find easier to remember (or type) than the principal name.

See also: common options

4.1 Commands

add	[-k kflag][-m 'message']files...
admin	[rcs -options]files...
checkout	[options]modules... [files ...]
commit	[-lnR][-m 'log_message' | -f file][-r revision][files...]
diff	[-kl][rcsdiff_options][[-r rev1 | -D date1][-r rev2 | -D date2]]
export	[-flNnQq]-r rev| -D date [-d dir]module...
history	[-report][-flags][-options args][files...]
import	[-options]repository vendortag releasetag...
log	[-l]rlog -options [files...]
rdiff	[-flags][-V vn][-r t| -D d [-r t2| -D d2]]modules...
release	[-dQq]modules...
remove	[-lR][files...]modules...
rtag	[-falnRQq][-b][-d][-r tag | -D date]symbolic_tag
status	[-lR][-v][files...]
tag	[-lQqR][-b][-d]symbolic_tag [files...]
update	[-AdflPpQqR][-d][-r tag| -D date]files...

4.2 add [-k kflag][-m 'message']files...

Requires: repository, working directory
Changes: working directory
Synonym: new

Use the add command to create a new file or directory in the RCS source repository. The files or directories specified with add must already exist in the current directory (which must have been created with the checkout command). To add a whole new directory hierarchy to the source repository (for example, files received from a third -party vendor), use the ' cvs import ' command instead.
If the argument to 'cvs add' refers to an immediate subdirectory, the directory is created at the correct place in the RCS source repository, and the necessary cvs administration files are created in your working directory. If the directory already exists in the source repository, 'cvs add' still creates the administration files in your version of the directory. This allows you to use 'cvsíadd' to add a particular directory to your private sources even if someone else created that directory after your checkout of the sources. You can do the following:

```
mkdir new_director
cvs add new_directory
cvs update new_directory
```

An alternate approach using 'cvs update' might be:

```
cvs update -d new_director
```

(To add any available new directories to your working directory, it's probably simpler to use 'cvs checkout' or 'cvs update -d'.)
The added files are not placed in the RCS source repository until you use 'cvs commit' to make the change permanent. Doing a 'cvs add' on a file that was removed with the 'cvs remove' command will resurrect the file, if no 'cvs commit' command intervened.
You will have the opportunity to specify a logging message, as usual, when you use 'cvs commit' to make the new file permanent. If you'd like to have another logging message associated with just creation of the file (for example, to describe the file's purpose), you can specify it with the ' -mímessage' option to the add command.
The ' -k kflag' option specifies the default way that this file will be checked out. The 'kflag' argument isstored in the RCS file and can be changed with 'cvsadmin'. Specifying ' -ko' is useful for checking in binaries that shouldn't have the RCS id strings expanded.

4.3 admin [rcs -options]files...

Requires: repository, working directory
Changes: repository.
Synonym: rcs
This is the cvs interface to assorted administrative RCS facilities, documented in rcs(1). 'cvsíadmin' simply passes all its options and arguments to the rcs command; it does no filtering or other processing. This command does work recursively, however, so extreme care should be used.

4.4 checkout [options]modules... [files ...]

checkout	[options]modules...
Requires:	repository.
Changes:	working directory
Synonyms:	co, get

Make a working directory containing copies of the source files specified by modules. You must execute 'cvs checkout' before using most of the other cvs commands, since most of them operate on your working directory.
Modules are either symbolic names (themselves defined as the module 'modules' in the source repository; see cvs(5)) for some collection of source directories and files, or paths to directories or files in the repository.
Depending on the modules you specify, checkout may recursively create directories and populate them with the appropriate source files. You can then edit these source files at any time (regardless of whether other software developers are editing their own copies of the sources); update them to include new changes applied by others to the source repository; or commit your work as a permanent change to the RCS repository.
Note that checkout is used to create directories. The toplevel directory created is always added to the directory where checkout is invoked, and usually has the same name as the specified module. In the case of a module alias, the created subdirectory may have a different name, but you can be sure that it will be a subdirectory, and that checkout will

