Version Management with CVS

for CVS 1.9

Per Cederqvist et al

About this manual

Up to this point, one of the weakest parts of CVS has been the documentation. CVS is a complex program. Previous versions of the manual were written in the manual page format, which is not really well suited for such a complex program.

When writing this manual, I had several goals in mind:

- No knowledge of RCS should be necessary.
- No previous knowledge of revision control software should be necessary. All terms, such as *revision numbers*, *revision trees* and *merging* are explained as they are introduced.
- The manual should concentrate on the things CVS users want to do, instead of what the CVS commands can do. The first part of this manual leads you through things you might want to do while doing development, and introduces the relevant CVS commands as they are needed.
- Information should be easy to find. In the reference manual in the appendices almost all information about every CVS command is gathered together. There is also an extensive index, and a lot of cross references.

This manual was contributed by Signum Support AB in Sweden. Signum is yet another in the growing list of companies that support free software. You are free to copy both this manual and the CVS program. See Appendix E [Copying], page 111, for the details. Signum Support offers support contracts and binary distribution for many programs, such as CVS, GNU Emacs, the GNU C compiler and others. Write to us for more information.

```
Signum Support AB
Box 2044
S-580 02  Linkoping
Sweden

Email: info@signum.se
Phone: +46 (0)13 - 21 46 00
Fax:   +46 (0)13 - 21 47 00
```

Another company selling support for CVS is Cyclic Software, web: `http://www.cyclic.com/`, email: `info@cyclic.com`.

Checklist for the impatient reader

CVS is a complex system. You will need to read the manual to be able to use all of its capabilities. There are dangers that can easily be avoided if you know about them, and this manual tries to warn you about them. This checklist is intended to help you avoid the dangers without reading the entire manual. If you intend to read the entire manual you can skip this table.

Binary files

CVS can handle binary files, but you must have RCS release 5.5 or later and a release of GNU diff that supports the '`-a`' flag (release 1.15 and later are OK). You must also configure both RCS and CVS to handle binary files when you install them.

Keword substitution can be a source of trouble with binary files. See Chapter 16 [Keyword substitution], page 61, for solutions.

The `admin` command
: Careless use of the `admin` command can cause CVS to cease working. See Section A.5 [admin], page 73, before trying to use it.

Credits

Roland Pesch, Cygnus Support `<pesch@cygnus.com>` wrote the manual pages which were distributed with CVS 1.3. Appendix A and B contain much text that was extracted from them. He also read an early draft of this manual and contributed many ideas and corrections.

The mailing-list `info-cvs` is sometimes informative. I have included information from postings made by the following persons: David G. Grubbs `<dgg@think.com>`.

Some text has been extracted from the man pages for RCS.

The CVS FAQ by David G. Grubbs has provided useful material. The FAQ is no longer maintained, however, and this manual about the closest thing there is to a successor (with respect to documenting how to use CVS, at least).

In addition, the following persons have helped by telling me about mistakes I've made: Roxanne Brunskill `<rbrunski@datap.ca>`, Kathy Dyer `<dyer@phoenix.ocf.llnl.gov>`, Karl Pingle `<pingle@acuson.com>`, Thomas A Peterson `<tap@src.honeywell.com>`, Inge Wallin `<ingwa@signum.se>`, Dirk Koschuetzki `<koschuet@fmi.uni-passau.de>` and Michael Brown `<brown@wi.extrel.com>`.

BUGS

This manual is known to have room for improvement. Here is a list of known deficiencies:

- In the examples, the output from CVS is sometimes displayed, sometimes not.
- The input that you are supposed to type in the examples should have a different font than the output from the computer.
- This manual should be clearer about what file permissions you should set up in the repository, and about setuid/setgid.
- Some of the chapters are not yet complete. They are noted by comments in the '`cvs.texinfo`' file.
- This list is not complete. If you notice any error, omission, or something that is unclear, please send mail to `bug-cvs@prep.ai.mit.edu`.

I hope that you will find this manual useful, despite the above-mentioned shortcomings.

Linkoping, October 1993
Per Cederqvist

1 What is CVS?

CVS is a version control system. Using it, you can record the history of your source files.

For example, bugs sometimes creep in when software is modified, and you might not detect the bug until a long time after you make the modification. With CVS, you can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.

You could of course save every version of every file you have ever created. This would however waste an enormous amount of disk space. CVS stores all the versions of a file in a single file in a clever way that only stores the differences between versions.

CVS also helps you if you are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless you are extremely careful. Some editors, like GNU Emacs, try to make sure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.

CVS started out as a bunch of shell scripts written by Dick Grune, posted to `comp.sources.unix` in the volume 6 release of December, 1986. While no actual code from these shell scripts is present in the current version of CVS much of the CVS conflict resolution algorithms come from them.

In April, 1989, Brian Berliner designed and coded CVS. Jeff Polk later helped Brian with the design of the CVS module and vendor branch support.

You can get CVS via anonymous ftp from a number of sites, for instance `prep.ai.mit.edu` in '`pub/gnu`'.

There is a mailing list, known as `info-cvs`, devoted to CVS. To subscribe or unsubscribe send a message to `info-cvs-request@prep.ai.mit.edu`. Please be specific about your email address. As of May 1996, subscription requests are handled by a busy human being, so you cannot expect to be added or removed immediately. The usenet group `comp.software.config-mgmt` is also a suitable place for CVS discussions (along with other configuration management systems).

CVS is not...

CVS can do a lot of things for you, but it does not try to be everything for everyone.

CVS is not a build system.

Though the structure of your repository and modules file interact with your build system (e.g. '`Makefile`'s), they are essentially independent.

CVS does not dictate how you build anything. It merely stores files for retrieval in a tree structure you devise.

CVS does not dictate how to use disk space in the checked out working directories. If you write your '`Makefile`'s or scripts in every directory so they have to know the relative positions of everything else, you wind up requiring the entire repository to be checked out.

If you modularize your work, and construct a build system that will share files (via links, mounts, `VPATH` in '`Makefile`'s, etc.), you can arrange your disk usage however you like.

But you have to remember that *any* such system is a lot of work to construct and maintain. CVS does not address the issues involved.

Of course, you should place the tools created to support such a build system (scripts, '`Makefile`'s, etc) under CVS.

Figuring out what files need to be rebuilt when something changes is, again, something to be handled outside the scope of CVS. One traditional approach is to use `make` for building, and use some automated tool for generating the depencies which `make` uses.

CVS is not a substitute for management.

Your managers and project leaders are expected to talk to you frequently enough to make certain you are aware of schedules, merge points, branch names and release dates. If they don't, CVS can't help.

CVS is an instrument for making sources dance to your tune. But you are the piper and the composer. No instrument plays itself or writes its own music.

CVS is not a substitute for developer communication.

When faced with conflicts within a single file, most developers manage to resolve them without too much effort. But a more general definition of "conflict" includes problems too difficult to solve without communication between developers.

CVS cannot determine when simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. Its concept of a *conflict* is purely textual, arising when two changes to the same base file are near enough to spook the merge (i.e. `diff3`) command.

CVS does not claim to help at all in figuring out non-textual or distributed conflicts in program logic.

For example: Say you change the arguments to function `X` defined in file '`A`'. At the same time, someone edits file '`B`', adding new calls to function `X` using the old arguments. You are outside the realm of CVS's competence.

Acquire the habit of reading specs and talking to your peers.

CVS does not have change control

Change control refers to a number of things. First of all it can mean *bug-tracking*, that is being able to keep a database of reported bugs and the status of each one (is it fixed? in what release? has the bug submitter agreed that it is fixed?). For interfacing CVS to an external bug-tracking system, see the '`rcsinfo`' and '`editinfo`' files (see Appendix B [Administrative files], page 97).

Another aspect of change control is keeping track of the fact that changes to several files were in fact changed together as one logical change. If you check in several files in a single `cvs commit` operation, CVS then forgets that those files were checked in together, and the fact that they have the same log message is the only thing tying them together. Keeping a GNU style '`ChangeLog`' can help somewhat.

Another aspect of change control, in some systems, is the ability to keep track of the status of each change. Some changes have been written by a developer, others have been reviewed by a second developer, and so on. Generally, the way to do this with CVS is to generate a diff (using `cvs diff` or `diff`) and email it to someone who can then apply it using the `patch` utility. This is very flexible, but depends on mechanisms outside CVS to make sure nothing falls through the cracks.

CVS is not an automated testing program

It should be possible to enforce mandatory use of a testsuite using the `commitinfo` file. I haven't heard a lot about projects trying to do that or whether there are subtle gotchas, however.

CVS does not have a builtin process model

Some systems provide ways to ensure that changes or releases go through various steps, with various approvals as needed. Generally, one can accomplish this with CVS but it might be a little more work. In some cases you'll want to use the '`commitinfo`', '`loginfo`', '`rcsinfo`', or '`editinfo`' files, to require that certain steps be performed before cvs will allow a checkin. Also consider whether features such as branches and tags can be used to perform tasks such as doing work in a development tree and then merging certain changes over to a stable tree only once they have been proven.

2 Basic concepts

CVS stores all files in a centralized *repository* (see Chapter 4 [Repository], page 13).

The repository contains directories and files, in an arbitrary tree. The *modules* feature can be used to group together a set of directories or files into a single entity (see Section B.1 [modules], page 97). A typical usage is to define one module per project.

2.1 Revision numbers

Each version of a file has a unique *revision number*. Revision numbers look like '1.1', '1.2', '1.3.2.2' or even '1.3.2.2.4.5'. A revision number always has an even number of period-separated decimal integers. By default revision 1.1 is the first revision of a file. Each successive revision is given a new number by increasing the rightmost number by one. The following figure displays a few revisions, with newer revisions to the right.

```
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !
+-----+    +-----+    +-----+    +-----+    +-----+
```

CVS is not limited to linear development. The *revision tree* can be split into *branches*, where each branch is a self-maintained line of development. Changes made on one branch can easily be moved back to the main trunk.

Each branch has a *branch number*, consisting of an odd number of period-separated decimal integers. The branch number is created by appending an integer to the revision number where the corresponding branch forked off. Having branch numbers allows more than one branch to be forked off from a certain revision.

All revisions on a branch have revision numbers formed by appending an ordinal number to the branch number. The following figure illustrates branching with an example.

```
                                                     +-------------+
                          Branch 1.2.2.3.2 ->        ! 1.2.2.3.2.1 !
                                                   / +-------------+
                                                  /
                                                 /
                 +---------+    +---------+    +---------+    +---------+
Branch 1.2.2 -> _! 1.2.2.1 !----! 1.2.2.2 !----! 1.2.2.3 !----! 1.2.2.4 !
               / +---------+    +---------+    +---------+    +---------+
              /
             /
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+    +-----+
                !
                !
                !   +---------+    +---------+    +---------+
Branch 1.2.4 -> +---! 1.2.4.1 !----! 1.2.4.2 !----! 1.2.4.3 !
                    +---------+    +---------+    +---------+
```

The exact details of how the branch number is constructed is not something you normally need to be concerned about, but here is how it works: When CVS creates a branch number it picks the first unused even integer, starting with 2. So when you want to create a branch from revision 6.4 it will be numbered 6.4.2. All branch numbers ending in a zero (such as 6.4.0) are used internally by CVS (see Section D.1 [Magic branch numbers], page 109). The branch 1.1.1 has a special meaning. See Chapter 12 [Tracking sources], page 53.

2.2 Versions, revisions and releases

A file can have several versions, as described above. Likewise, a software product can have several versions. A software product is often given a version number such as '`4.1.1`'.

Versions in the first sense are called *revisions* in this document, and versions in the second sense are called *releases*. To avoid confusion, the word *version* is almost never used in this document.

3 A sample session

This section describes a typical work-session using CVS. It assumes that a repository is set up (see Chapter 4 [Repository], page 13).

Suppose you are working on a simple compiler. The source consists of a handful of C files and a '`Makefile`'. The compiler is called '`tc`' (Trivial Compiler), and the repository is set up so that there is a module called '`tc`'.

3.1 Getting the source

The first thing you must do is to get your own working copy of the source for '`tc`'. For this, you use the `checkout` command:

```
$ cvs checkout tc
```

This will create a new directory called '`tc`' and populate it with the source files.

```
$ cd tc
$ ls
CVS         Makefile    backend.c   driver.c    frontend.c  parser.c
```

The '`CVS`' directory is used internally by CVS. Normally, you should not modify or remove any of the files in it.

You start your favorite editor, hack away at '`backend.c`', and a couple of hours later you have added an optimization pass to the compiler. A note to RCS and SCCS users: There is no need to lock the files that you want to edit. See Chapter 6 [Multiple developers], page 27 for an explanation.

3.2 Committing your changes

When you have checked that the compiler is still compilable you decide to make a new version of '`backend.c`'.

```
$ cvs commit backend.c
```

CVS starts an editor, to allow you to enter a log message. You type in "Added an optimization pass.", save the temporary file, and exit the editor.

The environment variable `$CVSEDITOR` determines which editor is started. If `$CVSEDITOR` is not set, then if the environment variable `$EDITOR` is set, it will be used. If both `$CVSEDITOR` and `$EDITOR` are not set then the editor defaults to `vi`. If you want to avoid the overhead of starting an editor you can specify the log message on the command line using the '`-m`' flag instead, like this:

```
$ cvs commit -m "Added an optimization pass" backend.c
```

3.3 Cleaning up

Before you turn to other tasks you decide to remove your working copy of tc. One acceptable way to do that is of course

```
$ cd ..
$ rm -r tc
```

but a better way is to use the `release` command (see Section A.14 [release], page 90):

```
$ cd ..
$ cvs release -d tc
M driver.c
? tc
You have [1] altered files in this repository.
Are you sure you want to release (and delete) module ‘tc’: n
** ‘release’ aborted by user choice.
```

The `release` command checks that all your modifications have been committed. If history logging is enabled it also makes a note in the history file. See Section B.9 [history file], page 104.

When you use the ‘`-d`’ flag with `release`, it also removes your working copy.

In the example above, the `release` command wrote a couple of lines of output. ‘`? tc`’ means that the file ‘`tc`’ is unknown to CVS. That is nothing to worry about: ‘`tc`’ is the executable compiler, and it should not be stored in the repository. See Section B.8 [cvsignore], page 103, for information about how to make that warning go away. See Section A.14.2 [release output], page 90, for a complete explanation of all possible output from `release`.

‘`M driver.c`’ is more serious. It means that the file ‘`driver.c`’ has been modified since it was checked out.

The `release` command always finishes by telling you how many modified files you have in your working copy of the sources, and then asks you for confirmation before deleting any files or making any note in the history file.

You decide to play it safe and answer *`n`* RET when `release` asks for confirmation.

3.4 Viewing differences

You do not remember modifying ‘`driver.c`’, so you want to see what has happened to that file.

```
$ cd tc
$ cvs diff driver.c
```

This command runs `diff` to compare the version of ‘`driver.c`’ that you checked out with your working copy. When you see the output you remember that you added a command line option that enabled the optimization pass. You check it in, and release the module.

```
$ cvs commit -m "Added an optimization pass" driver.c
Checking in driver.c;
/usr/local/cvsroot/tc/driver.c,v  <--  driver.c
new revision: 1.2; previous revision: 1.1
done
$ cd ..
$ cvs release -d tc
? tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module ‘tc’: y
```

4 The Repository

The CVS *repository* stores a complete copy of all the files and directories which are under version control.

Normally, you never access any of the files in the repository directly. Instead, you use CVS commands to get your own copy of the files, and then work on that copy. When you've finished a set of changes, you check (or *commit*) them back into the repository. The repository then contains the changes which you have made, as well as recording exactly what you changed, when you changed it, and other such information.

CVS can access a repository by a variety of means. It might be on the local computer, or it might be on a computer across the room or across the world. To distinguish various ways to access a repository, the repository name can start with an *access method*. For example, the access method `:local:` means to access a repository directory, so the repository `:local:/usr/local/cvsroot` means that the repository is in '`/usr/local/cvsroot`' on the computer running CVS. For information on other access methods, see Section 4.6 [Remote repositories], page 17.

If the access method is omitted, then if the repository does not contain '`:`', then `:local:` is assumed. If it does contain '`:`' than either `:ext:` or `:server:` is assumed. For example, if you have a local repository in '`/usr/local/cvsroot`', you can use `/usr/local/cvsroot` instead of `:local:/usr/local/cvsroot`. But if (under Windows NT, for example) your local repository is '`c:\src\cvsroot`', then you must specify the access method, as in `:local:c:\src\cvsroot`.

The repository is split in two parts. '`$CVSROOT/CVSROOT`' contains administrative files for CVS. The other directories contain the actual user-defined modules.

4.1 Telling CVS where your repository is

There are a couple of different ways to tell CVS where to find the repository. You can name the repository on the command line explicitly, with the `-d` (for "directory") option:

```
cvs -d /usr/local/cvsroot checkout yoyodyne/tc
```

Or you can set the `$CVSROOT` environment variable to an absolute path to the root of the repository, '`/usr/local/cvsroot`' in this example. To set `$CVSROOT`, all `csh` and `tcsh` users should have this line in their '`.cshrc`' or '`.tcshrc`' files:

```
setenv CVSROOT /usr/local/cvsroot
```

`sh` and `bash` users should instead have these lines in their '`.profile`' or '`.bashrc`':

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

A repository specified with `-d` will override the `$CVSROOT` environment variable. Once you've checked a working copy out from the repository, it will remember where its repository is (the information is recorded in the '`CVS/Root`' file in the working copy).

The `-d` option and the '`CVS/Root`' file both override the `$CVSROOT` environment variable. If `-d` option differs from '`CVS/Root`', the former is used (and specifying `-d` will cause '`CVS/Root`' to be updated). Of course, for proper operation they should be two ways of referring to the same repository.

4.2 How data is stored in the repository

For most purposes it isn't important *how* CVS stores information in the repository. In fact, the format has changed in the past, and is likely to change in the future. Since in almost all cases one accesses the repository via CVS commands; such changes need not be disruptive.

However, in some cases it may be necessary to understand how CVS stores data in the repository, for example you might need to track down CVS locks (see Section 6.5 [Concurrency], page 31) or you might need to deal with the file permissions appropriate for the repository.

4.2.1 Where files are stored within the repository

The overall structure of the repository is a directory tree corresponding to the directories in the working directory. For example, supposing the repository is in '`/usr/local/cvsroot`', here is a possible directory tree (showing only the directories):

```
/usr
 |
 +--local
 |    |
 |    +--cvsroot
 |    |    |
 |    |    +--CVSROOT
          |      (administrative files)
          |
          +--gnu
          |    |
          |    +--diff
          |    |   (source code to GNU diff)
          |    |
          |    +--rcs
          |    |   (source code to RCS)
          |    |
          |    +--cvs
          |        (source code to CVS)
          |
          +--yoyodyne
               |
               +--tc
               |    |
               |    +--man
               |    |
               |    +--testing
               |
                  +--(other Yoyodyne software)
```

With the directories are *history files* for each file under version control. The name of the history file is the name of the corresponding file with '`,v`' appended to the end. Here is what the repository for the '`yoyodyne/tc`' directory might look like:

```
$CVSROOT
  |
  +--yoyodyne
  |   |
  |   +--tc
  |   |   |
          +--Makefile,v
          +--backend.c,v
          +--driver.c,v
          +--frontend.c,v
          +--parser.c,v
          +--man
          |    |
          |    +--tc.1,v
          |
          +--testing
               |
               +--testpgm.t,v
               +--test2.t,v
```

The history files contain, among other things, enough information to recreate any revision of the file, a log of all commit messages and the user-name of the person who committed the revision. The history files are known as *RCS files*, because the first program to store files in that format was a version control system known as RCS. For a full description of the file format, see the `man` page *rcsfile(5)*, distributed with RCS. This file format has become very common—many systems other than CVS or RCS can at least import history files in this format.

