cvsnt--Concurrent Versions System (cvsnt) 2.8.01.9054

COLLABORATORS

	TITLE : cvsnt--Concurrent Versions System (cvsnt) 2.8.01.9054	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		May 2, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Overview **1**
1.1 What is CVS? . . . 1
1.2 What is CVS not? . . . 2
1.3 A sample session . . . 3
1.3.1 Getting the source . . . 3
1.3.2 Committing your changes . . . 3
1.3.3 Cleaning up . . . 4
1.3.4 Viewing differences . . . 4

2 The Repository **6**
2.1 Telling CVS where your repository is . . . 6
2.2 How data is stored in the repository . . . 7
2.2.1 Where files are stored within the repository . . . 7
2.2.2 File permissions . . . 8
2.2.3 The attic . . . 9
2.2.4 The CVS directory in the repository . . . 9
2.2.5 CVS locks in the repository . . . 9
2.2.6 How files are stored in the CVSROOT directory . . . 10
2.3 How data is stored in the working directory . . . 10
2.4 The administrative files . . . 12
2.4.1 Editing administrative files . . . 13
2.5 Multiple repositories . . . 13
2.6 Creating a repository . . . 13
2.7 Backing up a repository . . . 14
2.8 Moving a repository . . . 14
2.9 Remote repositories . . . 14
2.9.1 Server requirements . . . 15
2.9.2 Connecting with ssh . . . 15
2.9.3 Using 3rd party clients via the extnt wrapper . . . 16
2.9.4 Direct connection with password authentication . . . 16

2.9.4.1 Setting up the server for Authentication . . . 17
2.9.4.2 CVS passwd file . . . 17
2.9.4.3 Using the client with password authentication . . . 18
2.9.4.4 Security considerations with password authentication . . . 19
2.9.5 Direct connection with GSSAPI . . . 19
2.9.6 Connecting with fork . . . 19
2.9.7 Using repository aliases . . . 20

3 Security **21**
3.1 How to set up security . . . 21
3.2 How to add and delete users . . . 21
3.3 Setting permissions for files and directories . . . 21
3.4 Groups of users can be assigned permissions . . . 22
3.5 Running CVSNT as a nonprivileged user . . . 23
3.6 Running within a chroot jail . . . 23
3.7 Setting and changing passwords . . . 23
3.8 Repository administrators . . . 23
3.9 Read-only repository access . . . 23
3.10 Temporary directories for the server . . . 24
3.11 The CVSNT lockserver . . . 24

4 Starting a project with CVS **25**
4.1 Setting up the files . . . 25
4.1.1 Creating a directory tree from a number of files . . . 25
4.1.2 Creating Files From Other Version Control Systems . . . 26
4.1.3 Creating a directory tree from scratch . . . 26
4.2 Defining the module . . . 26

5 Revisions **28**
5.1 Revision numbers . . . 28
5.2 Versions, revisions and releases . . . 28
5.3 Assigning revisions . . . 28
5.4 Tags-Symbolic revisions . . . 29
5.5 Specifying what to tag from the working directory . . . 30
5.6 Specifying what to tag by date or revision . . . 31
5.7 Deleting, moving, and renaming tags . . . 31
5.8 Tagging and adding and removing files . . . 32
5.9 Alias tags . . . 32
5.10 Commit identifiers . . . 32
5.11 Sticky tags . . . 32

6 Branching and merging **34**
6.1 What branches are good for . . . 34
6.2 Creating a branch . . . 34
6.3 Accessing branches . . . 35
6.4 Branches and revisions . . . 36
6.5 Magic branch numbers . . . 36
6.6 Merging an entire branch . . . 37
6.7 Merging from a branch several times . . . 38
6.8 Merging differences between any two revisions . . . 38
6.9 Merging can add or remove files . . . 39
6.10 Merging and keywords . . . 39

7 Recursive behavior **40**

8 Adding, removing, and renaming files and directories **41**
8.1 Adding files to a directory . . . 41
8.2 Removing files . . . 42
8.3 Removing directories . . . 43
8.4 Moving and renaming files . . . 43
8.4.1 The Normal way to Rename . . . 44
8.4.2 The old way to Rename . . . 44
8.4.3 Moving the history file . . . 44
8.4.4 Copying the history file . . . 45
8.5 Moving and renaming directories . . . 45

9 History browsing **46**
9.1 Log messages . . . 46
9.2 The history database . . . 46
9.3 User-defined logging . . . 46
9.3.1 The taginfo file . . . 46
9.4 Annotate command . . . 47

10 Handling binary files **48**
10.1 The issues with binary files . . . 48
10.2 How to store binary files . . . 48

11 Multiple developers **50**
11.1 File status . . . 50
11.2 Bringing a file up to date . . . 51
11.3 Conflicts example . . . 51
11.4 Informing others about commits . . . 53
11.5 Several developers simultaneously attempting to run CVS . . . 53
11.6 Mechanisms to track who is editing files . . . 54
11.6.1 Setting up cooperative edits . . . 54
11.6.2 Telling CVS to notify you when someone modifies a file . . . 55
11.6.3 How to edit a file which is being watched . . . 56
11.6.4 Information about who is watching and editing . . . 56
11.6.5 Using watches with old versions of CVS . . . 57
11.7 Choosing between reserved or unreserved checkouts . . . 57

12 Revision management **58**
12.1 When to commit? . . . 58

13 Keyword substitution **59**
13.1 Keyword List . . . 59
13.2 Using keywords . . . 60
13.3 Avoiding substitution . . . 60
13.4 Substitution modes . . . 60
13.5 Log . . . 62

14 Tracking third-party sources **63**
14.1 Importing for the first time . . . 63
14.2 Updating with the import command . . . 63
14.3 Reverting to the latest vendor release . . . 64
14.4 How to handle binary files with cvs import . . . 64
14.5 How to handle keyword substitution with cvs import . . . 64
14.6 Multiple vendor branches . . . 64

15 How your build system interacts with CVS **66**

16 Special Files **67**

A Guide to CVS commands **68**
A.1 Overall structure of CVS commands . . . 68
A.2 CVS's exit status . . . 68
A.3 Default options and the ~/.cvsrc and CVSROOT/cvsrc files . . . 69
A.4 Global options . . . 69