show the relative path leading to each file as it is extracted into your private work area (unless you specify the -Q option).
Running 'cvs checkout' on a directory that was already built by a prior checkout is also permitted, and has the same effect as specifying the -d option to the update command described below.
The options permitted with 'cvs checkout' include the standard command optionsí -P, -Q, -f, -k kflag , -l, -n, -p, -q, -r tag, andí -D date.
In addition to those, you can use these special command options with checkout:
Use the -A option to reset any sticky tags, dates, or -k options. (If you get a working file using one ofthe -r, -D, or -k options, cvs remembers thecorresponding tag, date, or kflag and continues using it on future updates; use the -A option to make cvs forget these specifications, and retrieve the "head" version of the file).
The -j branch option merges the changes made between the resulting revision and the revision that it is based on (e.g., if the tag refers to a branch, cvs will merge all changes made in that branch into your working file).
With two -j options, cvs will merge in the changes between the two respective revisions. This can be used to "remove" a certain delta from your working file.
In addition, each -j option can contain on optional date specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag. An example might be what 'cvs import' tells you to do when you have just imported sources that have conflicts with local changes:
cvs checkout -jTAG:yesterday -jTAG module
Use the -N option with ' -d dir' to avoid shortening module paths in your working directory. (Normally, cvs shortens paths as much as possible when you specify an explicit target directory.)
Use the -c option to copy the module file, sorted, to the standard output, instead of creating or modifying any files or directories in your working directory.
Use the -d dir option to create a directory called dir for the working files, instead of using the module name. Unless you also use -N, the paths created under dir will be as short as possible.
Use the -s option to display per -module status information stored with the -s option within the modules file.

4.5 commit [-lnR][-m 'log_message' | -f file][-r revision][files...]

Requires:	working directory, repository
Changes:	repository.
Synonym:	ci

Use 'cvs commit' when you want to incorporate changes from your working source files into the general source repository.
If you don't specify particular files to commit, all of the files in your working current directory are examined. commit is careful to change in the repository only those files that you have really changed. By default (or if you explicitly specify the -R option), files in subdirectories are also examined and committed if they have changed; you can use the -l option to limit commit to the current directory only.
commit verifies that the selected files are up to date with the current revisions in the source repository; it will notify you, and exit without committing, if any of the specified files must be made current first with 'cvs update'. commit does not call the update command for you, but rather leaves that for you to do when the time is right.
When all is well, an editor is invoked to allow you to enter a log message that will be written to one or more logging programs and placed in the RCS source repository file. You can instead specify the log message on the command line with the -m option, thus

suppressing the editor invocation, or use the -f option to specify that the argument file contains the log message.
The -r option can be used to commit to a particular symbolic or numeric revision within the RCS file. For example, to bring all your files up to the RCS revision "3.0" (including those that haven't changed), you might do:

```
cvs commit -r3.0
```

cvs will only allow you to commit to a revision that is on the main trunk (a revision with a single dot). However, you can also commit to a branch revision (one that has an even number of dots) with the -r option. To create a branch revision, one typically uses the -b option of the rtag or tag commands. Then, either checkout or update can be used to base your sources on the newly created branch. From that point on, all commit changes made within these working sources will be automatically added to a branch revision, thereby not perturbing mainline development in any way. For example, if you had to create a patch to the 1.2 version of the product, even though the 2.0 version is already under development, you might do:

```
cvs rtag -b -rFCS1_2 FCS1_2_Patch product_module

cvs checkout -rFCS1_2_Patch product_module

cd product_module

[[ hack away ]]

cvs commit
```

Say you have been working on some extremely experimental software, based on whatever revision you happened to checkout last week. If others in your group would like to work on this software with you, but without disturbing mainline development, you could commit your change to a new branch. Others can then checkout your experimental stuff and utilize the full benefit of cvs conflict resolution. The scenario might look like:

```
cvs tag -b EXPR
cvs update -rEXPR1
[[ hack away ]]
cvs commit
```

Others would simply do 'cvs checkout -rEXPR1whatever_module' to work with you on the experimental change.

4.6 diff [-kl][rcsdiff_options][[-r rev1 | -D date1][-r rev2 | -D date2]]

Requires: working directory, repository
Changes: nothing.
You can compare your working files with revisions in the source repository, with the 'cvs diff' command. If you don't specify a particular revision, your files are compared with the revisions they were based on. You can also use the standard cvs command option -r to specify a particular revision to compare your files with. Finally, if you use -r twice, you can see differences between two revisions in the repository. You can also specify -D options to diff against a revision in the past. The -r and -D options can be mixed together with at most two options ever specified.

See rcsdiff(1) for a list of other accepted options.
If you don't specify any files, diff will display differences for all those files in the current directory (and its subdirectories, unless you use the standard option -l) that differ from the corresponding revision in the source repository (i.e. files that you have changed),

4.7 export [-flNnQq]-r rev| -D date [-d dir]module...