4.2.2 File permissions

All '`,v`' files are created read-only, and you should not change the permission of those files. The directories inside the repository should be writable by the persons that have permission to modify the files in each directory. This normally means that you must create a UNIX group (see group(5)) consisting of the persons that are to edit the files in a project, and set up the repository so that it is that group that owns the directory.

This means that you can only control access to files on a per-directory basis.

Note that users must also have write access to check out files, because CVS needs to create lock files (see Section 6.5 [Concurrency], page 31).

Also note that users must have write access to the '`CVSROOT/val-tags`' file. CVS uses it to keep track of what tags are valid tag names (it is sometimes updated when tags are used, as well as when they are created, though).

CVS tries to set up reasonable file permissions for new directories that are added inside the tree, but you must fix the permissions manually when a new directory should have different permissions than its parent directory. If you set the `CVSUMASK` environment variable that will control the file permissions which CVS uses in creating directories and/or files in the repository. `CVSUMASK` does not affect the file permissions in the working directory; such files have the permissions which are typical for newly created files, except that sometimes CVS creates them read-only (see the sections on watches, Section 6.6.1 [Setting a watch], page 32; -r, Section A.3 [Global options], page 70; or CVSREAD, Appendix C [Environment variables], page 107).

Since CVS was not written to be run setuid, it is unsafe to try to run it setuid. You cannot use the setuid features of RCS together with CVS.

4.3 The administrative files

The directory '`$CVSROOT/CVSROOT`' contains some *administrative files*. See Appendix B [Administrative files], page 97, for a complete description. You can use CVS without any of these files, but some commands work better when at least the '`modules`' file is properly set up.

The most important of these files is the '`modules`' file. It defines all modules in the repository. This is a sample '`modules`' file.

```
CVSROOT         CVSROOT
modules         CVSROOT modules
cvs             gnu/cvs
rcs             gnu/rcs
diff            gnu/diff
tc              yoyodyne/tc
```

The '`modules`' file is line oriented. In its simplest form each line contains the name of the module, whitespace, and the directory where the module resides. The directory is a path relative to `$CVSROOT`. The last four lines in the example above are examples of such lines.

The line that defines the module called '`modules`' uses features that are not explained here. See Section B.1 [modules], page 97, for a full explanation of all the available features.

4.3.1 Editing administrative files

You edit the administrative files in the same way that you would edit any other module. Use '`cvs checkout CVSROOT`' to get a working copy, edit it, and commit your changes in the normal way.

It is possible to commit an erroneous administrative file. You can often fix the error and check in a new revision, but sometimes a particularly bad error in the administrative file makes it impossible to commit new revisions.

4.4 Multiple repositories

In some situations it is a good idea to have more than one repository, for instance if you have two development groups that work on separate projects without sharing any code. All you have to do to have several repositories is to specify the appropriate repository, using the `CVSROOT` environment variable, the '`-d`' option to CVS, or (once you have checked out a working directory) by simply allowing CVS to use the repository that was used to check out the working directory (see Section 4.1 [Specifying a repository], page 13).

The big advantage of having multiple repositories is that they can reside on different servers. The big disadvantage is that you cannot have a single CVS command recurse into directories which comes from different repositories. Generally speaking, if you are thinking of setting up several repositories on the same machine, you might want to consider using several directories within the same repository.

None of the examples in this manual show multiple repositories.

4.5 Creating a repository

To set up a CVS repository, choose a directory with ample disk space available for the revision history of the source files. It should be accessable (directly or via a networked file system) from all machines which want to use CVS in server or local mode; the client machines need not have any access to it other than via the CVS protocol. It is not possible to use CVS to read from a repository which one only has read access to; CVS needs to be able to create lock files (see Section 6.5 [Concurrency], page 31).

To create a repository, run the `cvs init` command. It will set up an empty repository in the CVS root specified in the usual way (see Chapter 4 [Repository], page 13). For example,

```
cvs -d /usr/local/cvsroot init
```

`cvs init` is careful to never overwrite any existing files in the repository, so no harm is done if you run `cvs init` on an already set-up repository.

`cvs init` will enable history logging; if you don't want that, remove the history file after running `cvs init`. See Section B.9 [history file], page 104.

4.6 Remote repositories

Your working copy of the sources can be on a different machine than the repository. Generally, using a remote repository is just like using a local one, except that the format of the repository name is:

```
:method:user@hostname:/path/to/repository
```

The details of exactly what needs to be set up depend on how you are connecting to the server.

If *method* is not specified, and the repository name contains ':', then the default is `ext` or `server`, depending on your platform; both are described in Section 4.6.1 [Connecting via rsh], page 18.

4.6.1 Connecting with rsh

CVS uses the '`rsh`' protocol to perform these operations, so the remote user host needs to have a '`.rhosts`' file which grants access to the local user.

For example, suppose you are the user '`mozart`' on the local machine '`anklet.grunge.com`', and the server machine is '`chainsaw.brickyard.com`'. On chainsaw, put the following line into the file '`.rhosts`' in '`bach`''s home directory:

```
anklet.grunge.com  mozart
```

Then test that `rsh` is working with

```
rsh -l bach chainsaw.brickyard.com 'echo $PATH'
```

Next you have to make sure that `rsh` will be able to find the server. Make sure that the path which `rsh` printed in the above example includes the directory containing a program named `cvs` which is the server. You need to set the path in '`.bashrc`', '`.cshrc`', etc., not '`.login`' or '`.profile`'. Alternately, you can set the environment variable `CVS_SERVER` on the client machine to the filename of the server you want to use, for example '`/usr/local/bin/cvs-1.6`'.

There is no need to edit `inetd.conf` or start a CVS server daemon.

There are two access methods that you use in CVSROOT for rsh. `:server:` specifies an internal rsh client, which is supported only by some CVS ports. `:ext:` specifies an external rsh program. By default this is `rsh` but you may set the `CVS_RSH` environment variable to invoke another program which can access the remote server (for example, `remsh` on HP-UX 9 because `rsh` is something different). It must be a program which can transmit data to and from the server without modifying it; for example the Windows NT `rsh` is not suitable since it by default translates between CRLF and LF. The OS/2 CVS port has a hack to pass '`-b`' to `rsh` to get around this, but since this could potentially cause programs for programs other than the standard `rsh`, it may change in the future. If you set `CVS_RSH` to `SSH` or some other rsh replacement, the instructions in the rest of this section concerning '`.rhosts`' and so on are likely to be incorrect; consult the documentation for your rsh replacement.

Continuing our example, supposing you want to access the module '`foo`' in the repository '`/usr/local/cvsroot/`', on machine '`chainsaw.brickyard.com`', you are ready to go:

```
cvs -d :ext:bach@chainsaw.brickyard.com:/usr/local/cvsroot checkout foo
```

(The '`bach@`' can be omitted if the username is the same on both the local and remote hosts.)

4.6.2 Direct connection with password authentication

The CVS client can also connect to the server using a password protocol. This is particularly useful if using `rsh` is not feasible (for example, the server is behind a firewall), and Kerberos also is not available.

To use this method, it is necessary to make some adjustments on both the server and client sides.

4.6.2.1 Setting up the server for password authentication

On the server side, the file '`/etc/inetd.conf`' needs to be edited so `inetd` knows to run the command `cvs pserver` when it receives a connection on the right port. By default, the port number is 2401; it would be different if your client were compiled with `CVS_AUTH_PORT` defined to something else, though.

If your `inetd` allows raw port numbers in '`/etc/inetd.conf`', then the following (all on a single line in '`inetd.conf`') should be sufficient:

```
2401  stream  tcp  nowait  root  /usr/local/bin/cvs
cvs -b /usr/local/bin pserver
```

The '`-b`' option specifies the directory which contains the RCS binaries on the server. You could also use the '`-T`' option to specify a temporary directory.

If your `inetd` wants a symbolic service name instead of a raw port number, then put this in '`/etc/services`':

```
cvspserver      2401/tcp
```

and put `cvspserver` instead of `2401` in '`inetd.conf`'.

Once the above is taken care of, restart your `inetd`, or do whatever is necessary to force it to reread its initialization files.

Because the client stores and transmits passwords in cleartext (almost—see Section 4.6.2.3 [Password authentication security], page 21 for details), a separate CVS password file may be used, so people don't compromise their regular passwords when they access the repository. This file is '`$CVSROOT/CVSROOT/passwd`' (see Section 4.3 [Intro administrative files], page 16). Its format is similar to '`/etc/passwd`', except that it only has two fields, username and password. For example:

```
bach:ULtgRLXo7NRxs
cwang:1sOp854gDF3DY
```

The password is encrypted according to the standard Unix `crypt()` function, so it is possible to paste in passwords directly from regular Unix '`passwd`' files.

When authenticating a password, the server first checks for the user in the CVS '`passwd`' file. If it finds the user, it compares against that password. If it does not find the user, or if the CVS '`passwd`' file does not exist, then the server tries to match the password using the system's user-lookup routine. When using the CVS '`passwd`' file, the server runs under as the username specified in the the third argument in the entry, or as the first argument if there is no third argument (in this way CVS allows imaginary usernames provided the CVS '`passwd`' file indicates corresponding valid system usernames). In any case, CVS will have no privileges which the (valid) user would not have.

Right now, the only way to put a password in the CVS '`passwd`' file is to paste it there from somewhere else. Someday, there may be a `cvs passwd` command.

4.6.2.2 Using the client with password authentication

Before connecting to the server, the client must *log in* with the command `cvs login`. Logging in verifies a password with the server, and also records the password for later transactions with the server. The `cvs login` command needs to know the username, server hostname, and full repository path, and it gets this information from the repository argument or the `CVSROOT` environment variable.

`cvs login` is interactive — it prompts for a password:

```
cvs -d :pserver:bach@chainsaw.brickyard.com:/usr/local/cvsroot login
CVS password:
```

The password is checked with the server; if it is correct, the `login` succeeds, else it fails, complaining that the password was incorrect.

Once you have logged in, you can force CVS to connect directly to the server and authenticate with the stored password:

```
cvs -d :pserver:bach@chainsaw.brickyard.com:/usr/local/cvsroot checkout foo
```

The '`:pserver:`' is necessary because without it, CVS will assume it should use `rsh` to connect with the server (see Section 4.6.1 [Connecting via rsh], page 18). (Once you have a working copy checked out and are running CVS commands from within it, there is no longer any need to specify the repository explicitly, because CVS records it in the working copy's '`CVS`' subdirectory.)

Passwords are stored by default in the file '`$HOME/.cvspass`'. Its format is human-readable, but don't edit it unless you know what you are doing. The passwords are not stored in cleartext, but are trivially encoded to protect them from "innocent" compromise (i.e., inadvertently being seen by a system administrator who happens to look at that file).

The `CVS_PASSFILE` environment variable overrides this default. If you use this variable, make sure you set it *before* `cvs login` is run. If you were to set it after running `cvs login`, then later CVS commands would be unable to look up the password for transmission to the server.

The `CVS_PASSWORD` environment variable overrides *all* stored passwords. If it is set, CVS will use it for all password-authenticated connections.

4.6.2.3 Security considerations with password authentication

The passwords are stored on the client side in a trivial encoding of the cleartext, and transmitted in the same encoding. The encoding is done only to prevent inadvertent password compromises (i.e., a system administrator accidentally looking at the file), and will not prevent even a naive attacker from gaining the password.

The separate CVS password file (see Section 4.6.2.1 [Password authentication server], page 19) allows people to use a different password for repository access than for login access. On the other hand, once a user has access to the repository, she can execute programs on the server system through a variety of means. Thus, repository access implies fairly broad system access as well. It might be possible to modify CVS to prevent that, but no one has done so as of this writing. Furthermore, there may be other ways in which having access to CVS allows people to gain more general access to the system; noone has done a careful audit.

In summary, anyone who gets the password gets repository access, and some measure of general system access as well. The password is available to anyone who can sniff network packets or read a protected (i.e., user read-only) file. If you want real security, get Kerberos.

4.6.3 Direct connection with kerberos

The main disadvantage of using rsh is that all the data needs to pass through additional programs, so it may be slower. So if you have kerberos installed you can connect via a direct TCP connection, authenticating with kerberos.

To do this, CVS needs to be compiled with kerberos support; when configuring CVS it tries to detect whether kerberos is present or you can use the '`--with-krb4`' flag to configure.

The data transmitted is *not* encrypted by default. Encryption support must be compiled into both the client and server; use the '`--enable-encryption`' configure option to turn it on. You must then use the `-x` global option to request encryption.

You need to edit `inetd.conf` on the server machine to run `cvs kserver`. The client uses port 1999 by default; if you want to use another port specify it in the `CVS_CLIENT_PORT` environment variable on the client.

When you want to use CVS, get a ticket in the usual way (generally `kinit`); it must be a ticket which allows you to log into the server machine. Then you are ready to go:

```
cvs -d :kserver:chainsaw.brickyard.com:/user/local/cvsroot checkout foo
```

Previous versions of CVS would fall back to a connection via rsh; this version will not do so.

5 Starting a project with CVS

Because renaming files and moving them between directories is somewhat inconvenient, the first thing you do when you start a new project should be to think through your file organization. It is not impossible to rename or move files, but it does increase the potential for confusion and CVS does have some quirks particularly in the area of renaming directories. See Chapter 13 [Moving files], page 55.

What to do next depends on the situation at hand.

5.1 Setting up the files

The first step is to create the files inside the repository. This can be done in a couple of different ways.

5.1.1 Creating a directory tree from a number of files

When you begin using CVS, you will probably already have several projects that can be put under CVS control. In these cases the easiest way is to use the `import` command. An example is probably the easiest way to explain how to use it. If the files you want to install in CVS reside in '*wdir*', and you want them to appear in the repository as '`$CVSROOT/yoyodyne/`*rdir*', you can do this:

```
$ cd wdir
$ cvs import -m "Imported sources" yoyodyne/rdir yoyo start
```

Unless you supply a log message with the '`-m`' flag, CVS starts an editor and prompts for a message. The string '`yoyo`' is a *vendor tag*, and '`start`' is a *release tag*. They may fill no purpose in this context, but since CVS requires them they must be present. See Chapter 12 [Tracking sources], page 53, for more information about them.

You can now verify that it worked, and remove your original source directory.

```
$ cd ..
$ mv dir dir.orig
$ cvs checkout yoyodyne/dir       # Explanation below
$ ls -R yoyodyne
$ rm -r dir.orig
```

Erasing the original sources is a good idea, to make sure that you do not accidentally edit them in *dir*, bypassing CVS. Of course, it would be wise to make sure that you have a backup of the sources before you remove them.

The `checkout` command can either take a module name as argument (as it has done in all previous examples) or a path name relative to `$CVSROOT`, as it did in the example above.

It is a good idea to check that the permissions CVS sets on the directories inside '$CVSROOT' are reasonable, and that they belong to the proper groups. See Section 4.2.2 [File permissions], page 15.

If some of the files you want to import are binary, you may want to use the wrappers features to specify which files are binary and which are not. See Section B.2 [Wrappers], page 98.

5.1.2 Creating Files From Other Version Control Systems

If you have a project which you are maintaining with another version control system, such as RCS, you may wish to put the files from that project into CVS, and preserve the revision history of the files.

From RCS
: If you have been using RCS, find the RCS files—usually a file named '`foo.c`' will have its RCS file in '`RCS/foo.c,v`' (but it could be other places; consult the RCS documentation for details). Then create the appropriate directories in CVS if they do not already exist. Then copy the files into the appropriate directories in the CVS repository (the name in the repository must be the name of the source file with '`,v`' added; the files go directly in the appopriate directory of the repository, not in an '`RCS`' subdirectory). This is one of the few times when it is a good idea to access the CVS repository directly, rather than using CVS commands. Then you are ready to check out a new working directory.

  The RCS file should not be locked when you move it into CVS; if it is, CVS will have trouble letting you operate on it.

From another version control system
: Many version control systems have the ability to export RCS files in the standard format. If yours does, export the RCS files and then follow the above instructions.

From SCCS
: There is a script in the '`contrib`' directory of the CVS source distribution called '`sccs2rcs`' which converts SCCS files to RCS files. Note: you must run it on a machine which has both SCCS and RCS installed, and like everything else in contrib it is unsupported (your mileage may vary).

5.1.3 Creating a directory tree from scratch

For a new project, the easiest thing to do is probably to create an empty directory structure, like this:

```
$ mkdir tc
$ mkdir tc/man
$ mkdir tc/testing
```

After that, you use the `import` command to create the corresponding (empty) directory structure inside the repository:

```
$ cd tc
$ cvs import -m "Created directory structure" yoyodyne/dir yoyo start
```

Then, use `add` to add files (and new directories) as they appear.

Check that the permissions CVS sets on the directories inside '`$CVSROOT`' are reasonable.

5.2 Defining the module

The next step is to define the module in the '`modules`' file. This is not strictly necessary, but modules can be convenient in grouping together related files and directories.

In simple cases these steps are sufficient to define a module.

1. Get a working copy of the modules file.

   ```
   $ cvs checkout CVSROOT/modules
   $ cd CVSROOT
   ```
2. Edit the file and insert a line that defines the module. See Section 4.3 [Intro administrative files], page 16, for an introduction. See Section B.1 [modules], page 97, for a full description of the modules file. You can use the following line to define the module '`tc`':

   ```
   tc   yoyodyne/tc
   ```
3. Commit your changes to the modules file.

   ```
   $ cvs commit -m "Added the tc module." modules
   ```
4. Release the modules module.

   ```
   $ cd ..
   $ cvs release -d CVSROOT
   ```

6 Multiple developers

When more than one person works on a software project things often get complicated. Often, two people try to edit the same file simultaneously. One solution, known as *file locking* or *reserved checkouts*, is to allow only one person to edit each file at a time. This is the only solution with some version control systems, including RCS and SCCS. CVS doesn't have a very nice implementation of reserved checkouts (yet) but there are ways to get it working (for example, see the `cvs admin -l` command in Section A.5.1 [admin options], page 74). It also may be possible to use the watches features described below, together with suitable procedures (not enforced by software), to avoid having two people edit at the same time.

The default model with CVS is known as *unreserved checkouts*. In this model, developers can edit their own *working copy* of a file simultaneously. The first person that commits his changes has no automatic way of knowing that another has started to edit it. Others will get an error message when they try to commit the file. They must then use CVS commands to bring their working copy up to date with the repository revision. This process is almost automatic.

CVS also supports mechanisms which facilitate various kinds of communcation, without actually enforcing rules like reserved checkouts do.

The rest of this chapter describes how these various models work, and some of the issues involved in choosing between them.

6.1 File status

Based on what operations you have performed on a checked out file, and what operations others have performed to that file in the repository, one can classify a file in a number of states. The states, as reported by the `status` command, are:

Up-to-date
: The file is identical with the latest revision in the repository for the branch in use.

Locally Modified
: You have edited the file, and not yet committed your changes.

Locally Added
: You have added the file with `add`, and not yet committed your changes.