A.5 Common command options . . . 71
A.6 add--Add files to repository . . . 72
A.6.1 add options . . . 73
A.7 admin--Administration . . . 73
A.7.1 admin options . . . 73
A.8 annotate--find out who made changes to the files . . . 74
A.8.1 annotate options . . . 75
A.9 chacl--Change access control lists . . . 75
A.9.1 chacl options . . . 75
A.10 checkout--Check out sources for editing . . . 75
A.10.1 checkout options . . . 76
A.10.2 checkout examples . . . 77
A.11 chown--Change directory owner . . . 78
A.11.1 chown options . . . 78
A.12 commit--Check files into the repository . . . 78
A.12.1 commit options . . . 78
A.12.2 commit examples . . . 79
A.12.2.1 Committing to a branch . . . 79
A.12.2.2 Creating the branch after editing . . . 80
A.13 diff--Show differences between revisions . . . 80
A.13.1 diff options . . . 80
A.13.2 diff examples . . . 81
A.14 edit--Mark files for editing . . . 82
A.14.1 edit options . . . 82
A.15 editors--Find out who is editing a file . . . 83
A.15.1 editors options . . . 83
A.16 export--Export sources from CVS, similar to checkout . . . 83
A.16.1 export options . . . 84
A.17 history--Show status of files and users . . . 84
A.17.1 history options . . . 84
A.18 import--Import sources into CVS, using vendor branches . . . 86
A.18.1 import options . . . 86
A.18.2 import output . . . 87
A.18.3 import examples . . . 87
A.19 init--Initialise a new repository . . . 87
A.19.1 init options . . . 87
A.20 info--Get information about the client and server . . . 88
A.20.1 info options . . . 89
A.21 log--Print out log information for files . . . 89

A.21.1 log options . . . 89
A.21.2 log examples . . . 91
A.22 login--Cache a client password locally . . . 91
A.22.1 login options . . . 91
A.23 logout--Remove the cached entry for a password . . . 91
A.23.1 logout options . . . 91
A.24 ls--list modules, files and directories in the repository . . . 91
A.24.1 ls options . . . 92
A.25 lsacl--Show file/directory permissions . . . 92
A.25.1 lsacl options . . . 92
A.26 rlsacl--Show remote file/directory permissions . . . 93
A.27 passwd--Modify a user's password or create a user . . . 93
A.27.1 passwd options . . . 93
A.28 rannotate--Show who made changes to remote files . . . 93
A.29 rchacl--Change remote access control lists . . . 94
A.30 rchown--Change owner of a remote directory . . . 94
A.31 rdiff--'patch' format diffs between releases . . . 94
A.31.1 rdiff options . . . 94
A.31.2 rdiff examples . . . 95
A.32 release--Indicate that a Module is no longer in use . . . 95
A.32.1 release options . . . 96
A.32.2 release output . . . 96
A.32.3 release examples . . . 96
A.33 remove--Remove files from the working directory . . . 96
A.33.1 remove options . . . 96
A.34 rename--Rename files in the repository . . . 97
A.35 rlog--Return log history of remote file . . . 97
A.36 rtag--Mark a single revision over multiple files . . . 97
A.37 status--Display the state of a file in the working directory . . . 98
A.37.1 status options . . . 98
A.38 tag--Create a tag or branch . . . 98
A.38.1 tag options . . . 99
A.39 unedit--Mark edit as finished without committing . . . 99
A.39.1 unedit options . . . 100
A.40 update--Bring work tree in sync with repository . . . 100
A.40.1 update options . . . 100
A.40.2 update output . . . 102
A.41 version--Display client and server versions. . . . 102
A.41.1 version options . . . 102

A.42 watch--Watch for changes in a file . . . 103
A.42.1 watch options . . . 103
A.43 watchers--list watched files . . . 103
A.43.1 watchers options . . . 103
A.44 xdiff--External diff . . . 103
A.44.1 xdiff options . . . 104

B Reference manual for Administrative files 105

B.1 The modules file . . . 105
B.1.1 Alias modules . . . 105
B.1.2 Regular modules . . . 106
B.1.3 Ampersand modules . . . 106
B.1.4 Excluding directories . . . 107
B.1.5 Module options . . . 107
B.1.6 How the modules file "program options" programs are run . . . 108
B.2 The modules2 file . . . 108
B.2.1 How the modules2 file differs from the modules file . . . 108
B.2.2 Modules2 syntax . . . 108
B.3 The cvswrappers file . . . 109
B.3.1 default wrappers . . . 110
B.4 The commit support files . . . 110
B.4.1 The common syntax . . . 111
B.5 Triggers . . . 112
B.6 Commitinfo . . . 113
B.7 Verifying . . . 113
B.8 Loginfo . . . 114
B.8.1 Loginfo example . . . 115
B.8.2 Loginfo default standard input format . . . 115
B.9 Precommand . . . 116
B.10 postcommand . . . 116
B.11 premodule . . . 116
B.12 postmodule . . . 117
B.13 postcommit . . . 117
B.14 historyinfo . . . 117
B.15 rcsinfo . . . 117
B.16 notify . . . 118
B.17 keywords . . . 118
B.17.1 Storing user defined information using keywords . . . 119
B.18 Email notification . . . 119

B.18.1 Configure the commit support files . . . 120
B.18.2 Write the template . . . 120
B.18.3 Configure the server . . . 120
B.18.4 Keywords used in template files . . . 121
B.18.4.1 commit emails . . . 121
B.18.4.2 tag emails . . . 121
B.18.4.3 notify emails . . . 121
B.19 Ignoring files via cvsignore . . . 122
B.20 The checkoutlist file . . . 122
B.21 The history file . . . 123
B.22 The shadow file . . . 123
B.22.1 Keeping a checked out copy . . . 123
B.23 ActiveScript support . . . 123
B.24 Expansions in administrative files . . . 124
B.25 The CVSROOT/config configuration file . . . 125
B.26 The server configuration files . . . 126

C All environment variables which affect CVS 127

D Compatibility between CVS Versions 129

E Troubleshooting 130
E.1 Partial list of error messages . . . 130
E.2 Trouble making a connection to a CVS server . . . 133

F Credits 134

G Dealing with bugs in CVS or this manual 135

17 Index 137

Abstract

This is the open source reference manual for CVSNT version 2.8.01.9054 Published .