Requires: repository.
Changes: current directory
This command is a variant of 'cvs checkout'; use it when you want a copy of the source for module without the cvs administrative directories. For example, you might use 'cvs export' to prepare source for shipment off -site. This command requires that you specify a date or tag (with -D or -r), so that you can count on reproducing the source you ship to others.
The only non -standard options are ' -d dir' (write the source into directory dir) and ' -N' (don't shorten module paths). These have the same meanings as the same options in 'cvs checkout'.
The -kv option is always set when export is used. This causes any RCS keywords to be expanded such that an import done at some other site will not lose the keyword revision information. There is no way

4.8 history [-report][-flags][-options args][files...]

Requires: the file '$CVSROOT/CVSROOT/history'
Changes: nothing.

cvs keeps a history file that tracks each use of the checkout, commit, rtag, update, and release commands. You can use 'cvs history' to display this information in various formats.

Warning: 'cvs history' uses ' -f', ' -l', ' -n' and ' -p' in ways that conflict with the descriptions in "Common Command Options".

Several options (shown above as -report) control what kind of report is generated:

-c Report on each time commit was used (i.e., each time the repository was modified).

-m module Report on a particular module. (You can meaningfully use -m more than once on the command line.)

-o Report on checked -out modules.

-T Report on all tags.

-x typ Extract a particular set of record types X from the cvs history. The types are indicated by single letters, which you may specify in combination. Certain commands have a single record type: checkout (type 'O'), release (type 'F'), and rtag (type 'T'). One of four record types may result from an update: 'W', when the working copy of a file is deleted during update (because it was gone from the repository); 'U', when a working file was copied from the repository; 'G', when a merge was necessary and it succeeded; and 'C', when a merge was necessary but collisions were detected (requiring manual merging). Finally, one of three record types results from commit: 'M', when a file was modified; 'A', when a file is first added; and 'R', when a file is removed.

-e Everything (all record types); equivalent to specifying ' -xMACFROGWUT'.

The options shown as -flags constrain the report without requiring option arguments:

a Show data for all users (the default is to show data only for the user executing ’cvs history’).

-l Show last modification only.

-w Show only the records for modifications done from the same working directory where ’cvs history’ is executing.

The options shown as -options args constrain the report based on an argument:

b str Show data back to a record containing the string str in either the module name, the file name, or the repository path.

-D date Show data since date.

-p repository Show data for a particular source repository (you can specify several -p options on the same command line).

-r rev Show records referring to revisions since the revision or tag named rev appears in individual RCS files. Each RCS file is searched for the revision or tag.

-t tag Show records since tag tag was last added to the history file. This differs from the -r flagabove in that it reads only the history file, not the RCS files, and is much faster.

-u name Show records for user name.

4.9 import [-options]repository vendortag releasetag...

Requires: Repository, source distribution directory

Changes: repository.

Use ’cvs import’ to incorporate an entire source distribution from an outside source (e.g., a source vendor) into your source repository directory. You can use this command both for initial creation of a repository, and for wholesale updates to the module form the outside source.

The repository argument gives a directory name (or a path to a directory) under the CVS root directory for repositories; if the directory did not exist, import creates it.

When you use import for updates to source that has been modified in your source repository (since a prior import), it will notify you of any files that conflict in the two branches of development; use ’cvs checkout -j’ to reconcile the differences, as import instructs you to do.

By default, certain file names are ignored during ’cvs import’: names associated with CVS administration, or with other common source control systems; common names for patch files, object files, archive files, and editor backup files; and other names that are usually artifacts of assorted utilities. Currently, the default list of ignored files includes files matching these names:

RCSLOG RCS SCCS
CVS* cvslog.*
tags TAGS
~# .#* ,*
*.old *.bak *.orig *.rej .del -*
*.a *.o *.Z *.elc *.ln core

The outside source is saved in a first -level RCS branch, by default ’1.1.1’. Updates are leaves of this branch; for example, files from the first imported collection of source will be revision ’1.1.1.1’, then files from the first imported update will be revision ’1.1.1.2’, and so on.

At least three arguments are required. repository is needed to identify the collection of source. vendor tag is a tag for the entire branch (e.g., for ’1.1.1’). You must also specify

at least one release tag to identify the files at the leaves created each time you execute 'cvs import'.
Three of the standard cvs command options are available: -Q, -q, and -m message. If you do not specify logging message with -m, your editor is invoked (as with commit) to allow you to enter one.
There are two additional special options.
Use ' -b branch' to specify a first -level branch other than '1.1.1'.
Use ' -I name' to specify file names that should be ignored during import. You can use this option repeatedly. To avoid ignoring any files at all (even those ignored by default), specify ' -I !'.