Locally Removed
: You have removed the file with `remove`, and not yet committed your changes.

Needs Checkout
: Someone else has committed a newer revision to the repository. The name is slightly misleading; you will ordinarily use `update` rather than `checkout` to get that newer revision.

Needs Patch
: Like Needs Checkout, but the CVS server will send a patch rather than the entire file. Sending a patch or sending an entire file accomplishes the same thing.

Needs Merge
: Someone else has committed a newer revision to the repository, and you have also made modifications to the file.

Unresolved Conflict
: This is like Locally Modified, except that a previous `update` command gave a conflict. You need to resolve the conflict as described in Section 6.3 [Conflicts example], page 28.

Unknown
: CVS doesn't know anything about this file. For example, you have created a new file and have not run `add`.

To help clarify the file status, `status` also reports the `Working revision` which is the revision that the file in the working directory derives from, and the `Repository revision` which is the latest revision in the repository for the branch in use.

For information on the options to `status`, see Section A.16 [status], page 92. For information on its `Sticky tag` and `Sticky date` output, see Section 7.4 [Sticky tags], page 40. For information on its `Sticky options` output, see the '`-k`' option in Section A.18.1 [update options], page 94.

6.2 Bringing a file up to date

When you want to update or merge a file, use the `update` command. For files that are not up to date this is roughly equivalent to a `checkout` command: the newest revision of the file is extracted from the repository and put in your working copy of the module.

Your modifications to a file are never lost when you use `update`. If no newer revision exists, running `update` has no effect. If you have edited the file, and a newer revision is available, CVS will merge all changes into your working copy.

For instance, imagine that you checked out revision 1.4 and started editing it. In the meantime someone else committed revision 1.5, and shortly after that revision 1.6. If you run `update` on the file now, CVS will incorporate all changes between revision 1.4 and 1.6 into your file.

If any of the changes between 1.4 and 1.6 were made too close to any of the changes you have made, an *overlap* occurs. In such cases a warning is printed, and the resulting file includes both versions of the lines that overlap, delimited by special markers. See Section A.18 [update], page 94, for a complete description of the `update` command.

6.3 Conflicts example

Suppose revision 1.4 of '`driver.c`' contains this:

```
#include <stdio.h>

void main()
{
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? 0 : 1);
}
```

Revision 1.6 of '`driver.c`' contains this:

```
#include <stdio.h>

int main(int argc,
         char **argv)
{
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(!!nerr);
}
```

Your working copy of '`driver.c`', based on revision 1.4, contains this before you run '`cvs update`':

```
#include <stdlib.h>
#include <stdio.h>

void main()
{
    init_scanner();
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

You run '`cvs update`':

```
$ cvs update driver.c
RCS file: /usr/local/cvsroot/yoyodyne/tc/driver.c,v
retrieving revision 1.4
retrieving revision 1.6
Merging differences between 1.4 and 1.6 into driver.c
rcsmerge warning: overlaps during merge
cvs update: conflicts found in driver.c
C driver.c
```

CVS tells you that there were some conflicts. Your original working file is saved unmodified in '.#driver.c.1.4'. The new version of 'driver.c' contains this:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
<<<<<<< driver.c
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
=======
    exit(!!nerr);
>>>>>>> 1.6
}
```

Note how all non-overlapping modifications are incorporated in your working copy, and that the overlapping section is clearly marked with '<<<<<<<', '=======' and '>>>>>>>'.

You resolve the conflict by editing the file, removing the markers and the erroneous line. Suppose you end up with this file:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
```

```
            fprintf(stderr, "No code generated.\n");
        exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
    }
```

You can now go ahead and commit this as revision 1.7.

```
$ cvs commit -m "Initialize scanner. Use symbolic exit values." driver.c
Checking in driver.c;
/usr/local/cvsroot/yoyodyne/tc/driver.c,v  <--  driver.c
new revision: 1.7; previous revision: 1.6
done
```

For your protection, CVS will refuse to check in a file if a conflict occurred and you have not resolved the conflict. Currently to resolve a conflict, you must change the timestamp on the file, and must also insure that the file contains no conflict markers. If your file legitimately contains conflict markers (that is, occurrences of '`>>>>>>> `' at the start of a line that don't mark a conflict), then CVS has trouble handling this and you need to start hacking on the `CVS/Entries` file or other such workarounds.

If you use release 1.04 or later of pcl-cvs (a GNU Emacs front-end for CVS) you can use an Emacs package called emerge to help you resolve conflicts. See the documentation for pcl-cvs.

6.4 Informing others about commits

It is often useful to inform others when you commit a new revision of a file. The '`-i`' option of the '`modules`' file, or the '`loginfo`' file, can be used to automate this process. See Section B.1 [modules], page 97. See Section B.6 [loginfo], page 102. You can use these features of CVS to, for instance, instruct CVS to mail a message to all developers, or post a message to a local newsgroup.

6.5 Several developers simultaneously attempting to run CVS

If several developers try to run CVS at the same time, one may get the following message:

```
[11:43:23] waiting for bach's lock in /usr/local/cvsroot/foo
```

CVS will try again every 30 seconds, and either continue with the operation or print the message again, if it still needs to wait. If a lock seems to stick around for an undue amount of time, find the person holding the lock and ask them about the cvs command they are running. If they aren't running a cvs command, look for and remove files starting with '`#cvs.tfl`', '`#cvs.rfl`', or '`#cvs.wfl`' from the repository.

Note that these locks are to protect CVS's internal data structures and have no relationship to the word *lock* in the sense used by RCS—which refers to reserved checkouts (see Chapter 6 [Multiple developers], page 27).

Any number of people can be reading from a given repository at a time; only when someone is writing do the locks prevent other people from reading or writing.

One might hope for the following property

```
If someone commits some changes in one cvs command,
then an update by someone else will either get all the
changes, or none of them.
```

but CVS does *not* have this property. For example, given the files

```
a/one.c
a/two.c
b/three.c
b/four.c
```

if someone runs

```
cvs ci a/two.c b/three.c
```

and someone else runs `cvs update` at the same time, the person running `update` might get only the change to '`b/three.c`' and not the change to '`a/two.c`'.

6.6 Mechanisms to track who is editing files

For many groups, use of CVS in its default mode is perfectly satisfactory. Users may sometimes go to check in a modification only to find that another modification has intervened, but they deal with it and proceed with their check in. Other groups prefer to be able to know who is editing what files, so that if two people try to edit the same file they can choose to talk about who is doing what when rather than be surprised at check in time. The features in this section allow such coordination, while retaining the ability of two developers to edit the same file at the same time.

For maximum benefit developers should use `cvs edit` (not `chmod`) to make files read-write to edit them, and `cvs release` (not `rm`) to discard a working directory which is no longer in use, but CVS is not able to enforce this behavior.

6.6.1 Telling CVS to watch certain files

To enable the watch features, you first specify that certain files are to be watched.

cvs watch on [`-l`] *files* ... Command

Specify that developers should run `cvs edit` before editing *files*. CVS will create working copies of *files* read-only, to remind developers to run the `cvs edit` command before working on them.

If *files* includes the name of a directory, CVS arranges to watch all files added to the corresponding repository directory, and sets a default for files added in the future; this allows the user to set notification policies on a per-directory basis. The contents of the directory are processed recursively, unless the `-l` option is given.

If *files* is omitted, it defaults to the current directory.

cvs watch off [`-l`] *files* ... Command

Do not provide notification about work on *files*. CVS will create working copies of *files* read-write.

The *files* and `-l` arguments are processed as for `cvs watch on`.

6.6.2 Telling CVS to notify you

You can tell CVS that you want to receive notifications about various actions taken on a file. You can do this without using `cvs watch on` for the file, but generally you will want to use `cvs watch on`, so that developers use the `cvs edit` command.

cvs watch add [`-a` *action*] [`-l`] *files* ... Command

Add the current user to the list of people to receive notification of work done on *files*.

The `-a` option specifies what kinds of events CVS should notify the user about. *action* is one of the following:

`edit`	Another user has applied the `cvs edit` command (described below) to a file.
`unedit`	Another user has applied the `cvs unedit` command (described below) or the `cvs release` command to a file, or has deleted the file and allowed `cvs update` to recreate it.
`commit`	Another user has committed changes to a file.
`all`	All of the above.
`none`	None of the above. (This is useful with `cvs edit`, described below.)

The `-a` option may appear more than once, or not at all. If omitted, the action defaults to `all`.

The *files* and `-l` option are processed as for the `cvs watch` commands.

cvs watch remove [`-a` *action*] [`-l`] *files* ... Command

Remove a notification request established using `cvs watch add`; the arguments are the same. If the `-a` option is present, only watches for the specified actions are removed.

When the conditions exist for notification, CVS calls the '`notify`' administrative file. Edit '`notify`' as one edits the other administrative files (see Section 4.3 [Intro administrative files], page 16). This file follows the usual conventions for administrative files (see Section B.3.1 [syntax], page 99), where each line is a regular expression followed by a command to execute. The command should contain a single ocurrence of '`%s`' which will be replaced by the user to notify; the rest of the information regarding the notification will be supplied to the command on standard input. The standard thing to put in the `notify` file is the single line:

```
ALL mail %s -s \"CVS notification\"
```

This causes users to be notified by electronic mail.

Note that if you set this up in the straightforward way, users receive notifications on the server machine. One could of course write a '`notify`' script which directed notifications elsewhere, but to make this easy, CVS allows you to associate a notification address for each user. To do so create a file '`users`' in '`CVSROOT`' with a line for each user in the format *user*:*value*. Then instead of passing the name of the user to be notified to '`notify`', CVS will pass the *value* (normally an email address on some other machine).

6.6.3 How to edit a file which is being watched

Since a file which is being watched is checked out read-only, you cannot simply edit it. To make it read-write, and inform others that you are planning to edit it, use the `cvs edit` command. Some systems call this a *checkout*, but CVS uses that term for obtaining a copy of the sources (see Section 3.1 [Getting the source], page 9), an operation which those systems call a *get* or a *fetch*.

cvs edit [*options*] *files* ... Command

Prepare to edit the working files *files*. CVS makes the *files* read-write, and notifies users who have requested `edit` notification for any of *files*.

The `cvs edit` command accepts the same *options* as the `cvs watch add` command, and establishes a temporary watch for the user on *files*; CVS will remove the watch when *files* are `unedit`ed or `commit`ted. If the user does not wish to receive notifications, she should specify `-a none`.

The *files* and `-l` option are processed as for the `cvs watch` commands.

Normally when you are done with a set of changes, you use the `cvs commit` command, which checks in your changes and returns the watched files to their usual read-only state. But if you instead decide to abandon your changes, or not to make any changes, you can use the `cvs unedit` command.

cvs unedit [`-l`] *files* ... Command

Abandon work on the working files *files*, and revert them to the repository versions on which they are based. CVS makes those *files* read-only for which users have requested notification using `cvs watch on`. CVS notifies users who have requested `unedit` notification for any of *files*.

The *files* and `-l` option are processed as for the `cvs watch` commands.

If watches are not in use, the `unedit` command probably does not work, and the way to revert to the repository version is to remove the file and then use `cvs update` to get a new copy. The meaning is not precisely the same; removing and updating may also bring in some changes which have been made in the repository since the last time you updated.

When using client/server CVS, you can use the `cvs edit` and `cvs unedit` commands even if CVS is unable to succesfully communicate with the server; the notifications will be sent upon the next successful CVS command.

6.6.4 Information about who is watching and editing

cvs watchers [`-l`] *files* ... Command

List the users currently watching changes to *files*. The report includes the files being watched, and the mail address of each watcher.

The *files* and `-l` arguments are processed as for the `cvs watch` commands.

cvs editors [`-l`] *files* ... Command

List the users currently working on *files*. The report includes the mail address of each user, the time when the user began working with the file, and the host and path of the working directory containing the file.

The *files* and `-l` arguments are processed as for the `cvs watch` commands.

6.6.5 Using watches with old versions of CVS

If you use the watch features on a repository, it creates '`CVS`' directories in the repository and stores the information about watches in that directory. If you attempt to use CVS 1.6 or earlier with the repository, you get an error message such as

```
cvs update: cannot open CVS/Entries for reading: No such file or directory
```

and your operation will likely be aborted. To use the watch features, you must upgrade all copies of CVS which use that repository in local or server mode. If you cannot upgrade, use the `watch off` and `watch remove` commands to remove all watches, and that will restore the repository to a state which CVS 1.6 can cope with.

6.7 Choosing between reserved or unreserved checkouts

Reserved and unreserved checkouts each have pros and cons. Let it be said that a lot of this is a matter of opinion or what works given different groups' working styles, but here is an attempt to briefly describe the issues. There are many ways to organize a team of developers. CVS does not try to enforce a certain organization. It is a tool that can be used in several ways.

Reserved checkouts can be very counter-productive. If two persons want to edit different parts of a file, there may be no reason to prevent either of them from doing so. Also, it is common for someone to take out a lock on a file, because they are planning to edit it, but then forget to release the lock.

People, especially people who are familiar with reserved checkouts, often wonder how often conflicts occur if unreserved checkouts are used, and how difficult they are to resolve. The experience with many groups is that they occur rarely and usually are relatively straightforward to resolve.

The rarity of serious conflicts may be surprising, until one realizes that they occur only when two developers disagree on the proper design for a given section of code; such a disagreement suggests that the team has not been communicating properly in the first place. In order to collaborate under *any* source management regimen, developers must agree on the general design of the system; given this agreement, overlapping changes are usually straightforward to merge.

In some cases unreserved checkouts are clearly inappropriate. If no merge tool exists for the kind of file you are managing (for example word processor files or files edited by Computer Aided Design programs), and it is not desirable to change to a program which uses a mergeable data format, then resolving conflicts is going to be unpleasant enough that you generally will be better off to simply avoid the conflicts instead, by using reserved checkouts.

The watches features described above in Section 6.6 [Watches], page 32 can be considered to be an intermediate model between reserved checkouts and unreserved checkouts. When you go to edit a file, it is possible to find out who else is editing it. And rather than having the system simply forbid both people editing the file, it can tell you what the situation is and let you figure out whether it is a problem in that particular case or not. Therefore, for some groups it can be considered the best of both the reserved checkout and unreserved checkout worlds.

7 Branches

So far, all revisions shown in this manual have been on the *main trunk* of the revision tree, i.e., all revision numbers have been of the form *x.y*. One useful feature, especially when maintaining several releases of a software product at once, is the ability to make branches on the revision tree. *Tags*, symbolic names for revisions, will also be introduced in this chapter.

7.1 Tags–Symbolic revisions

The revision numbers live a life of their own. They need not have anything at all to do with the release numbers of your software product. Depending on how you use CVS the revision numbers might change several times between two releases. As an example, some of the source files that make up RCS 5.6 have the following revision numbers:

`ci.c`	`5.21`
`co.c`	`5.9`
`ident.c`	`5.3`
`rcs.c`	`5.12`
`rcsbase.h`	`5.11`
`rcsdiff.c`	`5.10`
`rcsedit.c`	`5.11`
`rcsfcmp.c`	`5.9`
`rcsgen.c`	`5.10`
`rcslex.c`	`5.11`
`rcsmap.c`	`5.2`
`rcsutil.c`	`5.10`

You can use the `tag` command to give a symbolic name to a certain revision of a file. You can use the '`-v`' flag to the `status` command to see all tags that a file has, and which revision numbers they represent. Tag names can contain uppercase and lowercase letters, digits, '`-`', and '`_`'. The two tag names `BASE` and `HEAD` are reserved for use by CVS. It is expected that future names which are special to CVS will contain characters such as '`%`' or '`=`', rather than being named analogously to `BASE` and `HEAD`, to avoid conflicts with actual tag names.

The following example shows how you can add a tag to a file. The commands must be issued inside your working copy of the module. That is, you should issue the command in the directory where '`backend.c`' resides.

```
$ cvs tag release-0-4 backend.c
T backend.c
$ cvs status -v backend.c
===================================================================
File: backend.c         Status: Up-to-date

    Version:            1.4     Tue Dec  1 14:39:01 1992
    RCS Version:        1.4     /usr/local/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         (none)
    Sticky Date:        (none)
    Sticky Options:     (none)
    Existing Tags:
        release-0-4                     (revision: 1.4)
```

There is seldom reason to tag a file in isolation. A more common use is to tag all the files that constitute a module with the same tag at strategic points in the development life-cycle, such as when a release is made.

```
$ cvs tag release-1-0 .
cvs tag: Tagging .
T Makefile
T backend.c
T driver.c
T frontend.c
T parser.c
```

(When you give CVS a directory as argument, it generally applies the operation to all the files in that directory, and (recursively), to any subdirectories that it may contain. See Chapter 9 [Recursive behavior], page 47.)

The `checkout` command has a flag, '`-r`', that lets you check out a certain revision of a module. This flag makes it easy to retrieve the sources that make up release 1.0 of the module '`tc`' at any time in the future:

```
$ cvs checkout -r release-1-0 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

You can also check out a module as it was at any given date. See Section A.6.1 [checkout options], page 77.

When you tag more than one file with the same tag you can think about the tag as "a curve drawn through a matrix of filename vs. revision number." Say we have 5 files with the following revisions:

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

At some time in the past, the `*` versions were tagged. You can think of the tag as a handle attached to the curve drawn through the tagged revisions. When you pull on the handle, you get all the tagged revisions. Another way to look at it is that you "sight" through a set of revisions that is "flat" along the tagged revisions, like this:

```
           file1   file2   file3   file4   file5

                           1.1
                           1.2
                   1.1     1.3                       _
           1.1     1.2     1.4     1.1              /
           1.2*----1.3*----1.5*----1.2*----1.1     (--- <--- Look here
           1.3             1.6     1.3              \_
           1.4                     1.4
                                   1.5
```

7.2 What branches are good for

Suppose that release 1.0 of tc has been made. You are continuing to develop tc, planning to create release 1.1 in a couple of months. After a while your customers start to complain about a fatal bug. You check out release 1.0 (see Section 7.1 [Tags], page 37) and find the bug (which turns out to have a trivial fix). However, the current revision of the sources are in a state of flux and are not expected to be stable for at least another month. There is no way to make a bugfix release based on the newest sources.

The thing to do in a situation like this is to create a *branch* on the revision trees for all the files that make up release 1.0 of tc. You can then make modifications to the branch without disturbing the main trunk. When the modifications are finished you can select to either incorporate them on the main trunk, or leave them on the branch.

7.3 Creating a branch

The `rtag` command can be used to create a branch. The `rtag` command is much like `tag`, but it does not require that you have a working copy of the module. See Section A.15 [rtag], page 91. (You can also use the `tag` command; see Section A.17 [tag], page 93).

```
$ cvs rtag -b -r release-1-0 release-1-0-patches tc
```

The '`-b`' flag makes `rtag` create a branch (rather than just a symbolic revision name). '`-r release-1-0`' says that this branch should be rooted at the node (in the revision tree) that corresponds to the tag '`release-1-0`'. Note that the numeric revision number that matches '`release-1-0`' will probably be different from file to file. The name of the new branch is '`release-1-0-patches`', and the module affected is '`tc`'.