Other documents available are:

- the eBook: All About CVS is included with CVS Suite in the Documentation menu [*download (login required)*] [*about*].
- Download Free Documentation: *Windows Installation Guide for CVS Suite 2009R2* PDF.
- Download Free Documentation: *Red Hat Enterprise Linux 7 Installation Guide for CVS Suite 2009R2* PDF.
- Download Free Documentation: *Windows Bugzilla/CVSWEB Install Guide* PDF.
- Download Free Documentation: *Using CVS Suite with Eclipse* PDF.
- Download Free Documentation: *Using CVS Suite with IntelliJ IDEA* PDF.
- Download Free Documentation: *Using CVS Suite with SQL Navigator* PDF.
- Download Free Documentation: *Implementing Workflow and Defect Tracking Integration with CVS Suite* PDF.
- Download Free Documentation: *Implementing Promotion Models with CVS Suite* PDF.
- Video Documentation: *Introduction to CVS Suite Studio* YouTube.
- Video Documentation: *March Hare Software Video Channel* YouTube.
- Download Release Notes: *CVSNT 2.5.03 to CVS Suite 2009R2* PDF.
- Download Release Notes: *CVS Suite 2008 to CVS Suite 2009R2* PDF.

Chapter 1

Overview

This chapter is for people who have never used cvsnt, and perhaps have never used version control software before.

If you are already familiar with cvsnt and are just trying to learn a particular feature or remember a certain command, you can probably skip everything here.

1.1 What is CVS?

cvsnt 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 cvsnt, 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. cvsnt stores all the versions of a file in a single file in a clever way that only stores the differences between versions.

cvsnt 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. cvsnt solves this problem by insulating the different developers from each other. Every developer works in his own directory, and cvsnt merges the work when each developer is done.

cvsnt started out as a bunch of shell scripts written by Dick Grune, posted to the newsgroup **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 cvsnt much of the cvsnt 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.

In December, 1999 Tony Hoyle converted the unix based CVS to run under Windows Server (Windows NT). This later became cvsnt, which developed into a project of its own.

CVSNT is now a major project with solid commercial backing, and an active support community.

You can get cvsnt in a variety of ways, including free download from the internet. For more information on downloading cvsnt and other cvsnt topics, see:

```
http://www.cvsnt.org/
```

or

```
http://www.march-hare.com/cvspro/
```

.

For support please contact *sales@march-hare.com*. The mailing list is no longer used for support, but the history is maintained online as well as the current bug database:

```
http://www.cvsnt.org/pipermail/cvsnt/
http://www.cvsnt.org/tt/
http://www.march-hare.com/pipermail/cvsnt/
http://customer.march-hare.com/webtools/bugzilla/tt.htm
```

1.2 What is CVS not?

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

cvsnt 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.

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

cvsnt 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. cvsnt 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 cvsnt.

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

See Chapter 15, for more information on doing builds in conjunction with cvsnt.

cvsnt 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, cvsnt can't help.

cvsnt 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.

cvsnt 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.

cvsnt 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.

cvsnt 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 cvsnt's competence.

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

cvsnt does not have change control (but it comes close) 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 cvsnt to an external bug-tracking system, see the **rcsinfo** and **verifymsg** files (Appendix B).

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, cvsnt marks that commit with a session identifier or **commitid**.

cvsnt is also able to group a set of commits under a logical group by its group identifier, also known as the **bugid**. You can also selectively merge changes based on this identifier.

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 cvsnt 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 cvsnt to make sure nothing falls through the cracks.

cvsnt is not an automated testing program It is possible to link into automated testing scripts using the **postcommit**and **trigger** functionality. This is outside the scope of this manual however.

cvsnt 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 cvsnt but it might be a little more work. In some cases you'll want to use the **commitinfo**, **loginfo**, **rcsinfo**, or **verifymsg** 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.

1.3 A sample session

As a way of introducing cvsnt, we'll go through a typical work-session using cvsnt. The first thing to understand is that cvsnt stores all files in a centralized *repository* (Chapter 2); this section assumes that a repository is set up.

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**.

1.3.1 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 (the commands used may be slightly different on Windows machines, but the output is the same).

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

The **CVS** directory is used internally by cvsnt (on Windows clients it is normally hidden). 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, sccs, visual studio, pvcs and users of all similar scm tools: Although you can lock the files that you want to edit in CVS, there is no need to. Chapter 11, for an explanation.

1.3.2 Committing your changes

When you have checked that the compiler is still compilable you decide to make a new version of **backend.c**. This will store your new **backend.c** in the repository and make it available to anyone else who is using that same repository.

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

cvsnt 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 there is a default which will vary with your operating system, for example **vi** for unix or **notepad** for Windows.

When cvsnt starts the editor, it includes a list of files which are modified. For the cvsnt client, this list is based on comparing the modification time of the file against the modification time that the file had when it was last gotten or updated. Therefore, if a file's modification time has changed but its contents have not, it will show up as modified. The simplest way to handle this is simply not to worry about it--if you proceed with the commit cvsnt will detect that the contents are not modified and treat it as an unmodified file. The next **update** will clue cvsnt in to the fact that the file is unmodified, and it will reset its stored timestamp so that the file will not show up in future editor sessions.

If you want to avoid 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
```

1.3.3 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 (Section A.32):

```
$ 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) directory '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. Section B.21.

When you use the **-d** flag with **release**, it also removes your working copy. The **-f** tells cvsnt to also delete unknown files (such as object files).

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

1.3.4 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) directory ‘tc’: y
```

Chapter 2

The Repository

The cvsnt *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 cvsnt commands to get your own copy of the files into a *working directory*, 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. Note that the repository is not a subdirectory of the working directory, or vice versa; they should be in separate locations.

cvsnt 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 cvsnt. For information on other access methods, see Section 2.9.

If the access method is omitted, then if the repository does not contain @, then **:local:** is assumed. If it does contain @ then **:ext:** 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**.

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

2.1 Telling CVS where your repository is

There are several ways to tell cvsnt 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**, **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
```

It is common for cvs frontends to set this up automatically. On most frontends there will be a dialog box which prompts you for the CVSROOT when it is first configured.

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. Of course, for proper operation they should be two ways of referring to the same repository.

2.2 How data is stored in the repository

For most purposes it isn't important *how* cvsnt 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 cvsnt commands, such changes need not be disruptive.

However, in some cases it may be necessary to understand how cvsnt stores data in the repository, for example you might need to track down cvsnt locks (Section 11.5) or you might need to deal with the file permissions appropriate for the repository.

2.2.1 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)
          |   |
          |   +--cvsnt
          |       (source code to cvsnt)
          |
          +--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 [**?**], distributed with rcs, or the file **doc/rcsfile** in the cvsnt source distribution. This file format has become very common--many systems other than cvsnt or rcs can at least import history files in this format.