4.10 log [-l]rlog -options [files...]

Requires:	repository, working directory
Changes:	nothing.
Synonym:	rlog

Display log information for files. 'cvs log' calls the RCS utility rlog; all the options described in rlog(1)are available. Among the more useful rlog options are -h to display only the header (including tag definitions, but omitting most of the full log); -r to select logs on particular revisions or ranges of revisions; and -d to select particular dates or date ranges. See rlog(1) for full explanations. This command is recursive by default, unless the -l option is specified.

4.11 rdiff [-flags][-V vn][-r t| -D d [-r t2| -D d2]]modules...

Requires: repository.
Changes: nothing.
Synonym: patch
Builds a Larry Wall format patch(1) file between two releases, that can be fed directly into the patch program to bring an old release up -to -date with the new release. (This is one of the few cvs commands that operates directly from the repository, and doesn't require a prior checkout.) The diff output is sent to the standard output device. You can specify (using the standard -r and -D options) any combination of one or two revisions or dates. If only one revision or date is specified, the patch file reflects differences between that revision or date and the current "head" revisions in the RCS file.
Note that if the software release affected is contained in more than one directory, then it may be necessary to specify the -p option to the patch command when patching the old sources, so that patch is able to find the files that are located in other directories.
If you use the option -V vn, RCS keywords are expanded according to the rules current in RCS version vn (the expansion format changed with RCS version 5).
The standard option flags -f, -l, -Q, and -q are available with this command. There are also several special options flags:
If you use the -s option, no patch output is produced. Instead, a summary of the files changed or added between the two releases is sent to the standard output device. This is useful for finding out, for example, which files have changed between two dates or revisions.
If you use the -t option, a diff of the top two revisions is sent to the standard output device. This is most useful for seeing what the last change to a file was.
If you use the -u option, the patch output uses the newer "unidiff" format for context diffs. You can use -c to explicitly specify the 'diff -c' form of context

4.12 release [-dQq]modules...

Requires: Working directory
Changes: Working directory, history log

This command is meant to safely cancel the effect of 'cvs checkout'.
Since cvs doesn't lock files, it isn't strictly necessary to use this command. You can always simply delete your working directory, if you like; but you risk losing changes you may have forgotten, and you leave no trace in the cvs history file that you've abandoned your checkout.
Use 'cvs release' to avoid these problems. This command checks that no uncommitted changes are present; that you are executing it from immediately above, or inside, a cvs working directory; and that the repository recorded for your files is the same as the repository defined in the module database.
If all these conditions are true, 'cvs release' leaves a record of its execution (attesting to your intentionally abandoning your checkout) in the cvs history log.
You can use the -d flag to request that your working copies of the source files be deleted if the release succeeds.

4.13 remove [-lR][files...]modules...

Requires: Working directory
Changes: Working directory
Synonyms: m, delete

Use this command to declare that you wish to remove files from the source repository. Like most cvs commands, 'cvs remove' works on files in your working directory, not directly on the repository. As a safeguard, it also requires that you first erase the specified files from your working directory.
The files are not actually removed until you apply your changes to the repository with commit; at that point, the corresponding RCS files in the source repository are moved into the 'Attic' directory (also within the source repository).
This command is recursive by default, scheduling all physically removed files that it finds for removal by the next commit. Use the -l option to avoid this recursion, or just specify that actual files that you wish remove to consider.

4.14 rtag [-falnRQq][-b][-d][-r tag | -D date]symbolic_tag

Requires: repository.
Changes: repository.
Synonym: rfreeze

You can use this command to assign symbolic tags to particular, explicitly specified source versions in the repository. 'cvs rtag' works directly on the repository contents (and requires no prior checkout). Use 'cvs tag' instead, to base the selection of versions to tag on the contents of your working directory.
In general, tags (often the symbolic names of software distributions) should not be removed, but the -d option is available as a means to remove completely obsolete symbolic names if necessary (as might be the case for an Alpha release, say).
The -b option makes the tag a "branch" tag, allowing concurrent, isolated development. This is most useful for creating a patch to a previously released software distribution.
You can use the standard -r and -D options to tag only those files that already contain a certain tag. This method would be used to rename a tag: tag only the files identified by the old tag, then delete the old tag, leaving the new tag on exactly the same files as the

old tag.
rtag executes recursively by default, tagging all subdirectories of modules you specify in the argument. You can restrict its operation to toplevel directories with the standard -l option; or you can explicitly request recursion with -R.
The modules database can specify a program to execute whenever a tag is specified; a typical use is to send electronic mail to a group of interested parties. If you want to bypass that program, use the standard -n option.
The standard options -Q and -q are also available with this command.
Use the -a option to have rtag look in the 'Attic' for removed files that contain the specified tag. The tag is removed from these files, which makes it convenient to reuse a symbolic tag as development continues (and files get removed from the upcoming distribution).