To fix the problem in release 1.0, you need a working copy of the branch you just created.

```
$ cvs checkout -r release-1-0-patches tc
$ cvs status -v driver.c backend.c
===================================================================
File: driver.c          Status: Up-to-date

    Version:            1.7     Sat Dec  5 18:25:54 1992
    RCS Version:        1.7     /usr/local/cvsroot/yoyodyne/tc/driver.c,v
    Sticky Tag:         release-1-0-patches (branch: 1.7.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-1-0-patches             (branch: 1.7.2)
        release-1-0                     (revision: 1.7)

====================================================================
File: backend.c         Status: Up-to-date

    Version:            1.4     Tue Dec  1 14:39:01 1992
    RCS Version:        1.4     /usr/local/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         release-1-0-patches (branch: 1.4.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-1-0-patches             (branch: 1.4.2)
        release-1-0                     (revision: 1.4)
        release-0-4                     (revision: 1.4)
```

As the output from the `status` command shows the branch number is created by adding a digit at the tail of the revision number it is based on. (If '`release-1-0`' corresponds to revision 1.4, the branch's revision number will be 1.4.2. For obscure reasons CVS always gives branches even numbers, starting at 2. See Section 2.1 [Revision numbers], page 7).

7.4 Sticky tags

The '`-r release-1-0-patches`' flag that was given to `checkout` in the previous example is *sticky*, that is, it will apply to subsequent commands in this directory. If you commit any modifications, they are committed on the branch. You can later merge the modifications into the main trunk. See Chapter 8 [Merging], page 43.

You can use the `status` command to see what sticky tags or dates are set:

```
$ vi driver.c   # Fix the bugs
$ cvs commit -m "Fixed initialization bug" driver.c
Checking in driver.c;
/usr/local/cvsroot/yoyodyne/tc/driver.c,v  <--  driver.c
new revision: 1.7.2.1; previous revision: 1.7
done
$ cvs status -v driver.c
====================================================================
File: driver.c          Status: Up-to-date

    Version:            1.7.2.1 Sat Dec  5 19:35:03 1992
    RCS Version:        1.7.2.1 /usr/local/cvsroot/yoyodyne/tc/driver.c,v
    Sticky Tag:         release-1-0-patches (branch: 1.7.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        release-1-0-patches             (branch: 1.7.2)
        release-1-0                     (revision: 1.7)
```

The sticky tags will remain on your working files until you delete them with '`cvs update -A`'. The '`-A`' option retrieves the version of the file from the head of the trunk, and forgets any sticky tags, dates, or options.

Sticky tags are not just for branches. For example, suppose that you want to avoid updating your working directory, to isolate yourself from possibly destabilizing changes other people are making. You can, of course, just refrain from running `cvs update`. But if you want to avoid updating only a portion of a larger tree, then sticky tags can help. If you check out a certain revision (such as 1.4) it will become sticky. Subsequent `cvs update` will not retrieve the latest revision until you reset the tag with `cvs update -A`. Likewise, use of the '`-D`' option to `update` or `checkout` sets a *sticky date*, which, similarly, causes that date to be used for future retrievals.

Many times you will want to retrieve an old version of a file without setting a sticky tag. The way to do that is with the '`-p`' option to `checkout` or `update`, which sends the contents of the file to standard output. For example, suppose you have a file named '`file1`' which existed as revision 1.1, and you then removed it (thus adding a dead revision 1.2). Now suppose you want to add it again, with the same contents it had previously. Here is how to do it:

```
$ cvs update -p -r 1.1 file1 >file1
===================================================================
Checking out file1
RCS:  /tmp/cvs-sanity/cvsroot/first-dir/Attic/file1,v
VERS: 1.1
***************
$ cvs add file1
cvs add: re-adding file file1 (in place of dead revision 1.2)
cvs add: use 'cvs commit' to add this file permanently
$ cvs commit -m test
Checking in file1;
/tmp/cvs-sanity/cvsroot/first-dir/file1,v  <--  file1
new revision: 1.3; previous revision: 1.2
done
$
```

8 Merging

You can include the changes made between any two revisions into your working copy, by *merging*. You can then commit that revision, and thus effectively copy the changes onto another branch.

8.1 Merging an entire branch

You can merge changes made on a branch into your working copy by giving the '`-j` *branch*' flag to the `update` command. With one '`-j` *branch*' option it merges the changes made between the point where the branch forked and newest revision on that branch (into your working copy).

The '`-j`' stands for "join".

Consider this revision tree:

```
+-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+
                !
                !
                !   +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                    +---------+    +---------+
```

The branch 1.2.2 has been given the tag (symbolic name) '`R1fix`'. The following example assumes that the module '`mod`' contains only one file, '`m.c`'.

```
$ cvs checkout mod               # Retrieve the latest revision, 1.4

$ cvs update -j R1fix m.c        # Merge all changes made on the branch,
                                 # i.e. the changes between revision 1.2
                                 # and 1.2.2.2, into your working copy
                                 # of the file.

$ cvs commit -m "Included R1fix" # Create revision 1.5.
```

A conflict can result from a merge operation. If that happens, you should resolve it before committing the new revision. See Section 6.3 [Conflicts example], page 28.

The `checkout` command also supports the '`-j` *branch*' flag. The same effect as above could be achieved with this:

```
$ cvs checkout -j R1fix mod
$ cvs commit -m "Included R1fix"
```

8.2 Merging from a branch several times

Continuing our example, the revision tree now looks like this:

```
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+    +-----+
                !                           *
                !                          *
                !   +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                    +---------+    +---------+
```

where the starred line represents the merge from the 'R1fix' branch to the main trunk, as just discussed.

Now suppose that development continues on the 'R1fix' branch:

```
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 1.2 !----! 1.3 !----! 1.4 !----! 1.5 !      <- The main trunk
+-----+    +-----+    +-----+    +-----+    +-----+
                !                           *
                !                          *
                !   +---------+    +---------+    +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !----! 1.2.2.3 !
                    +---------+    +---------+    +---------+
```

and then you want to merge those new changes onto the main trunk. If you just use the `cvs update -j R1fix m.c` command again, CVS will attempt to merge again the changes which you have already merged, which can have undesirable side effects.

So instead you need to specify that you only want to merge the changes on the branch which have not yet been merged into the trunk. To do that you specify two '`-j`' options, and CVS merges the changes from the first revision to the second revision. For example, in this case the simplest way would be

```
cvs update -j 1.2.2.2 -j R1fix m.c    # Merge changes from 1.2.2.2 to the
                                      # head of the R1fix branch
```

The problem with this is that you need to specify the 1.2.2.2 revision manually. A slightly better approach might be to use the date the last merge was done:

```
cvs update -j R1fix:yesterday -j R1fix m.c
```

Better yet, tag the R1fix branch after every merge into the trunk, and then use that tag for subsequent merges:

```
cvs update -j merged_from_R1fix_to_trunk -j R1fix m.c
```

8.3 Merging differences between any two revisions

With two '`-j` *revision*' flags, the `update` (and `checkout`) command can merge the differences between any two revisions into your working file.

```
$ cvs update -j 1.5 -j 1.3 backend.c
```

will *remove* all changes made between revision 1.3 and 1.5. Note the order of the revisions!

If you try to use this option when operating on multiple files, remember that the numeric revisions will probably be very different between the various files that make up a module. You almost always use symbolic tags rather than revision numbers when operating on multiple files.

8.4 Merging can add or remove files

If the changes which you are merging involve removing or adding some files, `update -j` will reflect such additions or removals.

For example:

```
cvs update -A
touch a b c
cvs add a b c ; cvs ci -m "added" a b c
cvs tag -b branchtag
cvs update -r branchtag
touch d ; cvs add d
rm a ; cvs rm a
cvs ci -m "added d, removed a"
cvs update -A
cvs update -jbranchtag
```

9 Recursive behavior

Almost all of the subcommands of CVS work recursively when you specify a directory as an argument. For instance, consider this directory structure:

```
      $HOME
        |
        +--tc
        |   |
            +--CVS
            |      (internal CVS files)
            +--Makefile
            +--backend.c
            +--driver.c
            +--frontend.c
            +--parser.c
            +--man
            |    |
            |    +--CVS
            |    |  (internal CVS files)
            |    +--tc.1
            |
            +--testing
                 |
                 +--CVS
                 |  (internal CVS files)
                 +--testpgm.t
                 +--test2.t
```

If '`tc`' is the current working directory, the following is true:

- '`cvs update testing`' is equivalent to '`cvs update testing/testpgm.t testing/test2.t`'
- '`cvs update testing man`' updates all files in the subdirectories
- '`cvs update .`' or just '`cvs update`' updates all files in the `tc` module

If no arguments are given to `update` it will update all files in the current working directory and all its subdirectories. In other words, '`.`' is a default argument to `update`. This is also true for most of the CVS subcommands, not only the `update` command.

The recursive behavior of the CVS subcommands can be turned off with the '`-l`' option.

```
  $ cvs update -l         # Don't update files in subdirectories
```

10 Adding files to a directory

To add a new file to a directory, follow these steps.

- You must have a working copy of the directory. See Section 3.1 [Getting the source], page 9.
- Create the new file inside your working copy of the directory.
- Use '`cvs add` *filename*' to tell CVS that you want to version control the file. If the file contains binary data, specify '`-kb`' (see Chapter 17 [Binary files], page 65).
- Use '`cvs commit` *filename*' to actually check in the file into the repository. Other developers cannot see the file until you perform this step.

You can also use the `add` command to add a new directory.

Unlike most other commands, the `add` command is not recursive. You cannot even type '`cvs add foo/bar`'! Instead, you have to

```
$ cd foo
$ cvs add bar
```

cvs add [`-k` *kflag*] [`-m` *message*] *files* . . . Command

Schedule *files* to be added to the repository. The files or directories specified with `add` must already exist in the current directory. To add a whole new directory hierarchy to the source repository (for example, files received from a third-party vendor), use the `import` command instead. See Section A.11 [import], page 85.

The added files are not placed in the source repository until you use `commit` to make the change permanent. Doing an `add` on a file that was removed with the `remove` command will undo the effect of the `remove`, unless a `commit` command intervened. See Chapter 11 [Removing files], page 51, for an example.

The '`-k`' option specifies the default way that this file will be checked out; for more information see Section 16.4 [Substitution modes], page 63.

The '`-m`' option specifies a description for the file. This description appears in the history log (if it is enabled, see Section B.9 [history file], page 104). It will also be saved in the version history inside the repository when the file is committed. The `log` command displays this description. The description can be changed using '`admin -t`'. See Section A.5 [admin], page 73. If you omit the '`-m` *description*' flag, an empty string will be used. You will not be prompted for a description.

For example, the following commands add the file '`backend.c`' to the repository:

```
$ cvs add backend.c
$ cvs commit -m "Early version. Not yet compilable." backend.c
```

When you add a file it is added only on the branch which you are working on (see Chapter 7 [Branches], page 37). You can later merge the additions to another branch if you want (see Section 8.4 [Merging adds and removals], page 45).

11 Removing files from a module

Modules change. New files are added, and old files disappear. Still, you want to be able to retrieve an exact copy of old releases of the module.

Here is what you can do to remove a file from a module, but remain able to retrieve old revisions:

- Make sure that you have not made any uncommitted modifications to the file. See Section 3.4 [Viewing differences], page 10, for one way to do that. You can also use the `status` or `update` command. If you remove the file without committing your changes, you will of course not be able to retrieve the file as it was immediately before you deleted it.
- Remove the file from your working copy of the module. You can for instance use `rm`.
- Use '`cvs remove` *filename*' to tell CVS that you really want to delete the file.
- Use '`cvs commit` *filename*' to actually perform the removal of the file from the repository.

When you commit the removal of the file, CVS records the fact that the file no longer exists. It is possible for a file to exist on only some branches and not on others, or to re-add another file with the same name later. CVS will correctly create or not create the file, based on the '`-r`' and '`-D`' options specified to `checkout` or `update`.

cvs remove [`-lR`] *files* ... Command

Schedule file(s) to be removed from the repository (files which have not already been removed from the working directory are not processed). This command does not actually remove the file from the repository until you commit the removal. The '`-R`' option (the default) specifies that it will recurse into subdirectories; '`-l`' specifies that it will not.

Here is an example of removing several files:

```
$ cd test
$ rm ?.c
$ cvs remove
cvs remove: Removing .
cvs remove: scheduling a.c for removal
cvs remove: scheduling b.c for removal
cvs remove: use 'cvs commit' to remove these files permanently
$ cvs ci -m "Removed unneeded files"
cvs commit: Examining .
cvs commit: Committing .
```

If you change your mind you can easily resurrect the file before you commit it, using the `add` command.

```
$ ls
CVS   ja.h  oj.c
$ rm oj.c
$ cvs remove oj.c
cvs remove: scheduling oj.c for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs add oj.c
U oj.c
cvs add: oj.c, version 1.1.1.1, resurrected
```

If you realize your mistake before you run the `remove` command you can use `update` to resurrect the file:

```
$ rm oj.c
$ cvs update oj.c
cvs update: warning: oj.c was lost
U oj.c
```

When you remove a file it is added only on the branch which you are working on (see Chapter 7 [Branches], page 37). You can later merge the additions to another branch if you want (see Section 8.4 [Merging adds and removals], page 45).

12 Tracking third-party sources

If you modify a program to better fit your site, you probably want to include your modifications when the next release of the program arrives. CVS can help you with this task.

In the terminology used in CVS, the supplier of the program is called a *vendor*. The unmodified distribution from the vendor is checked in on its own branch, the *vendor branch*. CVS reserves branch 1.1.1 for this use.

When you modify the source and commit it, your revision will end up on the main trunk. When a new release is made by the vendor, you commit it on the vendor branch and copy the modifications onto the main trunk.

Use the `import` command to create and update the vendor branch. After a successful `import` the vendor branch is made the 'head' revision, so anyone that checks out a copy of the file gets that revision. When a local modification is committed it is placed on the main trunk, and made the 'head' revision.

12.1 Importing a module for the first time

Use the `import` command to check in the sources for the first time. When you use the `import` command to track third-party sources, the *vendor tag* and *release tags* are useful. The *vendor tag* is a symbolic name for the branch (which is always 1.1.1, unless you use the '`-b` *branch*' flag—See Section A.11.1 [import options], page 86). The *release tags* are symbolic names for a particular release, such as '`FSF_0_04`'.

Suppose you use `wdiff` (a variant of `diff` that ignores changes that only involve whitespace), and are going to make private modifications that you want to be able to use even when new releases are made in the future. You start by importing the source to your repository:

```
$ tar xfz wdiff-0.04.tar.gz
$ cd wdiff-0.04
$ cvs import -m "Import of FSF v. 0.04" fsf/wdiff FSF_DIST WDIFF_0_04
```

The vendor tag is named '`FSF_DIST`' in the above example, and the only release tag assigned is '`WDIFF_0_04`'.

12.2 Updating a module with the import command

When a new release of the source arrives, you import it into the repository with the same `import` command that you used to set up the repository in the first place. The only difference is that you specify a different release tag this time.

```
$ tar xfz wdiff-0.05.tar.gz
$ cd wdiff-0.05
$ cvs import -m "Import of FSF v. 0.05" fsf/wdiff FSF_DIST WDIFF_0_05
```

For files that have not been modified locally, the newly created revision becomes the head revision. If you have made local changes, `import` will warn you that you must merge the changes into the main trunk, and tell you to use '`checkout -j`' to do so.

```
$ cvs checkout -jFSF_DIST:yesterday -jFSF_DIST wdiff
```

The above command will check out the latest revision of '`wdiff`', merging the changes made on the vendor branch '`FSF_DIST`' since yesterday into the working copy. If any conflicts arise during the merge they should be resolved in the normal way (see Section 6.3 [Conflicts example], page 28). Then, the modified files may be committed.

Using a date, as suggested above, assumes that you do not import more than one release of a product per day. If you do, you can always use something like this instead:

```
$ cvs checkout -jWDIFF_0_04 -jWDIFF_0_05 wdiff
```

In this case, the two above commands are equivalent.

12.3 How to handle binary files with cvs import

Use the '`-k`' wrapper option to tell import which files are binary. See Section B.2 [Wrappers], page 98.

13 Moving and renaming files

Moving files to a different directory or renaming them is not difficult, but some of the ways in which this works may be non-obvious. (Moving or renaming a directory is even harder. See Chapter 14 [Moving directories], page 57).

The examples below assume that the file *old* is renamed to *new*.

13.1 The Normal way to Rename

The normal way to move a file is to copy *old* to *new*, and then issue the normal CVS commands to remove *old* from the repository, and add *new* to it. (Both *old* and *new* could contain relative paths, for example '`foo/bar.c`').

```
$ mv old new
$ cvs remove old
$ cvs add new
$ cvs commit -m "Renamed old to new" old new
```

This is the simplest way to move a file, it is not error-prone, and it preserves the history of what was done. Note that to access the history of the file you must specify the old or the new name, depending on what portion of the history you are accessing. For example, `cvs log` *old* will give the log up until the time of the rename.

When *new* is committed its revision numbers will start at 1.0 again, so if that bothers you, use the '`-r rev`' option to commit (see Section A.7.1 [commit options], page 79)

13.2 Moving the history file

This method is more dangerous, since it involves moving files inside the repository. Read this entire section before trying it out!

```
$ cd $CVSROOT/module
$ mv old,v new,v
```

Advantages:

- The log of changes is maintained intact.
- The revision numbers are not affected.

Disadvantages:

- Old releases of the module cannot easily be fetched from the repository. (The file will show up as *new* even in revisions from the time before it was renamed).
- There is no log information of when the file was renamed.

- Nasty things might happen if someone accesses the history file while you are moving it. Make sure no one else runs any of the CVS commands while you move it.

13.3 Copying the history file

This way also involves direct modifications to the repository. It is safe, but not without drawbacks.

```
# Copy the RCS file inside the repository
$ cd $CVSROOT/module
$ cp old,v new,v
# Remove the old file
$ cd ~/module
$ rm old
$ cvs remove old
$ cvs commit old
# Remove all tags from new
$ cvs update new
$ cvs log new                # Remember the non-branch tag names
$ cvs tag -d tag1 new
$ cvs tag -d tag2 new
...
```

By removing the tags you will be able to check out old revisions of the module.

Advantages:

- Checking out old revisions works correctly, as long as you use '`-r`*tag*' and not '`-D`*date*' to retrieve the revisions.
- The log of changes is maintained intact.
- The revision numbers are not affected.

Disadvantages:

- You cannot easily see the history of the file across the rename.
- Unless you use the '`-r rev`' (see Section A.7.1 [commit options], page 79) flag when *new* is committed its revision numbers will start at 1.0 again.

14 Moving and renaming directories

If you want to be able to retrieve old versions of the module, you must move each file in the directory with the CVS commands. See Section 13.1 [Outside], page 55. The old, empty directory will remain inside the repository, but it will not appear in your workspace when you check out the module in the future.

If you really want to rename or delete a directory, you can do it like this:

1. Inform everyone who has a copy of the module that the directory will be renamed. They should commit all their changes, and remove their working copies of the module, before you take the steps below.
2. Rename the directory inside the repository.
   ```
   $ cd $CVSROOT/module
   $ mv old-dir new-dir
   ```
3. Fix the CVS administrative files, if necessary (for instance if you renamed an entire module).
4. Tell everyone that they can check out the module and continue working.

If someone had a working copy of the module the CVS commands will cease to work for him, until he removes the directory that disappeared inside the repository.

It is almost always better to move the files in the directory instead of moving the directory. If you move the directory you are unlikely to be able to retrieve old releases correctly, since they probably depend on the name of the directories.

15 History browsing

Once you have used CVS to store a version control history—what files have changed when, how, and by whom, there are a variety of mechanisms for looking through the history.