The rcs files used in cvs and cvsnt differ in a few ways from the standard format. The biggest difference in cvs is magic branches; for more information see Section 6.5. Also in cvsnt the valid tag names are a subset of what rcs accepts; for cvsnt's rules see Section 5.4. cvsnt also brings binary diffs and mergepoints to the table. Future versions of cvsnt may introduce still further changes, so it is unwise to try to read (or write to) the repository with rcs. cvsnt provides some rcs 'lookalike' comands for accessing the repository files.

2.2.2 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. On Unix, this normally means that you must create a 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. On Windows, you must allow write access to the files for each user or group that is accessing the repository. If impersonation is not enabled, then the repository is always accessed as SYSTEM.

This means that you can only control access to files on a per-directory basis using the operating system (however see the chacl and lsacl commands for a way to do this withing cvsnt itself).

Note that users must also have write access to check out files, because cvsnt needs to create lock files (Section 11.5).

Also note that users must have write access to the **CVSROOT/val-tags** file. cvsnt 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).

Normally each rcs file will be owned by the user who last checked it in. This has little significance; what really matters is who owns the directories. See also Section 3.5.

cvsnt 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 cvsnt 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 cvsnt creates them read-only (see the sections on watches, Section 11.6.1; -r, Section A.4; or **CVSREAD**, Appendix C).

Note that using the client/server cvsnt (Section 2.9), there is no good way to set **CVSUMASK**; the setting on the client machine has no effect. If you are connecting with **ssh**, you can set **CVSUMASK** in **.bashrc** or **.cshrc**, as described in the documentation for your operating system. This behavior might change in future versions of cvsnt; do not rely on the setting of **CVSUMASK** on the client having no effect.

Under Windows, because of the way directory permissions work on that platform, setting CVSUMASK will have no effect.

Using remote repositories, you will generally need stricter permissions on the cvsroot directory and directories above it in the tree; see Section 2.9.4.4.

Some operating systems have features which allow a particular program to run with the ability to perform operations which the caller of the program could not. For example, the set user ID (setuid) or set group ID (setgid) features of unix or the installed image feature of VMS. cvsnt was not written to use such features and therefore attempting to install cvsnt in this fashion will provide protection against only accidental lapses; anyone who is trying to circumvent the measure will be able to do so, and depending on how you have set it up may gain access to more than just cvsnt. You may wish to instead consider pserver or sserver. They shares some of the same attributes, in terms of possibly providing a false sense of security or opening security holes wider than the ones you are trying to fix, so read the documentation on pserver security carefully if you are considering this option (Section 2.9.4.4).

2.2.3 The attic

The attic was used in older versions of cvs to store files in the branches. Its use has been depreciated since cvsnt 2.0.15, and cvsnt no longer stores files in the **Attic**. It will, however, read files that have been stored in the **Attic** by previous versions of cvs.

instead. It should not matter from a user point of view whether a file is in the attic; cvsnt keeps track of this and looks in the attic when it needs to. But in case you want to know, the rule was that the rcs file is stored in the attic if and only if the head revision on the trunk has state **dead**. A **dead** state means that file has been removed, or never added, for that revision. For example, if you add a file on a branch, it will have a trunk revision in **dead** state, and a branch revision in a non-**dead** state.

2.2.4 The CVS directory in the repository

The **CVS** directory in each repository directory contains information such as file attributes (in a file called **CVS/fileattr.xml**. In the future additional files may be added to this directory, so implementations should silently ignore additional files.

The format of the fileattr.xml file is a series of XML entries describing the edit state of each file, and any access permissions that are current.

2.2.5 CVS locks in the repository

For an introduction to cvsnt locks focusing on user-visible behavior, see Section 11.5. The following section is aimed at people who are writing tools which want to access a cvsnt repository without interfering with other tools acessing the same repository. If you find yourself confused by concepts described here, like *read lock*, *write lock*, and *deadlock*, you might consult the literature on operating systems or databases.

cvsnt now uses the **LockServer** to handle lock concurrency in a dynamic way (see Section 3.11. This following section refers to the obsolete filesysem lock method, which may still be in use on some sites.

Any file in the repository with a name starting with **#cvs.rfl.** is a read lock. Any file in the repository with a name starting with **#cvs.wfl** is a write lock. Old versions of cvsnt (before cvsnt 1.5) also created files with names starting with **#cvs.tfl**, but they are not discussed here. The directory **#cvs.lock** serves as a master lock. That is, one must obtain this lock first before creating any of the other locks.

To obtain a readlock, first create the **#cvs.lock** directory. This operation must be atomic (which should be true for creating a directory under most operating systems). If it fails because the directory already existed, wait for a while and try again. After obtaining the **#cvs.lock** lock, create a file whose name is **#cvs.rfl.** followed by information of your choice (for example, hostname and process identification number). Then remove the **#cvs.lock** directory to release the master lock. Then proceed with reading the repository. When you are done, remove the **#cvs.rfl** file to release the read lock.

To obtain a writelock, first create the **#cvs.lock** directory, as with a readlock. Then check that there are no files whose names start with **#cvs.rfl.**. If there are, remove **#cvs.lock**, wait for a while, and try again. If there are no readers, then create a file whose name is **#cvs.wfl** followed by information of your choice (for example, hostname and process identification number). Hang on to the **#cvs.lock** lock. Proceed with writing the repository. When you are done, first remove the **#cvs.wfl** file and then the **#cvs.lock** directory. Note that unlike the **#cvs.rfl** file, the **#cvs.wfl** file is just informational; it has no effect on the locking operation beyond what is provided by holding on to the **#cvs.lock** lock itself.

Note that each lock (writelock or readlock) only locks a single directory in the repository, including **Attic** and **CVS** but not including subdirectories which represent other directories under version control. To lock an entire tree, you need to lock each

directory (note that if you fail to obtain any lock you need, you must release the whole tree before waiting and trying again, to avoid deadlocks).

Note also that cvsnt expects writelocks to control access to individual **foo,v** files. rcs has a scheme where the **,foo,** file serves as a lock, but cvsnt does not implement it and so taking out a cvsnt writelock is recommended. See the comments at rcs_internal_lockfile in the cvsnt source code for further discussion/rationale.