4.15 status [-lR][-v][files...]

Requires: working directory, repository
Changes: nothing.
Display a brief report on the current status of files with respect to the source repository, including any "sticky" tags, dates, or -k options. ("Sticky" options will restrict how 'cvs update' operates until you reset them; see the description of 'cvs update -A...'.)
You can also use this command to anticipate the potential impact of a 'cvs update' on your working source directory. If you do not specify any files explicitly, reports are shown for all files that cvs has placed in your working directory. You can limit the scope of this search to the current directory itself (not its subdirectories) with the standard -l option flag; or you can explicitly request recursive status reports with the -R option.
The -v option causes the symbolic tags for the RCS fileto be displayed

4.16 tag [-lQqR][-b][-d]symbolic_tag [files...]

Requires: working directory, repository
Changes: repository.
Synonym: freeze
Use this command to assign symbolic tags to the nearest repository versions to your working sources. The tags are applied immediately to the repository, as with rtag, but the versions are supplied implicitly by the cvs records of your working files' history rather than applied explicitly.
One use for tags is to record a "snapshot" of the current sources when the software freeze date of a project arrives. As bugs are fixed after the freeze date, only those changed sources that are to be part of the release need be retagged.
The symbolic tags are meant to permanently record which revisions of which files were used in creating a software distribution. The checkout and update commands allow you to extract an exact copy of a tagged release at any time in the future, regardless of whether files have been changed, added, or removed since the release was tagged.
If you use 'cvs tag -d symbolic_tag...', the symbolic tag you specify is deleted instead of being added.
Warning: Be very certain of your ground before you delete a tag; doing this effectively discards some historical information, which may later turn out to have been valuable.
The -b option makes the tag a "branch" tag, allowing concurrent, isolated development. This is most useful for creating a patch to a previously released software distribution.
Normally, tag executes recursively through subdirectories; you can prevent this by using the standard -l option, or specify the recursion

4.17 update [-AdflPpQqR][-d][-r tag| -D date]files...

Requires: repository, working directory
Changes: working directory
After you've run checkout to create your private copy of source from the common repository, other developers will continue changing the central source. From time to time, when it is convenient in your development process, you can use the update command from within your working directory to reconcile your work with any revisions applied to the source repository since your last checkout or update.
update keeps you informed of its progress by printing a line for each file, prefaced with one of the characters 'U A R M C ?' to indicate the status of the file:

U file The file was brought up to date with respect to the repository. This is done for any file that exists in the repository but not in your source, and for files that you haven't changed but are not the most recent versions available in the repository.

A file The file has been added to your private copy of the sources, and will be added to the RCS source repository when you run 'cvs commit' on the file. This is a reminder to you that the file needs to be committed.

R file The file has been removed from your private copy of the sources, and will be removed from the RCS source repository when you run 'cvs commit' on the file. This is a reminder to you that the file needs to be committed.

M file The file is modified in your working directory. 'M' can indicate one of two states for a file you're working on: either there were no modifications to the same file in the repository, so that your file remains as you last saw it; or there were modifications in the repository as well as in your copy, but they were merged successfully, without conflict, in your working directory.

C file A conflict was detected while trying to merge your changes to file with changes from the source repository. file (the copy in your working directory) is now the output of the rcsmerge(1) command on the two versions; an unmodified copy of your file is also in your working directory, with the name '.#file.version', where version is the RCS revision that your modified file started from. (Note that some systems automatically purge files that begin with '.#' if they have not been accessed for a few days. If you intend to keep a copy of your original file, it is a very good idea to rename it.)

? file file is in your working directory, but does not correspond to anything in the source repository, and is not in the list of files for cvs to ignore (see the description of the -I option).