15.1 Log messages

Whenever you commit a file you specify a log message.

To look through the log messages which have been specified for every revision which has been committed, use the `cvs log` command (see Section A.12 [log], page 87).

15.2 The history database

You can use the history file (see Section B.9 [history file], page 104) to log various CVS actions. To retrieve the information from the history file, use the `cvs history` command (see Section A.10 [history], page 84).

15.3 User-defined logging

You can customize CVS to log various kinds of actions, in whatever manner you choose. These mechanisms operate by executing a script at various times. The script might append a message to a file listing the information and the programmer who created it, or send mail to a group of developers, or, perhaps, post a message to a particular newsgroup. To log commits, use the '`loginfo`' file (see Section B.6 [loginfo], page 102). To log commits, checkouts, exports, and tags, respectively, you can also use the '`-i`', '`-o`', '`-e`', and '`-t`' options in the modules file. For a more flexible way of giving notifications to various users, which requires less in the way of keeping centralized scripts up to date, use the `cvs watch add` command (see Section 6.6.2 [Getting Notified], page 33); this command is useful even if you are not using `cvs watch on`.

The '`taginfo`' file defines programs to execute when someone executes a `tag` or `rtag` command. The '`taginfo`' file has the standard form for administrative files (see Appendix B [Administrative files], page 97), where each line is a regular expression followed by a command to execute. The arguments passed to the command are, in order, the *tagname*, *operation* (`add` for `tag`, `mov` for `tag -F`, and `del` for `tag -d`), *repository*, and any remaining are pairs of *filename revision*. A non-zero exit of the filter program will cause the tag to be aborted.

15.4 Annotate command

cvs annotate [`-lf`] [`-r rev|-D date`] *files* ... Command

For each file in *files*, print the head revision of the trunk, together with information on the last modification for each line. For example:

```
$ cvs annotate ssfile
```

```
        Annotations for ssfile
        ***************
        1.1          (mary     27-Mar-96): ssfile line 1
        1.2          (joe      28-Mar-96): ssfile line 2
```

The file '`ssfile`' currently contains two lines. The `ssfile line 1` line was checked in by `mary` on March 27. Then, on March 28, `joe` added a line `ssfile line 2`, without modifying the `ssfile line 1` line. This report doesn't tell you anything about lines which have been deleted or replaced; you need to use `cvs diff` for that (see Section A.8 [diff], page 81).

These standard options are available with `annotate` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date*
: Annotate the most recent revision no later than *date*.

`-f`
: Only useful with the '`-D` *date*' or '`-r` *tag*' flags. If no matching revision is found, annotate the most recent revision (instead of ignoring the file).

`-l`
: Local; run only in current working directory. See Chapter 9 [Recursive behavior], page 47.

`-r` *tag*
: Annotate revision *tag*.

16 Keyword substitution

As long as you edit source files inside your working copy of a module you can always find out the state of your files via '`cvs status`' and '`cvs log`'. But as soon as you export the files from your development environment it becomes harder to identify which revisions they are.

RCS uses a mechanism known as *keyword substitution* (or *keyword expansion*) to help identifying the files. Embedded strings of the form `$`*keyword*`$` and `$`*keyword*`:...$` in a file are replaced with strings of the form `$`*keyword*`:`*value*`$` whenever you obtain a new revision of the file.

16.1 RCS Keywords

This is a list of the keywords that RCS currently (in release 5.6.0.1) supports:

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

`$Date$` The date and time (UTC) the revision was checked in.

`$Header$` A standard header containing the full pathname of the RCS file, the revision number, the date (UTC), the author, the state, and the locker (if locked). Files will normally never be locked when you use CVS.

`$Id$` Same as `$Header$`, except that the RCS filename is without a path.

`$Name$` Tag name used to check out this file.

`$Locker$` The login name of the user who locked the revision (empty if not locked, and thus almost always useless when you are using CVS).

`$Log$` The log message supplied during commit, preceded by a header containing the RCS filename, the revision number, the author, and the date (UTC). Existing log messages are *not* replaced. Instead, the new log message is inserted after `$Log:...$`. Each new line is prefixed with a *comment leader* which RCS guesses from the file name extension. It can be changed with `cvs admin -c`. See Section A.5.1 [admin options], page 74. This keyword is useful for accumulating a complete change log in a source file, but for several reasons it can be problematic. See Section 16.5 [Log keyword], page 63.

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

`$Revision$`
The revision number assigned to the revision.

`$Source$` The full pathname of the RCS file.

`$State$` The state assigned to the revision. States can be assigned with `cvs admin -s`—See Section A.5.1 [admin options], page 74.

16.2 Using keywords

To include a keyword string you simply include the relevant text string, such as `$Id$`, inside the file, and commit the file. CVS will automatically expand the string as part of the commit operation.

It is common to embed `$Id$` string in the C source code. This example shows the first few lines of a typical file, after keyword substitution has been performed:

```
static char *rcsid="$Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $";
/* The following lines will prevent gcc version 2.x
   from issuing an "unused variable" warning. */
#if __GNUC__ == 2
#define USE(var) static void * use_##var = (&use_##var, (void *) &var)
USE (rcsid);
#endif
```

Even though a clever optimizing compiler could remove the unused variable `rcsid`, most compilers tend to include the string in the binary. Some compilers have a `#pragma` directive to include literal text in the binary.

The `ident` command (which is part of the RCS package) can be used to extract keywords and their values from a file. This can be handy for text files, but it is even more useful for extracting keywords from binary files.

```
$ ident samp.c
samp.c:
     $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
$ gcc samp.c
$ ident a.out
a.out:
     $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
```

SCCS is another popular revision control system. It has a command, `what`, which is very similar to `ident` and used for the same purpose. Many sites without RCS have SCCS. Since `what` looks for the character sequence `@(#)` it is easy to include keywords that are detected by either command. Simply prefix the RCS keyword with the magic SCCS phrase, like this:

```
static char *id="@(#) $Id: ab.c,v 1.5 1993/10/19 14:57:32 ceder Exp $";
```

16.3 Avoiding substitution

Keyword substitution has its disadvantages. Sometimes you might want the literal text string '`$Author$`' to appear inside a file without RCS interpreting it as a keyword and expanding it into something like '`$Author: ceder $`'.

There is unfortunately no way to selectively turn off keyword substitution. You can use '`-ko`' (see Section 16.4 [Substitution modes], page 63) to turn off keyword substitution entirely.

In many cases you can avoid using RCS keywords in the source, even though they appear in the final product. For example, the source for this manual contains '`$@asis{}Author$`' whenever the text '`$Author$`' should appear. In `nroff` and `troff` you can embed the null-character `\&` inside the keyword for a similar effect.

16.4 Substitution modes

Each file has a stored default substitution mode, and each working directory copy of a file also has a substitution mode. The former is set by the '`-k`' option to `cvs add` and `cvs admin`; the latter is set by the -k or -A options to `cvs checkout` or `cvs update`. `cvs diff` also has a '`-k`' option. For some examples, See Chapter 17 [Binary files], page 65.

The modes available are:

'`-kkv`' Generate keyword strings using the default form, e.g. `$Revision: 5.7 $` for the `Revision` keyword.

'`-kkvl`' Like '`-kkv`', except that a locker's name is always inserted if the given revision is currently locked. This option is normally not useful when CVS is used.

'`-kk`' Generate only keyword names in keyword strings; omit their values. For example, for the `Revision` keyword, generate the string `$Revision$` instead of `$Revision: 5.7 $`. This option is useful to ignore differences due to keyword substitution when comparing different revisions of a file.

'`-ko`' Generate the old keyword string, present in the working file just before it was checked in. For example, for the `Revision` keyword, generate the string `$Revision: 1.1 $` instead of `$Revision: 5.7 $` if that is how the string appeared when the file was checked in.

'`-kb`' Like '`-ko`', but also inhibit conversion of line endings between the canonical form in which they are stored in the repository (linefeed only), and the form appropriate to the operating system in use on the client. For systems, like unix, which use linefeed only to terminate lines, this is the same as '`-ko`'. For more information on binary files, see Chapter 17 [Binary files], page 65.

'`-kv`' Generate only keyword values for keyword strings. For example, for the `Revision` keyword, generate the string `5.7` instead of `$Revision: 5.7 $`. This can help generate files in programming languages where it is hard to strip keyword delimiters like `$Revision: $` from a string. However, further keyword substitution cannot be performed once the keyword names are removed, so this option should be used with care.

One often would like to use '`-kv`' with `cvs export`—see Section A.9 [export], page 83. But be aware that doesn't handle an export containing binary files correctly.

16.5 Problems with the Log keyword.

The `$Log$` keyword is somewhat controversial. As long as you are working on your development system the information is easily accessible even if you do not use the `$Log$` keyword—just do a `cvs log`. Once you export the file the history information might be useless anyhow.

A more serious concern is that RCS is not good at handling `$Log$` entries when a branch is merged onto the main trunk. Conflicts often result from the merging operation.

People also tend to "fix" the log entries in the file (correcting spelling mistakes and maybe even factual errors). If that is done the information from `cvs log` will not be consistent with the information inside the file. This may or may not be a problem in real life.

It has been suggested that the `$Log$` keyword should be inserted *last* in the file, and not in the files header, if it is to be used at all. That way the long list of change messages will not interfere with everyday source file browsing.

17 Handling binary files

There are two issues with using CVS to store binary files. The first is that CVS by default convert line endings between the canonical form in which they are stored in the repository (linefeed only), and the form appropriate to the operating system in use on the client (for example, carriage return followed by line feed for Windows NT).

The second is that a binary file might happen to contain data which looks like a keyword (see Chapter 16 [Keyword substitution], page 61), so keyword expansion must be turned off.

The '`-kb`' option available with some CVS commands insures that neither line ending conversion nor keyword expansion will be done. If you are using an old version of RCS without this option, and you are using an operating system, such as unix, which terminates lines with linefeeds only, you can use '`-ko`' instead; if you are on another operating system, upgrade to a version of RCS, such as 5.7 or later, which supports '`-kb`'.

Here is an example of how you can create a new file using the '`-kb`' flag:

```
$ echo '$Id$' > kotest
$ cvs add -kb -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
```

If a file accidentally gets added without '`-kb`', one can use the `cvs admin` command to recover. For example:

```
$ echo '$Id$' > kotest
$ cvs add -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
$ cvs admin -kb kotest
$ cvs update -A kotest
$ cvs commit -m "make it binary" kotest  # For non-unix systems
```

When you check in the file '`kotest`' the keywords are expanded. (Try the above example, and do a `cat kotest` after every command). The `cvs admin -kb` command sets the default keyword substitution method for this file, but it does not alter the working copy of the file that you have. The easiest way to get the unexpanded version of '`kotest`' is `cvs update -A`. If you need to cope with line endings (that is, you are using a CVS client on a non-unix system), then you need to check in a new copy of the file, as shown by the `cvs commit` command above.

However, in using `cvs admin -k` to change the keyword expansion, be aware that the keyword expansion mode is not version controlled. This means that, for example, that if you have a text file in old releases, and a binary file with the same name in new releases, CVS provides no way to check out the file in text or binary mode depending on what version you are checking out. There is no good workaround for this problem.

You can also set a default for whether `cvs add` and `cvs import` treat a file as binary based on its name; for example you could say that files who names end in '`.exe`' are binary. See Section B.2 [Wrappers], page 98.

18 Revision management

If you have read this far, you probably have a pretty good grasp on what CVS can do for you. This chapter talks a little about things that you still have to decide.

If you are doing development on your own using CVS you could probably skip this chapter. The questions this chapter takes up become more important when more than one person is working in a repository.

18.1 When to commit?

Your group should decide which policy to use regarding commits. Several policies are possible, and as your experience with CVS grows you will probably find out what works for you.

If you commit files too quickly you might commit files that do not even compile. If your partner updates his working sources to include your buggy file, he will be unable to compile the code. On the other hand, other persons will not be able to benefit from the improvements you make to the code if you commit very seldom, and conflicts will probably be more common.

It is common to only commit files after making sure that they can be compiled. Some sites require that the files pass a test suite. Policies like this can be enforced using the commitinfo file (see Section B.4 [commitinfo], page 100), but you should think twice before you enforce such a convention. By making the development environment too controlled it might become too regimented and thus counter-productive to the real goal, which is to get software written.

Appendix A Reference manual for CVS commands

This appendix describes how to invoke CVS, and describes in detail those subcommands of CVS which are not fully described elsewhere. To look up a particular subcommand, see [Index], page 113.

A.1 Overall structure of CVS commands

The overall format of all CVS commands is:

```
    cvs [ cvs_options ] cvs_command [ command_options ] [ command_args ]
```

`cvs` The name of the CVS program.

`cvs_options`
: Some options that affect all sub-commands of CVS. These are described below.

`cvs_command`
: One of several different sub-commands. Some of the commands have aliases that can be used instead; those aliases are noted in the reference manual for that 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.

`command_options`
: Options that are specific for the command.

`command_args`
: Arguments to the commands.

There is unfortunately some confusion between `cvs_options` and `command_options`. '`-l`', when given as a `cvs_option`, only affects some of the commands. When it is given as a `command_option` is has a different meaning, and is accepted by more commands. In other words, do not take the above categorization too seriously. Look at the documentation instead.

A.2 Default options and the ~/.cvsrc file

There are some `command_options` that are used so often that you might have set up an alias or some other means to make sure you always specify that option. One example (the one that drove the implementation of the .cvsrc support, actually) is that many people find the default output of the '`diff`' command to be very hard to read, and that either context diffs or unidiffs are much easier to understand.

The '`~/.cvsrc`' file is a way that you can add default options to `cvs_commands` within cvs, instead of relying on aliases or other shell scripts.

The format of the '`~/.cvsrc`' file is simple. The file is searched for a line that begins with the same name as the `cvs_command` being executed. If a match is found, then the remainder of the line is split up (at whitespace characters) into separate options and added to the command arguments *before* any options from the command line.

If a command has two names (e.g., `checkout` and `co`), the official name, not necessarily the one used on the command line, will be used to match against the file. So if this is the contents of the user's '`~/.cvsrc`' file:

```
log -N
diff -u
update -P
co -P
```

the command '`cvs checkout foo`' would have the '`-P`' option added to the arguments, as well as '`cvs co foo`'.

With the example file above, the output from '`cvs diff foobar`' will be in unidiff format. '`cvs diff -c foobar`' will provide context diffs, as usual. Getting "old" format diffs would be slightly more complicated, because `diff` doesn't have an option to specify use of the "old" format, so you would need '`cvs -f diff foobar`'.

In place of the command name you can use `cvs` to specify global options (see Section A.3 [Global options], page 70). For example the following line in '`.cvsrc`'

```
cvs -z6
```

causes CVS to use compression level 6

A.3 Global options

The available '`cvs_options`' (that are given to the left of '`cvs_command`') are:

`-b` *bindir*
: Use *bindir* as the directory where RCS programs are located. Overrides the setting of the `$RCSBIN` environment variable and any precompiled directory. This parameter should be specified as an absolute pathname.

`-T` *tempdir*
: Use *tempdir* as the directory where temporary files are located. Overrides the setting of the `$TMPDIR` environment variable and any precompiled directory. This parameter should be specified as an absolute pathname.

`-d` *cvs˙root˙directory*
: Use *cvs_root_directory* as the root directory pathname of the repository. Overrides the setting of the `$CVSROOT` environment variable. See Chapter 4 [Repository], page 13.

`-e` *editor*
: Use *editor* to enter revision log information. Overrides the setting of the `$CVSEDITOR` and `$EDITOR` environment variables.

`-f`
: Do not read the '`~/.cvsrc`' file. This option is most often used because of the non-orthogonality of the CVS option set. For example, the '`cvs log`' option '`-N`' (turn off display of tag names) does not have a corresponding option to turn the display on. So if you have '`-N`' in the '`~/.cvsrc`' entry for '`log`', you may need to use '`-f`' to show the tag names.

`-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.

`-l`
: Do not log the cvs_command in the command history (but execute it anyway). See Section A.10 [history], page 84, 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.

`-Q`
: Cause the command to be really quiet; the command will only generate output for serious problems.

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

`-r`
: Make new working files files read-only. Same effect as if the `$CVSREAD` environment variable is set (see Appendix C [Environment variables], page 107). The default is to make working files writable, unless watches are on (see Section 6.6 [Watches], page 32).

`-s` *variable*=*value*
: Set a user variable (see Section B.10 [Variables], page 104).

`-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.

`-v`
: Display version and copyright information for CVS.

`-w`
: Make new working files read-write. Overrides the setting of the `$CVSREAD` environment variable. Files are created read-write by default, unless `$CVSREAD` is set or '`-r`' is given.

`-x`
: Encrypt all communication between the client and the server. Only has an effect on the CVS client. As of this writing, this is only implemented when using a Kerberos connection (see Section 4.6.3 [Kerberos authenticated], page 21). Encryption support is not available by default; it must be enabled using a special configure option, '`--enable-encryption`', when you build CVS.

`-z` *gzip-level*
: Set the compression level. Only has an effect on the CVS client.

A.4 Common command options

This section describes the '`command_options`' that are available across several CVS commands. These options are always given to the right of '`cvs_command`'. 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 almost always count on the same behavior of the option as in other commands. (Other command options, which are listed with the individual commands, may have different behavior from one CVS command to the other).

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*. *date_spec* is a single argument, a date description specifying a date in the past.

: 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 (for more information on sticky tags/dates, see Section 7.4 [Sticky tags], page 40).