2.2.6 How files are stored in the CVSROOT directory

The **$CVSROOT/CVSROOT** directory contains the various administrative files. In some ways this directory is just like any other directory in the repository; it contains rcs files whose names end in **,v**, and many of the cvsnt commands operate on it the same way. However, there are a few differences.

For each administrative file, in addition to the rcs file, there is also a checked out copy of the file. For example, there is an rcs file **loginfo,v** and a file **loginfo** which contains the latest revision contained in **loginfo,v**. When you check in an administrative file, cvsnt should print

```
cvs commit: Rebuilding administrative file database
```

and update the checked out copy in **$CVSROOT/CVSROOT**. If it does not, there is something wrong (Appendix G). To add your own files to the files to be updated in this fashion, you can add them to the **checkoutlist** administrative file (Section B.20).

By default, the **modules** file behaves as described above. If the modules file is very large, storing it as a flat text file may make looking up modules slow (I'm not sure whether this is as much of a concern now as when cvsnt first evolved this feature; I haven't seen benchmarks). Therefore, by making appropriate edits to the cvsnt source code one can store the modules file in a database which implements the **ndbm** interface, such as Berkeley db or GDBM. If this option is in use, then the modules database will be stored in the files **modules.db**, **modules.pag**, and/or **modules.dir**.

For information on the meaning of the various administrative files, see Appendix B.

2.3 How data is stored in the working directory

While we are discussing cvsnt internals which may become visible from time to time, we might as well talk about what cvsnt puts in the **CVS** directories in the working directories. As with the repository, cvsnt handles this information and one can usually access it via cvsnt commands. But in some cases it may be useful to look at it, and other programs, such as the **jCVS** graphical user interface or the **VC** package for emacs, may need to look at it. Such programs should follow the recommendations in this section if they hope to be able to work with other programs which use those files, including future versions of the programs just mentioned and the command-line cvsnt client.

The **CVS** directory contains several files. Programs which are reading this directory should silently ignore files which are in the directory but which are not documented here, to allow for future expansion.

The files are stored according to the text file convention for the system in question. This means that working directories are not portable between systems with differing conventions for storing text files. This is intentional, on the theory that the files being managed by cvsnt probably will not be portable between such systems either.

Root This file contains the current cvsnt root, as described in Section 2.1.

Repository This file contains the directory within the repository which the current directory corresponds with. It can be either an absolute pathname or a relative pathname; cvsnt has had the ability to read either format since at least version 1.3 or so. The relative pathname is relative to the root, and is the more sensible approach, but the absolute pathname is quite common and implementations should accept either. For example, after the command

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

Root will contain

```
:local:/usr/local/cvsroot
```

and **Repository** will contain either

```
/usr/local/cvsroot/yoyodyne/tc
```

or

```
yoyodyne/tc
```

If the particular working directory does not correspond to a directory in the repository, then **Repository** should contain **CVSROOT/Emptydir**.

Entries This file lists the files and directories in the working directory. The first character of each line indicates what sort of line it is. If the character is unrecognized, programs reading the file should silently skip that line, to allow for future expansion.

If the first character is **/**, then the format is:

```
/name/revision/timestamp[+conflict]/options/tagdate
```

where **[** and **]** are not part of the entry, but instead indicate that the **+** and conflict marker are optional. `name` is the name of the file within the directory. `revision` is the revision that the file in the working derives from, or **0** for an added file, or **-** followed by a revision for a removed file. `timestamp` is the timestamp of the file at the time that cvsnt created it; if the timestamp differs with the actual modification time of the file it means the file has been modified. It is stored in the format used by the ISO C asctime() function (for example, **Sun Apr 7 01:29:26 1996**). One may write a string which is not in that format, for example, **Result of merge**, to indicate that the file should always be considered to be modified. This is not a special case; to see whether a file is modified a program should take the timestamp of the file and simply do a string compare with `timestamp`. If there was a conflict, `conflict` can be set to the modification time of the file after the file has been written with conflict markers (Section 11.3). Thus if `conflict` is subsequently the same as the actual modification time of the file it means that the user has obviously not resolved the conflict. `options` contains sticky options (for example **-kb** for a binary file). `tagdate` contains **T** followed by a tag name, or **D** for a date, followed by a sticky tag or date. Note that if `timestamp` contains a pair of timestamps separated by a space, rather than a single timestamp, you are dealing with a version of cvsnt earlier than cvsnt 1.5 (not documented here).

The timezone on the timestamp in CVS/Entries (local or universal) should be the same as the operating system stores for the timestamp of the file itself. For example, on Unix the file's timestamp is in universal time (UT), so the timestamp in CVS/Entries should be too. On vms, the file's timestamp is in local time, so cvsnt on vms should use local time. This rule is so that files do not appear to be modified merely because the timezone changed (for example, to or from summer time).

If the first character of a line in **Entries** is **D**, then it indicates a subdirectory. **D** on a line all by itself indicates that the program which wrote the **Entries** file does record subdirectories (therefore, if there is such a line and no other lines beginning with **D**, one knows there are no subdirectories). Otherwise, the line looks like:

```
D/name/filler1/filler2/filler3/filler4
```

where `name` is the name of the subdirectory, and all the `filler` fields should be silently ignored, for future expansion. Programs which modify **Entries** files should preserve these fields.

The lines in the **Entries** file can be in any order.

Entries.Log This file does not record any information beyond that in **Entries**, but it does provide a way to update the information without having to rewrite the entire **Entries** file, including the ability to preserve the information even if the program writing **Entries** and **Entries.Log** abruptly aborts. Programs which are reading the **Entries** file should also check for **Entries.Log**. If the latter exists, they should read **Entries** and then apply the changes mentioned in **Entries.Log**. After applying the changes, the recommended practice is to rewrite **Entries** and then delete **Entries.Log**. The format of a line in **Entries.Log** is a single character command followed by a space followed by a line in the format specified for a line in **Entries**. The single character command is **A** to indicate that the entry is being added, **R** to indicate that the entry is being removed, or any other character to indicate that the entire line in **Entries.Log** should be silently ignored (for future expansion). If the second character of the line in **Entries.Log** is not a space, then it was written by an older version of cvsnt (not documented here).

Programs which are writing rather than reading can safely ignore **Entries.Log** if they so choose.

Entries.Backup This is a temporary file. Recommended usage is to write a new entries file to **Entries.Backup**, and then to rename it (atomically, where possible) to **Entries**.

Entries.Old This is the previous generation of the 'Entries' file. If your program modifies the 'Entries' file rename the existing file to 'Entries.Old' so that frontends are able to find out what has changed.