Use the -A option to reset any sticky tags, dates, or -k options. (If you get a working copy of a file by using one of the -r, -D, or -k options, cvs remembers the corresponding tag, date, or kflag and continues using it on future updates; use the -A option to make cvs forget these specifications, and retrieve the "head" version of the file).
The -j branch option merges the changes made between the resulting revision and the revision that it is based on (e.g., if the tag refers to a branch, cvs will merge all changes made in that branch into your working file).
With two -j options, cvs will merge in the changes between the two respective revisions. This can be used to "remove" a certain delta from your working file. E.g., if the file foo.c

is based on revision 1.6 and I want to remove the changes made between 1.3 and 1.5, I might do:
cvs update -j1.5 -j1.3 foo.c # note the order..
In addition, each -j option can contain on optionaldate specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag.
-jSymbolic_Tag:Date_Specifier
Use the -d option to create any directories that exist in the repository if they're missing from the working directory. (Normally, update acts only on directories and files that were already enrolled in your working directory.) This is useful for updating directories that were created in the repository since the initial checkout; but it has an unfortunate side effect. If you deliberately avoided certain directories in the repository when you created your working directory (either through use of a module name or by listing explicitly the files and directories you wanted on the command line), then updating with -d will create those directories, which may not be what you want.
Use -Iíname to ignore files whose names match name (in your working directory) during the update. You can specify -I more than once on the command line to specify several files to ignore. By default, update ignores files whose names match any of the following:
RCSLOG RCS SCCS
CVS* cvslog.*
tags TAGS
#̃ .#* ,*
*.old *.bak *.orig *.rej .del -*
*.a *.o *.Z *.elc *.ln core
Use ' -I !' to avoid ignoring any files at all.
The standard cvs command options -f, -k, -l, -P, -p, -Q, -q, and -r are also available with update.

5. Use of CVS

See also:

- History of introducing CVS in WWW code management
- Short Description of CVS
- CVS Manual, commands

5.1 WWW installation

The directory CVSROOT is where CVS stores its bookkeeping files.
The directory CVSRepository is where CVS stores the sources in the RCS format, i.e. one file for all versions and branches of each source file.
Please do not touch CVSROOT nor CVSRepository, they are to be maintained by use of CVS commands only.

5.2 Your .cshrc file

Code management in WWW is done with tools to be found in the directory hypertext/WWW/CVS-RCS.
The CVS repository for product xxx (xxx=Daemon, Library, LineMode, etc) is in WWW/xxx/Repository. Set envirnment variables to point to the repository and binary directories, for example:

```
setenv RCSBIN  $(HOME)/hypertext/WWW/CVS-RCS/next
setenv PATH ( $PATH $RCSBIN )
setenv CVSROOT $(HOME)/hypertext/WWW/CVS
setenv CVSREAD YES
```

You must also be a member of group www (gid=69) and have your umask set so that other group members will be able to write to files you create:

```
umask 2
```

5.3 Changing a module

This assumes you have NFS access to the WWW source tree.

5.3.1 Get the source

First of all, make yourself a working directory to be equivalent to "WWW" in the main tree. Call it WWW somewhere else, or WWW-joe for example. In that directory, if (say) you want to develop the Library code, do

```
cvs get Library
```

This will build you the sources. Alternatives are Line Mode and Daemon .

- Work on the sources.
- Test them.

5.3.2 Synchronise with other mods

Pick up any changes others have made using, in the Library/Implementation directory, the command

```
cvs update
```

5.3.3 Fix clashes

It is possible but surprisingly unlikely that someone else has been changing the same part of the same file and RCS can't figure out what the result should be. In this case, the file is flagged with big marks in it and a copy of both orginal files are left in your directory. Figure out what to do about it to combine your mods with the other guy's before proceeding.

5.3.4 Commit your work

```
cvs commit -m "Message describing changes and tested status"
```

while in the same (Implementation) directory. Your work will now be picked up by others working on the project.

5.4 Setting up a new module

If I remember rightly the sequence was to make a new subproduct xxx as follows

- cd $CVSROOT/CVSROOT
- Modify the "modules" file in that directory to add your module just like the others
- cvs commit modules
- Under WWW, make a directories WWW/xxx and WWW/Implementation
- cd WWW/xxx/Implementation
- Put the files in here –– actually they could be anywhere. Make sure the current directory has exactly and only the files you want to import. CVS will run through subdirectories recusively ok.
- cvs import xxx/Repository vendortag start
- cd ../.. (ie back up to WWW)
- cvs get xxx
- cd xxx/Implementation

5.5 CVS manual

The CVS and RCS manuals are in the web.
(Note: Terry Hung introduced it at SLAC, but I believe we got it from the "standard" sources: prep.ai.mit.edu.)
RC