A wide variety of date formats are supported by CVS. The *date_spec* is interpreted as being in the local timezone, unless a specific timezone is specified. 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
```

'`-D`' is available with the `checkout`, `diff`, `export`, `history`, `rdiff`, `rtag`, and `update` commands. (The `history` command uses this option in a slightly different way; see Section A.10.1 [history options], page 84). Note that when specifying a date like '`3/31/92`' it is *month*/*day*/*year*. So '`1/4/96`' is January 4, not March 1.

Remember to quote the argument to the '`-D`' flag so that your shell doesn't interpret spaces as argument separators. A command using the '`-D`' flag can look like this:

```
$ cvs diff -D "1 hour ago" cvs.texinfo
```

`-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 prior to 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 revision of the file will be used).

'`-f`' is available with these commands: `checkout`, `export`, `rdiff`, `rtag`, and `update`.

Warning: The `commit` command also has a '`-f`' option, but it has a different behavior for that command. See Section A.7.1 [commit options], page 79.

`-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. See Chapter 16 [Keyword substitution], page 61, for the meaning of *kflag*. 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 associates your selected *kflag* with the file, and continues to use it with future update commands on the same file until you specify otherwise.

The '`-k`' option is available with the `add`, `checkout`, `diff` and `update` commands.

`-l` Local; run only in current working directory, rather than recursing through subdirectories.

Warning: this is not the same as the overall '`cvs -l`' option, which you can specify to the left of a cvs command!

Available with the following commands: `checkout`, `commit`, `diff`, `export`, `log`, `remove`, `rdiff`, `rtag`, `status`, `tag`, and `update`.

`-m` *message*

Use *message* as log information, instead of invoking an editor.

Available with the following commands: `add`, `commit` and `import`.

`-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 (see Section B.1 [modules], page 97); this option bypasses it).

Warning: this is not the same as the overall '`cvs -n`' option, which you can specify to the left of a cvs command!

Available with the `checkout`, `commit`, `export`, and `rtag` commands.

`-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.

`-W`
: Specify file names that should be filtered. You can use this option repeatedly. The spec can be a file name pattern of the same type that you can specify in the '`.cvswrappers`' file. Avaliable with the following commands: `import`, 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 the current working directory.

The tag specification is sticky when you use this with `checkout` or `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 (for more information on sticky tags/dates, see Section 7.4 [Sticky tags], page 40). The tag can be either a symbolic or numeric tag. See Section 7.1 [Tags], page 37.

Specifying the '`-q`' global option along with the '`-r`' command option is often useful, to suppress the warning messages when the RCS history file does not contain the specified tag.

Warning: this is not the same as the overall 'cvs -r' option, which you can specify to the left of a cvs command!

'`-r`' is available with the `checkout`, `commit`, `diff`, `history`, `export`, `rdiff`, `rtag`, and `update` commands.

A.5 admin—Administration front end for rcs

- Requires: repository, working directory.
- Changes: repository.
- Synonym: rcs

This is the CVS interface to assorted administrative RCS facilities, documented in rcs(1). `admin` 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.

If there is a group whose name matches a compiled in value which defaults to `cvsadmin`, only members of that group can use `cvs admin`. To disallow `cvs admin` for all users, create a group with no users in it.

A.5.1 admin options

Not all valid `rcs` options are useful together with CVS. Some even makes it impossible to use CVS until you undo the effect!

This description of the available options is based on the '`rcs(1)`' man page, but modified to suit readers that are more interrested in CVS than RCS.

`-A`*oldfile*
: Might not work together with CVS. Append the access list of *oldfile* to the access list of the RCS file.

`-a`*logins*
: Might not work together with CVS. Append the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

`-b[`*rev*`]`
: When used with bare RCS, this option sets the default branch to *rev*; in CVS sticky tags (see Section 7.4 [Sticky tags], page 40) are a better way to decide which branch you want to work on. With CVS, this option can be used to control behavior with respect to the vendor branch.

`-c`*string*
: Useful with CVS. Sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword `$Log$` (see Chapter 16 [Keyword substitution], page 61). This is useful for programming languages without multi-line comments. RCS initially guesses the value of the comment leader from the file name extension when the file is first committed.

`-e[`*logins*`]`
: Might not work together with CVS. Erase the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, erase the entire access list.

`-I`
: Run interactively, even if the standard input is not a terminal.

`-i`
: Useless with CVS. When using bare RCS, this is used to create and initialize a new RCS file, without depositing a revision.

`-k`*subst*
: Useful with CVS. Set the default keyword substitution to *subst*. See Chapter 16 [Keyword substitution], page 61. Giving an explicit '`-k`' option to `cvs update`, `cvs export`, or `cvs checkout` overrides this default.

`-l[`*rev*`]`
: Lock the revision with number *rev*. If a branch is given, lock the latest revision on that branch. If *rev* is omitted, lock the latest revision on the default branch.

: This can be used in conjunction with the '`rcslock.pl`' script in the '`contrib`' directory of the CVS source distribution to provide reserved checkouts (where only one user can be editing a given file at a time). See the comments in that file for details (and see the '`README`' file in that directory for disclaimers about the unsupported nature of contrib). According to comments in that file, locking must set to strict (which is the default).

`-L`
: Set locking to strict. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. For use with CVS, strict locking must be set; see the discussion under the '`-l`' option above.

`-m`*rev*`:`*msg*
: Replace the log message of revision *rev* with *msg*.

`-N`*name*`[:[`*rev*`]]`
: Act like '`-n`', except override any previous assignment of *name*.

`-n`*name*`[:[`*rev*`]]`
: Associate the symbolic name *name* with the branch or revision *rev*. It is normally better to use '`cvs tag`' or '`cvs rtag`' instead. Delete the symbolic name if both '`:`'

and *rev* are omitted; otherwise, print an error message if *name* is already associated with another number. If *rev* is symbolic, it is expanded before association. A *rev* consisting of a branch number followed by a '`.`' stands for the current latest revision in the branch. A '`:`' with an empty *rev* stands for the current latest revision on the default branch, normally the trunk. For example, '`rcs -n`*name*`: RCS/*`' associates *name* with the current latest revision of all the named RCS files; this contrasts with '`rcs -n`*name*`:$ RCS/*`' which associates *name* with the revision numbers extracted from keyword strings in the corresponding working files.

`-o`*range*
: Potentially useful, but dangerous, with CVS (see below). 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.

  Due to the way CVS handles branches *rev* cannot be specified symbolically if it is a branch. See Section D.1 [Magic branch numbers], page 109, for an explanation.

  Make sure that no-one has checked out a copy of the revision you outdate. Strange things will happen if he starts to edit it and tries to check it back in. For this reason, this option is not a good way to take back a bogus commit; commit a new revision undoing the bogus change instead (see Section 8.3 [Merging two revisions], page 45).

`-q`
: Run quietly; do not print diagnostics.

`-s`*state*`[:`*rev*`]`
: Useful with CVS. Set the state attribute of the revision *rev* to *state*. If *rev* is a branch number, assume the latest revision on that branch. If *rev* is omitted, assume the latest revision on the default branch. Any identifier is acceptable for *state*. A useful set of states is '`Exp`' (for experimental), '`Stab`' (for stable), and '`Rel`' (for released). By default, the state of a new revision is set to '`Exp`' when it is created. The state is visible in the output from *cvs log* (see Section A.12 [log], page 87), and in the '`$Log$`' and '`$State$`' keywords (see Chapter 16 [Keyword substitution], page 61). Note that CVS uses the `dead` state for its own purposes; to take a file to or from the `dead` state use commands like `cvs remove` and `cvs add`, not `cvs admin -s`.

`-t[`*file*`]`
: Useful with CVS. Write descriptive text from the contents of the named *file* into the RCS file, deleting the existing text. The *file* pathname may not begin with '`-`'. If *file* is omitted, obtain the text from standard input, terminated by end-of-file or by a line containing '`.`' by itself. Prompt for the text if interaction is possible; see '`-I`'. The descriptive text can be seen in the output from '`cvs log`' (see Section A.12 [log], page 87).

`-t-`*string*
: Similar to '`-t`*file*'. Write descriptive text from the *string* into the RCS file, deleting the existing text.

`-U`
: Set locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. For use with CVS, strict locking must be set; see the discussion under the '`-l`' option above.

`-u[`*rev*`]`
: See the option '`-l`' above, for a discussion of using this option with CVS. Unlock the revision with number *rev*. If a branch is given, unlock the latest revision on that branch. If *rev* is omitted, remove the latest lock held by the caller. 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 by end-of-file or by a line containing `.` by itself.

`-V`*n* Emulate RCS version *n*. Use -V*n* to make an RCS file acceptable to RCS version *n* by discarding information that would confuse version *n*.

`-x`*suffixes* Useless with CVS. Use *suffixes* to characterize RCS files.

A.5.2 admin examples

A.5.2.1 Outdating is dangerous

First, an example of how *not* to use the `admin` command. It is included to stress the fact that this command can be quite dangerous unless you know *exactly* what you are doing.

The '`-o`' option can be used to *outdate* old revisions from the history file. If you are short on disc this option might help you. But think twice before using it—there is no way short of restoring the latest backup to undo this command!

The next line is an example of a command that you would *not* like to execute.

```
$ cvs admin -o:R_1_02 .
```

The above command will delete all revisions up to, and including, the revision that corresponds to the tag R_1_02. But beware! If there are files that have not changed between R_1_02 and R_1_03 the file will have *the same* numerical revision number assigned to the tags R_1_02 and R_1_03. So not only will it be impossible to retrieve R_1_02; R_1_03 will also have to be restored from the tapes!

A.5.2.2 Comment leaders

If you use the `$Log$` keyword and you do not agree with the guess for comment leader that CVS has done, you can enforce your will with `cvs admin -c`. This might be suitable for `nroff` source:

```
$ cvs admin -c'.\" ' *.man
$ rm *.man
$ cvs update
```

The two last steps are to make sure that you get the versions with correct comment leaders in your working files.

A.6 checkout—Check out sources for editing

- Synopsis: 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 `checkout` before using most of the other CVS commands, since most of them operate on your working directory.

The *modules* part of the command are either symbolic names for some collection of source directories and files, or paths to directories or files in the repository. The symbolic names are defined in the '`modules`' file. See Section B.1 [modules], page 97.

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 source repository.

Note that `checkout` is used to create directories. The top-level 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 sub-directory may have a different name, but you can be sure that it will be a sub-directory, 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`' global option).

The files created by `checkout` are created read-write, unless the '`-r`' option to CVS (see Section A.3 [Global options], page 70) is specified, the `CVSREAD` environment variable is specified (see Appendix C [Environment variables], page 107), or a watch is in effect for that file (see Section 6.6 [Watches], page 32).

Running `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, that is, any new directories that have been created in the repository will appear in your work area. See Section A.18 [update], page 94.

For the output produced by the `checkout` command see Section A.18.2 [update output], page 95.

A.6.1 checkout options

These standard options are supported by `checkout` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date*
: Use the most recent revision no later than *date*. This option is sticky, and implies '`-P`'. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

`-f`
: Only useful with the '`-D` *date*' or '`-r` *tag*' flags. If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-k` *kflag*
: Process RCS keywords according to *kflag*. See co(1). This option is sticky; future updates of this file in this working directory will use the same *kflag*. The `status` command can be viewed to see the sticky options. See Section A.16 [status], page 92.

`-l`
: Local; run only in current working directory.

`-n`
: Do not run any checkout program (as specified with the '`-o`' option in the modules file; see Section B.1 [modules], page 97).

`-P`
: Prune empty directories.

`-p` Pipe files to the standard output.

`-r` *tag* Use revision *tag*. This option is sticky, and implies '`-P`'. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

In addition to those, you can use these special command options with `checkout`:

`-A` Reset any sticky tags, dates, or '`-k`' options. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

`-c` Copy the module file, sorted, to the standard output, instead of creating or modifying any files or directories in your working directory.

`-d` *dir* 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.

`-j` *tag* With two '`-j`' options, merge changes from the revision specified with the first '`-j`' option to the revision specified with the second '`j`' option, into the working directory.

With one '`-j`' option, merge changes from the ancestor revision to the revision specified with the '`-j`' option, into the working directory. The ancestor revision is the common ancestor of the revision which the working directory is based on, and the revision specified in the '`-j`' option.

In addition, each -j option can contain an 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: '`-j`*Symbolic_Tag*`:`*Date_Specifier*'.

See Chapter 8 [Merging], page 43.

`-N` Only useful together with '`-d` *dir*'. With this option, CVS will not shorten module paths in your working directory. (Normally, CVS shortens paths as much as possible when you specify an explicit target directory).

`-s` Like '`-c`', but include the status of all modules, and sort it by the status string. See Section B.1 [modules], page 97, for info about the '`-s`' option that is used inside the modules file to set the module status.

A.6.2 checkout examples

Get a copy of the module '`tc`':

```
$ cvs checkout tc
```

Get a copy of the module '`tc`' as it looked one day ago:

```
$ cvs checkout -D yesterday tc
```

A.7 commit—Check files into the repository

- Version 1.3 Synopsis: commit [-lnR] [-m 'log_message' | -f file] [-r revision] [files...]
- Version 1.3.1 Synopsis: commit [-lnRf] [-m 'log_message' | -F file] [-r revision] [files...]
- Requires: working directory, repository.

- Changes: repository.
- Synonym: ci

Warning: The '`-f` *file*' option will probably be renamed to '`-F` *file*', and '`-f`' will be given a new behavior in future releases of CVS.

Use `commit` when you want to incorporate changes from your working source files into the 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 `update` (see Section A.18 [update], page 94). `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 (see Section B.1 [modules], page 97, and see Section B.6 [loginfo], page 102) and placed in the RCS history file inside the repository. This log message can be retrieved with the `log` command; See Section A.12 [log], page 87. You can specify the log message on the command line with the '`-m` *message*' option, and thus avoid the editor invocation, or use the '`-f` *file*' option to specify that the argument file contains the log message.

A.7.1 commit options

These standard options are supported by `commit` (see Section A.4 [Common options], page 71, for a complete description of them):

`-l` Local; run only in current working directory.

`-n` Do not run any module program.

`-R` Commit directories recursively. This is on by default.

`-r` *revision*
: Commit to *revision*. *revision* must be either a branch, or a revision on the main trunk that is higher than any existing revision number. You cannot commit to a specific revision on a branch.

`commit` also supports these options:

`-F` *file* This option is present in CVS releases 1.3-s3 and later. Read the log message from *file*, instead of invoking an editor.

`-f` This option is present in CVS 1.3-s3 and later releases of CVS. Note that this is not the standard behavior of the '`-f`' option as defined in See Section A.4 [Common options], page 71.

Force CVS to commit a new revision even if you haven't made any changes to the file. If the current revision of *file* is 1.7, then the following two commands are equivalent:

```
$ cvs commit -f file
$ cvs commit -r 1.8 file
```

`-f` *file*
: This option is present in CVS releases 1.3, 1.3-s1 and 1.3-s2. Note that this is not the standard behavior of the '`-f`' option as defined in See Section A.4 [Common options], page 71.

  Read the log message from *file*, instead of invoking an editor.

`-m` *message*
: Use *message* as the log message, instead of invoking an editor.

A.7.2 commit examples

A.7.2.1 New major release number

When you make a major release of your product, you might want the revision numbers to track your major release number. You should normally not care about the revision numbers, but this is a thing that many people want to do, and it can be done without doing any harm.

To bring all your files up to the RCS revision 3.0 (including those that haven't changed), you might do:

```
$ cvs commit -r 3.0
```

Note that it is generally a bad idea to try to make the RCS revision number equal to the current release number of your product. You should think of the revision number as an internal number that the CVS package maintains, and that you generally never need to care much about. Using the `tag` and `rtag` commands you can give symbolic names to the releases instead. See Section A.17 [tag], page 93 and See Section A.15 [rtag], page 91.

Note that the number you specify with '`-r`' must be larger than any existing revision number. That is, if revision 3.0 exists, you cannot '`cvs commit -r 1.3`'.

A.7.2.2 Committing to a branch

You can commit to a branch revision (one that has an even number of dots) with the '`-r`' option. To create a branch revision, use the '`-b`' option of the `rtag` or `tag` commands (see Section A.17 [tag], page 93 or see Section A.15 [rtag], page 91). 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 disturbing main-line 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 -r FCS1_2 FCS1_2_Patch product_module
$ cvs checkout -r FCS1_2_Patch product_module
$ cd product_module
[[ hack away ]]
```

```
$ cvs commit
```

This works automatically since the '`-r`' option is sticky.

A.7.2.3 Creating the branch after editing

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 main-line 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:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs update -r EXPR1
$ cvs commit
```

The `update` command will make the '`-r EXPR1`' option sticky on all files. Note that your changes to the files will never be removed by the `update` command. The `commit` will automatically commit to the correct branch, because the '`-r`' is sticky. You could also do like this:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs commit -r EXPR1
```

but then, only those files that were changed by you will have the '`-r EXPR1`' sticky flag. If you hack away, and commit without specifying the '`-r EXPR1`' flag, some files may accidentally end up on the main trunk.

To work with you on the experimental change, others would simply do

```
$ cvs checkout -r EXPR1 whatever_module
```

A.8 diff—Run diffs between revisions

- Synopsis: diff [-l] [rcsdiff_options] [[-r rev1 | -D date1] [-r rev2 | -D date2]] [files...]
- Requires: working directory, repository.
- Changes: nothing.

The `diff` command is used to compare different revisions of files. The default action is to compare your working files with the revisions they were based on, and report any differences that are found.

If any file names are given, only those files are compared. If any directories are given, all files under them will be compared.

The exit status will be 0 if no differences were found, 1 if some differences were found, and 2 if any error occurred.

A.8.1 diff options

These standard options are supported by `diff` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date*
: Use the most recent revision no later than *date*. See '`-r`' for how this affects the comparison.

  CVS can be configured to pass the '`-D`' option through to `rcsdiff` (which in turn passes it on to `diff`. GNU diff uses '`-D`' as a way to put `cpp`-style '`#define`' statements around the output differences. There is no way short of testing to figure out how CVS was configured. In the default configuration CVS will use the '`-D` *date*' option.

`-k` *kflag*
: Process RCS keywords according to *kflag*. See co(1).

`-l`
: Local; run only in current working directory.

`-R`
: Examine directories recursively. This option is on by default.

`-r` *tag*
: Compare with revision *tag*. Zero, one or two '`-r`' options can be present. With no '`-r`' option, the working file will be compared with the revision it was based on. With one '`-r`', that revision will be compared to your current working file. With two '`-r`' options those two revisions will be compared (and your working file will not affect the outcome in any way).

  One or both '`-r`' options can be replaced by a '`-D` *date*' option, described above.

Any other options that are found are passed through to `rcsdiff`, which in turn passes them to `diff`. The exact meaning of the options depends on which `diff` you are using. The long options introduced in GNU diff 2.0 are not yet supported in CVS. See the documentation for your `diff` to see which options are supported.

A.8.2 diff examples

The following line produces a Unidiff ('`-u`' flag) between revision 1.14 and 1.19 of '`backend.c`'. Due to the '`-kk`' flag no keywords are substituted, so differences that only depend on keyword substitution are ignored.

```
$ cvs diff -kk -u -r 1.14 -r 1.19 backend.c
```

Suppose the experimental branch EXPR1 was based on a set of files tagged RELEASE_1_0. To see what has happened on that branch, the following can be used:

```
$ cvs diff -r RELEASE_1_0 -r EXPR1
```

A command like this can be used to produce a context diff between two releases:

```
$ cvs diff -c -r RELEASE_1_0 -r RELEASE_1_1 > diffs
```

If you are maintaining ChangeLogs, a command like the following just before you commit your changes may help you write the ChangeLog entry. All local modifications that have not yet been committed will be printed.

```
$ cvs diff -u | less
```

A.9 export—Export sources from CVS, similar to checkout

- Synopsis: export [-flNn] [-r rev|-D date] [-k subst] [-d dir] module...
- Requires: repository.
- Changes: current directory.

This command is a variant of `checkout`; use it when you want a copy of the source for module without the CVS administrative directories. For example, you might use `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.

One often would like to use '`-kv`' with `cvs export`. This causes any RCS keywords to be expanded such that an import done at some other site will not lose the keyword revision information. But be aware that doesn't handle an export containing binary files correctly. Also be aware that after having used '`-kv`', one can no longer use the `ident` command (which is part of the RCS suite—see ident(1)) which looks for RCS keyword strings. If you want to be able to use `ident` you must not use '`-kv`'.

A.9.1 export options

These standard options are supported by `export` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date*	Use the most recent revision no later than *date*.
`-f`	If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).
`-l`	Local; run only in current working directory.
`-n`	Do not run any checkout program.
`-R`	Export directories recursively. This is on by default.
`-r` *tag*	Use revision *tag*.

In addition, these options (that are common to `checkout` and `export`) are also supported:

`-d` *dir*	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.
`-k` *subst*	Set keyword expansion mode (see Section 16.4 [Substitution modes], page 63).
`-N`	Only useful together with '`-d` *dir*'. With this option, CVS will not shorten module paths in your working directory. (Normally, CVS shortens paths as much as possible when you specify an explicit target directory.)

A.10 history—Show status of files and users

- Synopsis: history [-report] [-flags] [-options args] [files...]
- Requires: the file '`$CVSROOT/CVSROOT/history`'
- Changes: nothing.