Entries.Static The only relevant thing about this file is whether it exists or not. If it exists, then it means that only part of a directory was gotten and cvsnt will not create additional files in that directory. To clear it, use the **update** command with the **-d** option, which will get the additional files and remove **Entries.Static**.

Entries.Extra This holds extra information about the file that was not recorded by the original CVS client. If this file exists there is a line for each file in the Entries file. The lines can be in any order.

The format of the lines is:

/name/saved mergepoint/filler1/rcstime/edit_revision/edit_tag/edit_bugid/

If there are any extra fields after rcstime these should be ignored.

The second field is the saved tag from an update -j, which is recorded the next time the file is committed to store the mergepoint data.

The third field is unused (and reserved) at present.

The fourth field is the last checkin time of the file, expressed as a time_t. Do not assume that this value will fit into a 32bit variable, as this will cause problems in 2038.

The fifth, sixth and seventh fields define the revision, tag/branch and bug identifier of the last **cvs edit** that was invoked by the client.

Entries.Extra.Old This is the previous generation of the 'Entries.Extra' file. If your program modifies the 'Entries' file rename the existing file to 'Entries.Extra.Old' so that frontends are able to find out what has changed.

Rename This file contains information about any renames that have not yet been committed to the repository. The file is stored as pairs of filenames within the directory, with a blank meaning 'removed'.

Tag This file contains per-directory sticky tags or dates. The first character is **T** for a branch tag, **N** for a non-branch tag, or **D** for a date, or another character to mean the file should be silently ignored, for future expansion. This character is followed by the tag or date. Note that per-directory sticky tags or dates are used for things like applying to files which are newly added; they might not be the same as the sticky tags or dates on individual files. For general information on sticky tags and dates, see Section 5.11.

CVSNT also stores the directory revision in this file, as a numeric tag.

Notify This file stores notifications (for example, for **edit** or **unedit**) which have not yet been sent to the server. Its format is not yet documented here.

Notify.tmp This file is to **Notify** as **Entries.Backup** is to **Entries**. That is, to write **Notify**, first write the new contents to **Notify.tmp** and then (atomically where possible), rename it to **Notify**.

Base If watches are in use, then an **edit** command stores the original copy of the file in the **Base** directory. This allows the **unedit** command to operate even if it is unable to communicate with the server.

Template This file contains the template specified by the **rcsinfo** file (Section B.15). It is only used by the client; the non-client/server cvsnt consults **rcsinfo** directly.

2.4 The administrative files

The directory **$CVSROOT/CVSROOT** contains some *administrative files*. Appendix B, for a complete description. You can use cvsnt 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. Section B.1, for a full explanation of all the available features.

In many cases the **modules2** file may be more suitable for defining modules. See Section B.2 for details.

2.4.1 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.

2.5 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 cvsnt, or (once you have checked out a working directory) by simply allowing cvsnt to use the repository that was used to check out the working directory (Section 2.1).

The big advantage of having multiple repositories is that they can reside on different servers. With CVS version 1.10, a single command cannot recurse into directories from different repositories. With development versions of cvsnt, you can check out code from multiple servers into your working directory. cvsnt will recurse and handle all the details of making connections to as many server machines as necessary to perform the requested command. Here is an example of how to set up a working directory:

```
cvs -d server1:/cvs co dir1
cd dir1
cvs -d server2:/root co sdir
cvs update
```

The **cvs co** commands set up the working directory, and then the **cvs update** command will contact server2, to update the dir1/sdir subdirectory, and server1, to update everything else.

2.6 Creating a repository

To set up a cvsnt repository, first choose the machine and disk on which you want to store the revision history of the source files. CPU and memory requirements are modest, so most machines should be adequate. For details see Section 2.9.1.

To estimate disk space requirements, if you are importing rcs files from another system, the size of those files is the approximate initial size of your repository, or if you are starting without any version history, a rule of thumb is to allow for the server approximately three times the size of the code to be under cvsnt for the repository (you will eventually outgrow this, but not for a while). On the machines on which the developers will be working, you'll want disk space for approximately one working directory for each developer (either the entire tree or a portion of it, depending on what each developer uses).

The repository should be accessible (directly or via a networked file system) from all machines which want to use cvsnt in server or local mode; the client machines need not have any access to it other than via the cvsnt protocol. It is not possible to use cvsnt to read from a repository which one only has read access to; cvsnt needs to be able to create lock files (Section 11.5).

To create a repository, run the **cvs init** command. It will set up an empty repository in the cvsnt root specified in the usual way (Chapter 2). 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.

2.7 Backing up a repository

There is nothing particularly magical about the files in the repository; for the most part it is possible to back them up just like any other files. However, there are a few issues to consider.

The first is that to be paranoid, one should either not use cvsnt during the backup, or have the backup program lock cvsnt while doing the backup. To not use cvsnt, you might forbid logins to machines which can access the repository, turn off your cvsnt server, or similar mechanisms. The details would depend on your operating system and how you have cvsnt set up. To lock cvsnt, you would create **#cvs.rfl** locks in each repository directory. See Section 11.5, for more on cvsnt locks. Having said all this, if you just back up without any of these precautions, the results are unlikely to be particularly dire. Restoring from backup, the repository might be in an inconsistent state, but this would not be particularly hard to fix manually.

When you restore a repository from backup, assuming that changes in the repository were made after the time of the backup, working directories which were not affected by the failure may refer to revisions which no longer exist in the repository. Trying to run cvsnt in such directories will typically produce an error message. One way to get those changes back into the repository is as follows:

- Get a new working directory.
- Copy the files from the working directory from before the failure over to the new working directory (do not copy the contents of the **CVS** directories, of course).
- Working in the new working directory, use commands such as **cvs update** and **cvs diff** to figure out what has changed, and then when you are ready, commit the changes into the repository.

2.8 Moving a repository

Just as backing up the files in the repository is pretty much like backing up any other files, if you need to move a repository from one place to another it is also pretty much like just moving any other collection of files.

The main thing to consider is that working directories point to the repository. The simplest way to deal with a moved repository is to just get a fresh working directory after the move. Of course, you'll want to make sure that the old working directory had been checked in before the move, or you figured out some other way to make sure that you don't lose any changes. If you really do want to reuse the existing working directory, it should be possible with manual surgery on the **CVS/Repository** files. You can see Section 2.3, for information on the **CVS/Repository** and **CVS/Root** files, but unless you are sure you want to bother, it probably isn't worth it.

2.9 Remote repositories

Your working copy of the sources can be on a different machine than the repository. Using cvsnt in this manner is known as *client/server* operation. You run cvsnt on a machine which can mount your working directory, known as the *client*, and tell it to communicate to a machine which can mount the repository, known as the *server*. Generally, using a remote repository is just like using a local one, except that the format of the repository name is:

```
:method[;keywords...]:[[user][:password]@]hostname[:[port]][:]/path/to/repository
```

Specifying a password in the repository name is not recommended during checkout, since this will cause cvsnt to store a cleartext copy of the password in each created directory. **cvs login** first instead (Section 2.9.4.3).

With most protocols the username is optional. Your current login name will be used in this case. Also, a special username of '.' (dot) can be used, and means the same as not specifying a username. This can be used for frontends built for older cvs versions which required a username to be specified.

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 2.9.2.

2.9.1 Server requirements

The quick answer to what sort of machine is suitable as a server is that requirements are modest--a server with 32M of memory or even less can handle a fairly large source tree with a fair amount of activity.

The real answer, of course, is more complicated. Estimating the known areas of large memory consumption should be sufficient to estimate memory requirements. There are two such areas documented here; other memory consumption should be small by comparison (if you find that is not the case, let us know, as described in Appendix G, so we can update this documentation).

The first area of big memory consumption is large checkouts, when using the cvsnt server. The server consists of two processes for each client that it is serving. Memory consumption on the child process should remain fairly small. Memory consumption on the parent process, particularly if the network connection to the client is slow, can be expected to grow to slightly more than the size of the sources in a single directory, or two megabytes, whichever is larger.

Multiplying the size of each cvsnt server by the number of servers which you expect to have active at one time should give an idea of memory requirements for the server. For the most part, the memory consumed by the parent process probably can be swap space rather than physical memory.

The second area of large memory consumption is **diff**, when checking in large files. This is required even for binary files. The rule of thumb is to allow about ten times the size of the largest file you will want to check in, although five times may be adequate. For example, if you want to check in a file which is 10 megabytes, you should have 100 megabytes of memory on the machine doing the checkin (the server machine for client/server, or the machine running cvsnt for non-client/server). This can be swap space rather than physical memory. Because the memory is only required briefly, there is no particular need to allow memory for more than one such checkin at a time.

Resource consumption for the client is even more modest--any machine with enough capacity to run the operating system in question should have little trouble.

For information on disk space requirements, see Section 2.6.

2.9.2 Connecting with ssh

cvsnt uses the **ssh** 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 **toe.example.com**, and the server machine is **faun.example.org**. On faun, put the following line into the file **.rhosts** in **bach**'s home directory:

```
toe.example.com  mozart
```

Then test that **ssh** is working with

```
ssh -l bach faun.example.org 'echo $PATH'
```

On some versions of cvsnt :ssh: protocol is available. This is a builtin ssh client which integrates cvs authentication with ssh security.

At its simplest, this is used like :pserver:, as in:

```
cvs -d :ssh:user@machine.example.org:/usr/local/cvs login
Password: *****
cvs -d :ssh:user@machine.example.org:/usr/local/cvs co myproject
```

However you can also register your private key with cvs, which it will use instead of a password:

```
cvs -d :ssh;key='c:\user.ppk':user@machine.example.org:/usr/local/cvs login
Password: ******** (if your key has no passphrase, just press enter here)
cvs -d :ssh:user@machine.example.org:/usr/local/cvs co myproject
```

The keys should be in the putty private key format. You can use PuttyGen to convert an existing Openssh private key to this format.

:ext: specifies an external ssh program. By default this is **ssh** but you may specify another program in the CVSROOT using the optional :ext:{program} command. You may alternatively set the **CVS_EXT** 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

Continuing our example, supposing you want to access the module **foo** in the repository **/usr/local/cvsroot/**, on machine **faun.example.org**, you are ready to go:

```
cvs -d :ext:bach@faun.example.org/usr/local/cvsroot checkout foo
```

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

2.9.3 Using 3rd party clients via the extnt wrapper

(Windows only at present) The extnt.exe program is a wrapper client which allows 3rd-party clients to use CVSNT protocols to access a CVSNT server. It uses the :ext: protocol on the client so should be compatible with all existing clients.

The program takes a number of optional parameters, which may or may not be defined by the client. A standard cvs client will only usually pass the -l (username) option.

-l username Username to use.

-p protocol protocol to use. Default comes from the protocol= line in extnt.ini. If that is absent uses sspi.

-d directory repository directory. Default comes from the directory= line in extnt.ini

-P password Password to use. Default comes from the password= line in extnt.ini

Unless all parameters are passed on the command line by the client, you need to setup extnt.ini with the correct details. This file is laid out as a standard windows .ini file, with the section name based on the hostname to connect to.

```
[cvs.myserver.org]
protocol=sspi
directory=/cvs
```

You can define multiple connections to the same host by using the hostname= entry, eg:

```
[cvs-1]
protocol=sspi
directory=/cvs-repo-1
hostname=cvs.myserver.org

[cvs-2]
protocol=sspi
directory=/cvs-repo-2
hostname=cvs.myserver.org
```

Configuring the client to call extnt.exe as its :ext: application is client specific.

2.9.4 Direct connection with password authentication

The cvsnt client can also connect to the server using a password protocol. This is particularly useful if using **ssh** 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.

2.9.4.1 Setting up the server for Authentication

First of all, you probably want to tighten the permissions on the **$CVSROOT** and **$CVSROOT/CVSROOT** directories. See Section 2.9.4.4, for more details.

On Windows Server (2003, 2008, 2012 etc.), on the server side, you must run the cvsmanager.exe program which calls the cvs.exe when required. Setup is done using the cvsnt server control panel. the rest of this chapter is mostly Unix related.

On Unix, on the server side, the file **/etc/cvsnt/PServer**, **/etc/cvsnt/Plugins**, and **/etc/cvsnt/server** need to be edited so **cvsmanager** knows the location of your repositories and other parameters (eg: connection port etc.).

Older versions of CVSNT used **/etc/inetd.conf** or **/etc/xinetd.conf** for this purpose - these are no longer used with CVSNT 2.8 and later.