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

Logging must be enabled by creating the file '`$CVSROOT/CVSROOT/history`'.

Warning: `history` uses '`-f`', '`-l`', '`-n`', and '`-p`' in ways that conflict with the normal use inside CVS (see Section A.4 [Common options], page 71).

A.10.1 history 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).

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

`-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` *type* Extract a particular set of record types *type* from the CVS history. The types are indicated by single letters, which you may specify in combination.

Certain commands have a single record type:

`F`	release
`O`	checkout
`T`	rtag

One of four record types may result from an update:

`C`	A merge was necessary but collisions were detected (requiring manual merging).
`G`	A merge was necessary and it succeeded.
`U`	A working file was copied from the repository.
`W`	The working copy of a file was deleted during update (because it was gone from the repository).

One of three record types results from commit:

`A`	A file was added for the first time.
`M`	A file was modified.
`R`	A file was removed.

The options shown as '`-flags`' constrain or expand the report without requiring option arguments:

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

`-l` Show last modification only.

`-w` Show only the records for modifications done from the same working directory where `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*. This is slightly different from the normal use of '`-D` *date*', which selects the newest revision older than *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 the history file. This differs from the '`-r`' flag above in that it reads only the history file, not the RCS files, and is much faster.

`-u` *name* Show records for user *name*.

A.11 import—Import sources into CVS, using vendor branches

- Synopsis: import [-options] repository vendortag releasetag...
- Requires: Repository, source distribution directory.
- Changes: repository.

Use `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 from the outside source. See Chapter 12 [Tracking sources], page 53, for a discussion on this subject.

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 '`checkout -j`' to reconcile the differences, as import instructs you to do.

If CVS decides a file should be ignored (see Section B.8 [cvsignore], page 103), it does not import it and prints '`I `' followed by the filename (see Section A.11.2 [import output], page 86, for a complete description of the output).

If the file '`$CVSROOT/CVSROOT/cvswrappers`' exists, any file whose names match the specifications in that file will be treated as packages and the appropriate filtering will be performed on the file/directory before being imported, See Section B.2 [Wrappers], page 98.

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. *vendortag* is a tag for the entire branch (e.g., for 1.1.1). You must also specify at least one *releasetag* to identify the files at the leaves created each time you execute `import`.

A.11.1 import options

This standard option is supported by `import` (see Section A.4 [Common options], page 71, for a complete description):

`-m` *message*
: Use *message* as log information, instead of invoking an editor.

There are three additional special options.

`-b` *branch*
: Specify a first-level branch other than 1.1.1. Unless the '`-b` *branch*' flag is given, revisions will *always* be made to the branch 1.1.1—even if a *vendortag* that matches another branch is given! What happens in that case, is that the tag will be reset to 1.1.1. Warning: This behavior might change in the future.

`-k` *subst*
: Indicate the RCS keyword expansion mode desired. This setting will apply to all files created during the import, but not to any files that previously existed in the repository. See Section 16.4 [Substitution modes], page 63 for a list of valid '`-k`' settings.

`-I` *name*
: 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 !'.

  name can be a file name pattern of the same type that you can specify in the '`.cvsignore`' file. See Section B.8 [cvsignore], page 103.

`-W` *spec*
: Specify file names that should be filtered during import. You can use this option repeatedly.

  spec can be a file name pattern of the same type that you can specify in the '`.cvswrappers`' file. See Section B.2 [Wrappers], page 98.

A.11.2 import output

`import` keeps you informed of its progress by printing a line for each file, preceded by one character indicating the status of the file:

`U` *file*
: The file already exists in the repository and has not been locally modified; a new revision has been created (if necessary).

`N` *file* The file is a new file which has been added to the repository.

`C` *file* The file already exists in the repository but has been locally modified; you will have to merge the changes.

`I` *file* The file is being ignored (see Section B.8 [cvsignore], page 103).

`L` *file* The file is a symbolic link; at the moment (and for the forseeable future), symbolic links are ignored. (Various options in the '`modules`' file can be used to recreate symbolic links on checkout, update, etc.; see Section B.1 [modules], page 97.)

A.11.3 import examples

See Chapter 12 [Tracking sources], page 53, and See Section 5.1.1 [From files], page 23.

A.12 log—Print out log information for files

- Synopsis: log [options] [files...]
- Requires: repository, working directory.
- Changes: nothing.

Display log information for files. `log` used to call the RCS utility `rlog`. Although this is no longer true in the current sources, this history determines the format of the output and the options, which are not quite in the style of the other CVS commands.

The output includes the location of the RCS file, the *head* revision (the latest revision on the trunk), all symbolic names (tags) and some other things. For each revision, the revision number, the author, the number of lines added/deleted and the log message are printed. All times are displayed in Coordinated Universal Time (UTC). (Other parts of CVS print times in the local timezone).

A.12.1 log options

By default, `log` prints all information that is available. All other options restrict the output.

`-b` Print information about the revisions on the default branch, normally the highest branch on the trunk.

`-d` *dates* Print information about revisions with a checkin date/time in the range given by the semicolon-separated list of dates. The date formats accepted are those accepted by the '`-D`' option to many other CVS commands (see Section A.4 [Common options], page 71). Dates can be combined into ranges as follows:

d1`<`*d2*
d2`>`*d1* Select the revisions that were deposited between *d1* and *d2*.

`<`*d*
d`>` Select all revisions dated *d* or earlier.

d`<`
`>`*d* Select all revisions dated *d* or later.

d Select the single, latest revision dated *d* or earlier.

The '`>`' or '`<`' characters may be followed by '`=`' to indicate an inclusive range rather than an exclusive one.

Note that the separator is a semicolon (;).

`-h`
: Print only the RCS pathname, working pathname, head, default branch, access list, locks, symbolic names, and suffix.

`-l`
: Local; run only in current working directory. (Default is to run recursively).

`-N`
: Do not print the list of tags for this file. This option can be very useful when your site uses a lot of tags, so rather than "more"'ing over 3 pages of tag information, the log information is presented without tags at all.

`-R`
: Print only the name of the RCS history file.

`-r`*revisions*
: Print information about revisions given in the comma-separated list *revisions* of revisions and ranges. The following table explains the available range formats:

rev1`:`*rev2*	Revisions *rev1* to *rev2* (which must be on the same branch).
`:`*rev*	Revisions from the beginning of the branch up to and including *rev*.
rev`:`	Revisions starting with *rev* to the end of the branch containing *rev*.
branch	An argument that is a branch means all revisions on that branch.
branch1`:`*branch2*	A range of branches means all revisions on the branches in that range.
branch`.`	The latest revision in *branch*.

A bare '`-r`' with no revisions means the latest revision on the default branch, normally the trunk. There can be no space between the '`-r`' option and its argument.

`-s` *states*
: Print information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

`-t`
: Print the same as '`-h`', plus the descriptive text.

`-w`*logins*
: Print 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. There can be no space between the '`-w`' option and its argument.

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

A.12.2 log examples

Contributed examples are gratefully accepted.

A.13 rdiff—'patch' format diffs between releases

- 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.

A.13.1 rdiff options

These standard options are supported by `rdiff` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date* Use the most recent revision no later than *date*.

`-f` If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-l` Local; don't descend subdirectories.

`-r` *tag* Use revision *tag*.

In addition to the above, these options are available:

`-c` Use the context diff format. This is the default format.

`-s` Create a summary change report instead of a patch. The summary includes information about files that were changed or added between the releases. It is sent to the standard output device. This is useful for finding out, for example, which files have changed between two dates or revisions.

`-t` 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.

`-u` Use the unidiff format for the context diffs. This option is not available if your diff does not support the unidiff format. Remember that old versions of the `patch` program can't handle the unidiff format, so if you plan to post this patch to the net you should probably not use '`-u`'.

`-V` *vn* Expand RCS keywords according to the rules current in RCS version *vn* (the expansion format changed with RCS version 5).

A.13.2 rdiff examples

Suppose you receive mail from `foo@bar.com` asking for an update from release 1.2 to 1.4 of the tc compiler. You have no such patches on hand, but with CVS that can easily be fixed with a command such as this:

```
$ cvs rdiff -c -r FOO1_2 -r FOO1_4 tc | \
$$ Mail -s 'The patches you asked for' foo@bar.com
```

Suppose you have made release 1.3, and forked a branch called 'R_1_3fix' for bugfixes. 'R_1_3_1' corresponds to release 1.3.1, which was made some time ago. Now, you want to see how much development has been done on the branch. This command can be used:

```
$ cvs patch -s -r R_1_3_1 -r R_1_3fix module-name
cvs rdiff: Diffing module-name
File ChangeLog,v changed from revision 1.52.2.5 to 1.52.2.6
File foo.c,v changed from revision 1.52.2.3 to 1.52.2.4
File bar.h,v changed from revision 1.29.2.1 to 1.2
```

A.14 release—Indicate that a Module is no longer in use

- release [-d] directories...
- 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 (see Section B.9 [history file], page 104) 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 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.

A.14.1 release options

The `release` command supports one command option:

`-d` Delete your working copy of the file if the release succeeds. If this flag is not given your files will remain in your working directory.

Warning: The `release` command deletes all directories and files recursively. This has the very serious side-effect that any directory that you have created inside your checked-out sources, and not added to the repository (using the `add` command; see Chapter 10 [Adding files], page 49) will be silently deleted—even if it is non-empty!

A.14.2 release output

Before `release` releases your sources it will print a one-line message for any file that is not up-to-date.

Warning: Any new directories that you have created, but not added to the CVS directory hierarchy with the `add` command (see Chapter 10 [Adding files], page 49) will be silently ignored (and deleted, if '`-d`' is specified), even if they contain files.

`U` *file*	There exists a newer revision of this file in the repository, and you have not modified your local copy of the file.
`A` *file*	The file has been added to your private copy of the sources, but has not yet been committed to the repository. If you delete your copy of the sources this file will be lost.
`R` *file*	The file has been removed from your private copy of the sources, but has not yet been removed from the repository, since you have not yet committed the removal. See Section A.7 [commit], page 78.
`M` *file*	The file is modified in your working directory. There might also be a newer revision inside the repository.
`?` *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, and see Section B.8 [cvsignore], page 103). If you remove your working sources, this file will be lost. Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

A.14.3 release examples

Release the module, and delete your local working copy of the files.

```
$ cd ..         # You must stand immediately above the
                # sources when you issue 'cvs release'.
$ cvs release -d tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module 'tc': y
$
```

A.15 rtag—Add a symbolic tag to a module

- rtag [-falnR] [-b] [-d] [-r tag | -Ddate] symbolic_tag modules...
- Requires: repository.
- Changes: repository.
- Synonym: rfreeze

You can use this command to assign symbolic tags to particular, explicitly specified source revisions in the repository. `rtag` works directly on the repository contents (and requires no prior checkout). Use `tag` instead (see Section A.17 [tag], page 93), to base the selection of revisions on the contents of your working directory.

If you attempt to use a tag name that already exists, CVS will complain and not overwrite that tag. Use the '`-F`' option to force the new tag value.

A.15.1 rtag options

These standard options are supported by `rtag` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D` *date*
: Tag the most recent revision no later than *date*.

`-f`
: Only useful with the '`-D` *date*' or '`-r` *tag*' flags. If no matching revision is found, use the most recent revision (instead of ignoring the file).

`-F`
: Overwrite an existing tag of the same name on a different revision. This option is new in CVS 1.4. The old behavior is matched by '`cvs tag -F`'.

`-l`
: Local; run only in current working directory.

`-n`
: Do not run any tag program that was specified with the '`-t`' flag inside the '`modules`' file. (see Section B.1 [modules], page 97).

`-R`
: Commit directories recursively. This is on by default.

`-r` *tag*
: Only tag those files that contain *tag*. This can 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.

In addition to the above common options, these options are available:

`-a`
: Use the '`-a`' option to have `rtag` look in the '`Attic`' (see Chapter 11 [Removing files], page 51) for removed files that contain the specified tag. The tag is removed from these files, which makes it convenient to re-use a symbolic tag as development continues (and files get removed from the up-coming distribution).

`-b`
: Make the tag a branch tag. See Chapter 7 [Branches], page 37.

`-d`
: Delete the tag instead of creating it.

  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, or if you mistagged a module).

A.16 status—Display status information on checked out files

- 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. For information on the basic output see Section 6.1 [File status], page 27. For information on the `Sticky tag` and `Sticky date` output, see Section 7.4 [Sticky tags], page 40. For information on the `Sticky options` output, see the '`-k`' option in Section A.18.1 [update options], page 94.

You can also use this command to determine the potential impact of a '`cvs update`' on your working source directory—but remember that things might change in the repository before you run `update`.

A.16.1 status options

These standard options are supported by `status` (see Section A.4 [Common options], page 71, for a complete description of them):

`-l`
: Local; run only in current working directory.

`-R`
: Commit directories recursively. This is on by default.

There is one additional option:

`-v`
: Verbose. In addition to the information normally displayed, print all symbolic tags, together with the numerical value of the revision or branch they refer to. For more information, see Section 7.1 [Tags], page 37

A.17 tag—Add a symbolic tag to checked out versions of files

- tag [-lR] [-b] [-c] [-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 re-tagged.

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.

This command can also be used to delete a symbolic tag, or to create a branch. See the options section below.

If you attempt to use a tag name that already exists, CVS will complain and not overwrite that tag. Use the '`-F`' option to force the new tag value.

A.17.1 tag options

These standard options are supported by `tag` (see Section A.4 [Common options], page 71, for a complete description of them):

`-F` Overwrite an existing tag of the same name on a different revision. This option is new in CVS 1.4. The old behavior is matched by '`cvs tag -F`'.

`-l` Local; run only in current working directory.

`-R` Commit directories recursively. This is on by default.

Two special options are available:

`-b` The -b option makes the tag a branch tag (see Chapter 7 [Branches], page 37), allowing concurrent, isolated development. This is most useful for creating a patch to a previously released software distribution.

`-c` The -c option checks that all files which are to be tagged are unmodified. This can be used to make sure that you can reconstruct the current file contents.

`-d` Delete a tag.

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 permanently discards some historical information, which may later turn out to be valuable.

A.18 update—Bring work tree in sync with repository

- update [-AdflPpR] [-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.

A.18.1 update options

These standard options are available with `update` (see Section A.4 [Common options], page 71, for a complete description of them):

`-D date` Use the most recent revision no later than *date*. This option is sticky, and implies '`-P`'. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

`-f` Only useful with the '`-D` *date*' or '`-r` *tag*' flags. If no matching revision is found, retrieve the most recent revision (instead of ignoring the file).

`-k` *kflag* Process RCS keywords according to *kflag*. See co(1). This option is sticky; future updates of this file in this working directory will use the same *kflag*. The `status` command can be viewed to see the sticky options. See Section A.16 [status], page 92.

`-l` Local; run only in current working directory. See Chapter 9 [Recursive behavior], page 47.

`-P` Prune empty directories.

`-p` Pipe files to the standard output.

`-R` Operate recursively. This is on by default. See Chapter 9 [Recursive behavior], page 47.

`-r tag` Retrieve revision *tag*. This option is sticky, and implies '`-P`'. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

These special options are also available with `update`.

`-A` Reset any sticky tags, dates, or '`-k`' options. See Section 7.4 [Sticky tags], page 40, for more information on sticky tags/dates.

`-d` 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.

`-I` *name* 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. Use '`-I !`' to avoid ignoring any files at all. See Section B.8 [cvsignore], page 103, for other ways to make CVS ignore some files.

`-W`*spec* Specify file names that should be filtered during update. You can use this option repeatedly.

spec can be a file name pattern of the same type that you can specify in the '`.cvswrappers`' file. See Section B.2 [Wrappers], page 98.

`-j`*revision* With two '`-j`' options, merge changes from the revision specified with the first '`-j`' option to the revision specified with the second '`j`' option, into the working directory.

With one '`-j`' option, merge changes from the ancestor revision to the revision specified with the '`-j`' option, into the working directory. The ancestor revision is the common ancestor of the revision which the working directory is based on, and the revision specified in the '`-j`' option.

In addition, each -j option can contain an 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: '`-j`*Symbolic_Tag*`:`*Date_Specifier*'.

See Chapter 8 [Merging], page 43.

A.18.2 update output

`update` and `checkout` keep you informed of its progress by printing a line for each file, preceded by one character indicating 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 source repository when you run `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 source repository when you run `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.

  CVS will print some messages if it merges your work, and a backup copy of your working file (as it looked before you ran `update`) will be made. The exact name of that file is printed while `update` runs.

`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 revisions; an unmodified copy of your file is also in your working directory, with the name '`.#`*file*`.`*revision*' where *revision* is the RCS revision that your modified file started from. Resolve the conflict as described in Section 6.3 [Conflicts example], page 28 (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.) Under VMS, the file name starts with '`__`' rather than '`.#`'.

`?` *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, and see Section B.8 [cvsignore], page 103).

  Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

A.18.3 update examples

The following line will display all files which are not up-to-date without actually change anything in your working directory. It can be used to check what has been going on with the project.

```
$ cvs -n -q update
```

Appendix B Reference manual for the Administrative files

Inside the repository, in the directory '`$CVSROOT/CVSROOT`', there are a number of supportive files for CVS. You can use CVS in a limited fashion without any of them, but if they are set up properly they can help make life easier. For a discussion of how to edit them, See Section 4.3 [Intro administrative files], page 16.

The most important of these files is the '`modules`' file, which defines the modules inside the repository.

B.1 The modules file

The '`modules`' file records your definitions of names for collections of source code. CVS will use these definitions if you use CVS to update the modules file (use normal commands like `add`, `commit`, etc).

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, `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 the source repository. 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.

`-d` *name* Name the working directory something other than the module name.

`-e` *prog* Specify a program *prog* to run whenever files in a module are exported. *prog* runs with a single argument, the module name.

`-i` *prog*
: 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*
: Specify a program *prog* to run whenever files in a module are checked out. *prog* runs with a single argument, the module name.

`-s` *status*
: Assign a status to the module. When the module file is printed with '`cvs checkout -s`' the modules are sorted according to primarily module status, and secondarily according to the module name. This option has no other meaning. You can use this option for several things besides status: for instance, list the person that is responsible for this module.

`-t` *prog*
: Specify a program *prog* to run whenever files in a module are tagged with `rtag`. *prog* runs with two arguments: the module name and the symbolic tag specified to `rtag`. There is no way to specify a program to run when `tag` is executed.

`-u` *prog*
: 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.

B.2 The cvswrappers file

Wrappers allow you to set a hook which transforms files on their way in and out of CVS. Most or all of the wrappers features do not work with client/server CVS.

The file '`cvswrappers`' defines the script that will be run on a file when its name matches a regular expresion. There are two scripts that can be run on a file or directory. One script is executed on the file/directory before being checked into the repository (this is denoted with the `-t` flag) and the other when the file is checked out of the repository (this is denoted with the `-f` flag)

The '`cvswrappers`' also has a '`-m`' option to specify the merge methodology that should be used when the file is updated. `MERGE` means the usual CVS behavior: try to merge the files (this generally will not work for binary files). `COPY` means that `cvs update` will merely copy one version over the other, and require the user using mechanisms outside CVS, to insert any necessary changes. The '`-m`' wrapper option only affects behavior when merging is done on update; it does not affect how files are stored. See See Chapter 17 [Binary files], page 65, for more on binary files.

The basic format of the file '`cvswrappers`' is:

```
wildcard     [option value][option value]...

where option is one of
-f           from cvs filter         value: path to filter
-t           to cvs filter           value: path to filter
-m           update methodology      value: MERGE or COPY
-k           keyword expansion       value: expansion mode

and value is a single-quote delimited value.

*.nib    -f 'unwrap %s' -t 'wrap %s %s' -m 'COPY'
```

```
*.c      -t 'indent %s %s'
```

The above example of a '`cvswrappers`' file states that all files/directories that end with a `.nib` should be filtered with the '`wrap`' program before checking the file into the repository. The file should be filtered though the '`unwrap`' program when the file is checked out of the repository. The '`cvswrappers`' file also states that a `COPY` methodology should be used when updating the files in the repository (that is no merging should be performed).

The last example line says that all files that end with a `*.c` should be filtered with '`indent`' before being checked into the repository. Unlike the previous example no filtering of the `*.c` file is done when it is checked out of the repository. The `-t` filter is called with two arguments, the first is the name of the file/directory to filter and the second is the pathname to where the resulting filtered file should be placed.

The `-f` filter is called with one argument, which is the name of the file to filter from. The end result of this filter will be a file in the users directory that they can work on as they normally would.

For another example, the following command imports a directory, treating files whose name ends in '`.exe`' as binary:

```
cvs import -I ! -W "*.exe -k 'b'" first-dir vendortag reltag
```

B.3 The commit support files

The '`-i`' flag in the '`modules`' file can be used to run a certain program whenever files are committed (see Section B.1 [modules], page 97). The files described in this section provide other, more flexible, ways to run programs whenever something is committed.

There are three kind of programs that can be run on commit. They are specified in files in the repository, as described below. The following table summarizes the file names and the purpose of the corresponding programs.

'`commitinfo`'
: The program is responsible for checking that the commit is allowed. If it exits with a non-zero exit status the commit will be aborted.

'`editinfo`'
: The specified program is used to edit the log message, and possibly verify that it contains all required fields. This is most useful in combination with the '`rcsinfo`' file, which can hold a log message template (see Section B.7 [rcsinfo], page 103).

'`loginfo`'
: The specified program is called when the commit is complete. It receives the log message and some additional information and can store the log message in a file, or mail it to appropriate persons, or maybe post it to a local newsgroup, or... Your imagination is the limit!

B.3.1 The common syntax

The four files '`commitinfo`', '`loginfo`', '`rcsinfo`' and '`editinfo`' all have a common format. The purpose of the files are described later on. The common syntax is described here.

Each line contains the following:

- A regular expression
- A whitespace separator—one or more spaces and/or tabs.
- A file name or command-line template.

Blank lines are ignored. Lines that start with the character '`#`' are treated as comments. Long lines unfortunately can *not* be broken in two parts in any way.

The first regular expression that matches the current directory name in the repository is used. The rest of the line is used as a file name or command-line as appropriate.

B.4 Commitinfo

The '`commitinfo`' file defines 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. This could be used, for instance, to verify that the changed files conform to to your site's standards for coding practice.

As mentioned earlier, each line in the '`commitinfo`' file consists of a regular expression and a command-line template. The template can include a program name and any number of arguments you wish to supply to it. 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).

The first line with a regular expression matching the relative path to the module will be used. If the command returns a non-zero exit status the commit will be aborted.

If the repository name does not match any of the regular expressions in this file, the '`DEFAULT`' line is used, if it is specified.

All occurances of the name '`ALL`' appearing as a regular expression are used in addition to the first matching regular expression or the name '`DEFAULT`'.

Note: when CVS is accessing a remote repository, '`commitinfo`' will be run on the *remote* (i.e., server) side, not the client side (see Section 4.6 [Remote repositories], page 17).

B.5 Editinfo

If you want to make sure that all log messages look the same way, you can use the '`editinfo`' file to specify a program that is used to edit the log message. This program could be a custom-made editor that always enforces a certain style of the log message, or maybe a simple shell script that calls an editor, and checks that the entered message contains the required fields.

If no matching line is found in the '`editinfo`' file, the editor specified in the environment variable `$CVSEDITOR` is used instead. If that variable is not set, then the environment variable `$EDITOR` is used instead. If that variable is not set a precompiled default, normally `vi`, will be used.

The 'editinfo' file is often most useful together with the 'rcsinfo' file, which can be used to specify a log message template.

Each line in the 'editinfo' file consists of a regular expression and a command-line template. The template must include a program name, and can include any number of arguments. The full path to the current log message template file is appended to the template.

One thing that should be noted is that the 'ALL' keyword is not supported. If more than one matching line is found, the first one is used. This can be useful for specifying a default edit script in a module, and then overriding it in a subdirectory.

If the repository name does not match any of the regular expressions in this file, the 'DEFAULT' line is used, if it is specified.

If the edit script exits with a non-zero exit status, the commit is aborted.

Note: when CVS is accessing a remote repository, or when the '-m' or '-F' options to `cvs commit` are used, 'editinfo' will not be consulted. There is no good workaround for this.

B.5.1 Editinfo example

The following is a little silly example of a 'editinfo' file, together with the corresponding 'rcsinfo' file, the log message template and an editor script. We begin with the log message template. We want to always record a bug-id number on the first line of the log message. The rest of log message is free text. The following template is found in the file '/usr/cvssupport/tc.template'.

```
BugId:
```

The script '/usr/cvssupport/bugid.edit' is used to edit the log message.

```
#!/bin/sh
#
#       bugid.edit filename
#
#  Call $EDITOR on FILENAME, and verify that the
#  resulting file contains a valid bugid on the first
#  line.
if [ "x$EDITOR" = "x" ]; then EDITOR=vi; fi
if [ "x$CVSEDITOR" = "x" ]; then CVSEDITOR=$EDITOR; fi
$CVSEDITOR $1
until head -1|grep '^BugId:[ ]*[0-9][0-9]*$' < $1
do  echo -n  "No BugId found.  Edit again? ([y]/n)"
    read ans
    case ${ans} in
        n*) exit 1;;
    esac
    $CVSEDITOR $1
done
```

The '`editinfo`' file contains this line:

```
^tc     /usr/cvssupport/bugid.edit
```

The '`rcsinfo`' file contains this line:

```
^tc     /usr/cvssupport/tc.template
```

B.6 Loginfo

The '`loginfo`' file is used to control where '`cvs commit`' log information is sent. The first entry on a line is a regular expression which is tested against the directory that the change is being made to, relative to the `$CVSROOT`. If a match is found, then the remainder of the line is a filter program that should expect log information on its standard input.

The filter program may use one and only one % modifier (a la printf). If '`%s`' is specified in the filter program, a brief title is included (enclosed in single quotes) showing the modified file names.

If the repository name does not match any of the regular expressions in this file, the '`DEFAULT`' line is used, if it is specified.

All occurances of the name '`ALL`' appearing as a regular expression are used in addition to the first matching regular expression or '`DEFAULT`'.

The first matching regular expression is used.

See Section B.3 [commit files], page 99, for a description of the syntax of the '`loginfo`' file.

Note: when CVS is accessing a remote repository, '`loginfo`' will be run on the *remote* (i.e., server) side, not the client side (see Section 4.6 [Remote repositories], page 17).

B.6.1 Loginfo example

The following '`loginfo`' file, together with the tiny shell-script below, appends all log messages to the file '`$CVSROOT/CVSROOT/commitlog`', and any commits to the administrative files (inside the '`CVSROOT`' directory) are also logged in '`/usr/adm/cvsroot-log`'.

```
ALL             /usr/local/bin/cvs-log $CVSROOT/CVSROOT/commitlog
^CVSROOT        /usr/local/bin/cvs-log /usr/adm/cvsroot-log
```

The shell-script '`/usr/local/bin/cvs-log`' looks like this:

```
#!/bin/sh
(echo "------------------------------------------------------------";
 echo -n $USER"  ";
 date;
 echo;
```

```
    sed '1s+'${CVSROOT}'++') >> $1
```

B.6.2 Keeping a checked out copy

It is often useful to maintain a directory tree which contains files which correspond to the latest version in the repository. For example, other developers might want to refer to the latest sources without having to check them out, or you might be maintaining a web site with CVS and want every checkin to cause the files used by the web server to be updated.

The way to do this is by having loginfo invoke `cvs update`. Doing so in the naive way will cause a problem with locks, so the `cvs update` must be run in the background. Here is an example (this should all be on one line):

```
^cyclic-pages (date; cat; (sleep 2; cd /u/www/local-docs;
 cvs -q update -d) &) >> $CVSROOT/CVSROOT/updatelog 2>&1
```

This will cause checkins to repository directories starting with `cyclic-pages` to update the checked out tree in '`/u/www/local-docs`'.

B.7 Rcsinfo

The '`rcsinfo`' file can be used to specify a form to edit when filling out the commit log. The '`rcsinfo`' file has a syntax similar to the '`editinfo`', '`commitinfo`' and '`loginfo`' files. See Section B.3.1 [syntax], page 99. Unlike the other files the second part is *not* a command-line template. Instead, the part after the regular expression should be a full pathname to a file containing the log message template.

If the repository name does not match any of the regular expressions in this file, the '`DEFAULT`' line is used, if it is specified.

All occurances of the name '`ALL`' appearing as a regular expression are used in addition to the first matching regular expression or '`DEFAULT`'.

The log message template will be used as a default log message. If you specify a log message with '`cvs commit -m` *message*' or '`cvs commit -f` *file*' that log message will override the template.

See Section B.5.1 [editinfo example], page 101, for an example '`rcsinfo`' file.

When CVS is accessing a remote repository, the contents of '`rcsinfo`' at the time a directory is first checked out will specify a template which does not then change. If you edit '`rcsinfo`' or its templates, you may need to check out a new working directory.

B.8 Ignoring files via cvsignore

There are certain file names that frequently occur inside your working copy, but that you don't want to put under CVS control. Examples are all the object files that you get while you compile your sources. Normally, when you run '`cvs update`', it prints a line for each file it encounters that it doesn't know about (see Section A.18.2 [update output], page 95).

CVS has a list of files (or sh(1) file name patterns) that it should ignore while running `update`, `import` and `release`. This list is constructed in the following way.

- The list is initialized to include certain file name patterns: 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 file name patterns is:

```
RCS     SCCS     CVS     CVS.adm
RCSLOG  cvslog.*
tags    TAGS
.make.state      .nse_depinfo
*~      #*       .#*     ,*      _$*     *$
*.old   *.bak    *.BAK   *.orig  *.rej   .del-*
*.a     *.olb    *.o     *.obj   *.so    *.exe
*.Z     *.elc    *.ln
core
```

- The per-repository list in '`$CVSROOT/CVSROOT/cvsignore`' is appended to the list, if that file exists.
- The per-user list in '`.cvsignore`' in your home directory is appended to the list, if it exists.
- Any entries in the environment variable `$CVSIGNORE` is appended to the list.
- Any '`-I`' options given to CVS is appended.
- As CVS traverses through your directories, the contents of any '`.cvsignore`' will be appended to the list. The patterns found in '`.cvsignore`' are only valid for the directory that contains them, not for any sub-directories.

In any of the 5 places listed above, a single exclamation mark ('`!`') clears the ignore list. This can be used if you want to store any file which normally is ignored by CVS.

B.9 The history file

The file '`$CVSROOT/CVSROOT/history`' is used to log information for the `history` command (see Section A.10 [history], page 84). This file must be created to turn on logging. This is done automatically if the `cvs init` command is used to set up the repository (see Section 4.5 [Creating a repository], page 17).

The file format of the '`history`' file is documented only in comments in the CVS source code, but generally programs should use the `cvs history` command to access it anyway, in case the format changes with future releases of CVS.

B.10 Expansions in administrative files

Sometimes in writing an administrative file, you might want the file to be able to know various things based on environment CVS is running in. There are several mechanisms to do that.

To find the home directory of the user running CVS (from the `HOME` environment variable), use '`~`' followed by '`/`' or the end of the line. Likewise for the home directory of *user*, use '`~`*user*'. These variables are expanded on the server machine, and don't get any resonable expansion if pserver (see Section 4.6.2 [Password authenticated], page 19) is in used; therefore user variables (see below) may be a better choice to customize behavior based on the user running CVS.

One may want to know about various pieces of information internal to CVS. A CVS internal variable has the syntax `${`*variable*`}`, where *variable* starts with a letter and consists of alphanumberic characters and '`_`'. If the character following *variable* is a non-alphanumeric character other than '`_`', the '`{`' and '`}`' can be omitted. The CVS internal variables are:

`CVSROOT`
: This is the value of the CVS root in use. See Chapter 4 [Repository], page 13, for a description of the various ways to specify this.

`RCSBIN`
: This is the value CVS is using for where to find RCS binaries. See Section A.3 [Global options], page 70, for a description of how to specify this.

`CVSEDITOR`
`VISUAL`
`EDITOR`
: These all expand to the same value, which is the editor that CVS is using. See Section A.3 [Global options], page 70, for how to specify this.

`USER`
: Username of the user running CVS (on the CVS server machine).

If you want to pass a value to the administrative files which the user that is running CVS can specify, use a user variable. To expand a user variable, the administrative file contains `${=`*variable*`}`. To set a user variable, specify the global option '`-s`' to CVS, with argument *variable*`=`*value*. It may be particularly useful to specify this option via '`.cvsrc`' (see Section A.2 [~/.cvsrc], page 69).

For example, if you want the administrative file to refer to a test directory you might create a user variable `TESTDIR`. Then if CVS is invoked as `cvs -s TESTDIR=/work/local/tests`, and the administrative file contains `sh ${=TESTDIR}/runtests`, then that string is expanded to `sh /work/local/tests/runtests`.

All other strings containing '`$`' are reserved; there is no way to quote a '`$`' character so that '`$`' represents itself.

Appendix C All environment variables which affect CVS

This is a complete list of all environment variables that affect CVS.

`$CVSIGNORE`
: A whitespace-separated list of file name patterns that CVS should ignore. See Section B.8 [cvsignore], page 103.

`$CVSWRAPPERS`
: A whitespace-separated list of file name patterns that CVS should treat as wrappers. See Section B.2 [Wrappers], page 98.

`$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.

`$CVSROOT`
: Should contain the full pathname to the root of the CVS source repository (where the RCS history files are kept). This information must be available to CVS for most commands to execute; if `$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...`' Once you have checked out a working directory, CVS stores the appropriate root (in the file '`CVS/Root`'), so normally you only need to worry about this when initially checking out a working directory.

`$EDITOR`
`$CVSEDITOR`
: Specifies the program to use for recording log messages during commit. If not set, the default is '`/usr/ucb/vi`'. `$CVSEDITOR` overrides `$EDITOR`. `$CVSEDITOR` does not exist in CVS 1.3, but the next release will probably include it.

`$PATH`
: If `$RCSBIN` is not set, and no path is compiled into CVS, it will use `$PATH` to try to find all programs it uses.

`$RCSBIN`
: This is the value CVS is using for where to find RCS binaries. See Section A.3 [Global options], page 70, for a description of how to specify this. If not set, a compiled-in value is used, or your `$PATH` is searched.

`$HOME`

`$HOMEPATH`
: Used to locate the directory where the '`.cvsrc`' file is searched (`$HOMEPATH` is used for Windows-NT). see Section A.2 [~/.cvsrc], page 69

`$CVS_RSH`
: Specifies the external program which CVS connects with, when `:ext:` access method is specified. see Section 4.6.1 [Connecting via rsh], page 18.

`$CVS_SERVER`
: Used in client-server mode when accessing a remote repository using RSH. It specifies the name of the program to start on the server side when accessing a remote repository using RSH. The default value is `cvs`. see Section 4.6.1 [Connecting via rsh], page 18

`$CVS_PASSFILE`
: Used in client-server mode when accessing the `cvs login server`. Default value is '`$HOME/.cvspass`'. see Section 4.6.2.2 [Password authentication client], page 20

`$CVS_PASSWORD`
: Used in client-server mode when accessing the `cvs login server`. see Section 4.6.2.2 [Password authentication client], page 20

`$CVS_CLIENT_PORT`
: Used in client-server mode when accessing the server via Kerberos. see Section 4.6.3 [Kerberos authenticated], page 21

`$CVS_RCMD_PORT`
: Used in client-server mode. If set, specifies the port number to be used when accessing the RCMD demon on the server side. (Currently not used for Unix clients).

`$CVS_CLIENT_LOG`
: Used for debugging only in client-server mode. If set, everything send to the server is logged into '`$CVS_CLIENT_LOG.in`' and everything send from the server is logged into '`$CVS_CLIENT_LOG.out`'.

`$CVS_SERVER_SLEEP`
: Used only for debugging the server side in client-server mode. If set, delays the start of the server child process the the specified amount of seconds so that you can attach to it with a debugger.

`$CVS_IGNORE_REMOTE_ROOT`
: (What is the purpose of this variable?)

`$COMSPEC`
: Used under OS/2 only. It specifies the name of the command interpreter and defaults to CMD.EXE.

`$TMPDIR`
`$TMP`
`$TEMP`
: Directory in which temporary files are located. Those parts of CVS which are implemented using RCS inspect the above variables in the order they appear above and the first value found is taken; if none of them are set, a host-dependent default is used, typically '`/tmp`'. The CVS server uses `TMPDIR`. See Section A.3 [Global options], page 70, for a description of how to specify this. Some parts of CVS will always use '`/tmp`' (via the `tmpnam` function provided by the system).

  On Windows NT, `TMP` is used (via the `_tempnam` function provided by the system).

  The `patch` program which is used by the CVS client uses `TMPDIR`, and if it is not set, uses '`/tmp`' (at least with GNU patch 2.1).

CVS invokes RCS to perform certain operations. The following environment variables affect RCS. Note that if you are using the client/server CVS, these variables need to be set on the server side (which may or not may be possible depending on how you are connecting). There is probably not any need to set any of them, however.

`$LOGNAME`
`$USER`
: If set, they affect who RCS thinks you are. If you have trouble checking in files it might be because your login name differs from the setting of e.g. `$LOGNAME`.

`$RCSINIT`
: Options prepended to the argument list, separated by spaces. A backslash escapes spaces within an option. The `$RCSINIT` options are prepended to the argument lists of most RCS commands.

Appendix D Troubleshooting

D.1 Magic branch numbers

Externally, branch numbers consist of an odd number of dot-separated decimal integers. See Section 2.1 [Revision numbers], page 7. That is not the whole truth, however. For efficiency reasons CVS sometimes inserts an extra 0 in the second rightmost position (1.2.3 becomes 1.2.0.3, 8.9.10.11.12 becomes 8.9.10.11.0.12 and so on).

CVS does a pretty good job at hiding these so called magic branches, but in a few places the hiding is incomplete:

- The magic branch number appears in the output from `cvs log`.
- You cannot specify a symbolic branch name to `cvs admin`.

You can use the `admin` command to reassign a symbolic name to a branch the way RCS expects it to be. If `R4patches` is assigned to the branch 1.4.2 (magic branch number 1.4.0.2) in file '`numbers.c`' you can do this:

```
$ cvs admin -NR4patches:1.4.2 numbers.c
```

It only works if at least one revision is already committed on the branch. Be very careful so that you do not assign the tag to the wrong number. (There is no way to see how the tag was assigned yesterday).

Appendix E GNU GENERAL PUBLIC LICENSE

Index

Short Contents

Table of Contents