By default, the port number is for pserver is 2401 it would be different for pserver if your client were compiled with **CVS_AUTH_PORT** defined to something else, though. This can also be sepcified in the CVSROOT variable (Section 2.9) or overridden with the CVS_CLIENT_PORT environment variable (Appendix C), or, on Windows Server, set in the cvsnt server control panel.

You can use the **/etc/cvsnt/PServer** file to specify a temporary directory, or, on Windows Server set this within the cvsnt server control panel.

If you are having trouble setting this up, see Section E.2.

2.9.4.2 CVS passwd file

Because the client stores and transmits passwords in cleartext (almost--see Section 2.9.4.4, for details), a separate cvsnt password file is generally used, so people don't compromise their regular passwords when they access the repository. This file is **$CVSROOT/CVSROOT/passwd** (Section 2.4). It uses a colon-separated format, similar to **/etc/passwd** on Unix systems, except that it has fewer fields: cvsnt username, optional password, and an optional system username for cvsnt to run as if authentication succeeds. Here is an example **passwd** file with five entries:

```
anonymous:
bach:ULtgRLXo7NRxs
spwang:1sOp854gDF3DY
melissa:tGX1fS8sun6rY:pubcvs
qproj:XR4EZcEs0szik:pubcvs
```

(The passwords are encrypted according to the standard Unix **crypt()** function, so it is possible to paste in passwords directly from regular Unix **/etc/passwd** files.)

The first line in the example will grant access to any cvsnt client attempting to authenticate as user **anonymous**, no matter what password they use, including an empty password. (This is typical for sites granting anonymous read-only access; for information on how to do the "read-only" part, see Section 3.9.)

The second and third lines will grant access to **bach** and **spwang** if they supply their respective plaintext passwords.

The fourth line will grant access to **melissa**, if she supplies the correct password, but her cvsnt operations will actually run on the server side under the system user **pubcvs**. Thus, there need not be any system user named **melissa**, but there *must* be one named **pubcvs**.

The fifth line shows that system user identities can be shared: any client who successfully authenticates as **qproj** will actually run as **pubcvs**, just as **melissa** does. That way you could create a single, shared system user for each project in your repository, and give each developer their own line in the **$CVSROOT/CVSROOT/passwd** file. The cvsnt username on each line would be different, but the system username would be the same. The reason to have different cvsnt usernames is that cvsnt will log their actions under those names: when **melissa** commits a change to a project, the checkin is recorded in the project's history under the name **melissa**, not **pubcvs**. And the reason to have them share a system username is so that you can arrange permissions in the relevant area of the repository such that only that account has write-permission there.

If the system-user field is present, all password-authenticated cvsnt commands run as that user; if no system user is specified, cvsnt simply takes the cvsnt username as the system username and runs commands as that user. In either case, if there is no such user on the system, then the cvsnt operation will fail (regardless of whether the client supplied a valid password).

The password and system-user fields can both be omitted (and if the system-user field is omitted, then also omit the colon that would have separated it from the encrypted password). For example, this would be a valid **$CVSROOT/CVSROOT/passwd** file:

```
anonymous::pubcvs
fish:rKa5jzULzmhOo:kfogel
sussman:1sOp854gDF3DY
```

When the password field is omitted or empty, then the client's authentication attempt will succeed with any password, including the empty string. However, the colon after the cvsnt username is always necessary, even if the password is empty.

cvsnt can also fall back to use system authentication. When authenticating a password, the server first checks for the user in the **$CVSROOT/CVSROOT/passwd** file. If it finds the user, it will use that entry for authentication as described above. But if it does not find the user, or if the cvsnt **passwd** file does not exist, then the server can try to authenticate the username and password using the operating system's user-lookup routines (this "fallback" behavior can be disabled by setting **SystemAuth=no** in the cvsnt **config** file, Section B.25). Be aware, however, that falling back to system authentication might be a security risk: cvsnt operations would then be authenticated with that user's regular login password, and the p assword flies across the network in plaintext. See Section 2.9.4.4 for more on this.

You can setup the passwd file by logging in to cvs using another method (local, sserver, gserver, ssh, sspi) and using the cvsnt passwd command to add new users.

2.9.4.3 Using the client with password authentication

To run a cvsnt command on a remote repository via the password-authenticating server, one specifies the protocol, optional username, repository host, an optional port number, and path to the repository. For example:

```
cvs -d :pserver:faun.example.org:/usr/local/cvsroot checkout someproj

cvs -d :sserver:faun.example.org:/usr/local/cvsroot checkout someproj

cvs -d :sspi:faun.example.org:/usr/local/cvsroot checkout someproj
```

With certain protcols, unless you're connecting to a public-access repository (i.e., one where that username doesn't require a password), you'll need to supply a password or *log in* first. Logging in verifies your password with the repository and stores it in a file. It's done with the **login** command, which will prompt you interactively for the password if you didn't supply one as part of `$CVSROOT`:

```
cvs -d :pserver:bach@faun.example.org:/usr/local/cvsroot login
CVS password:
```

or

```
cvs -d :pserver:bach:p4ss30rd@faun.example.org:/usr/local/cvsroot login
```

After you enter the password, cvsnt verifies it with the server. If the verification succeeds, then that combination of username, host, repository, and password is permanently recorded, so future transactions with that repository won't require you to run **cvs login**. (If verification fails, cvsnt will exit complaining that the password was incorrect, and nothing will be recorded.)

The records are stored, by default, in the file **$HOME/.cvspass** (Unix) or the Registry (Windows). The format human-readable, and to a degree human-editable, but note that the passwords are not stored in cleartext--they are trivially encoded to protect them from "innocent" compromise (i.e., inadvertent viewing by a system administrator or other non-malicious person).

Once you have logged in, all cvsnt commands using that remote repository and username will authenticate with the stored password. So, for example

```
cvs -d :pserver:bach@faun.example.org:/usr/local/cvsroot checkout foo
```

should just work (unless the password changes on the server side, in which case you'll have to re-run **cvs login**).

Note that if the **:pserver:** were not present in the repository specification, cvsnt would assume it should use **ssh** to connect with the server instead (Section 2.9.2).

Of course, once you have a working copy checked out and are running cvsnt commands from within it, there is no longer any need to specify the repository explicitly, because cvsnt can deduce the repository from the working copy's **CVS** subdirectory.

The password for a given remote repository can be removed from the password cache by using the **cvs logout** command.

2.9.4.4 Security considerations with password authentication

With pserver and sserver, the passwords are stored on the client side in a trivial encoding of the cleartext and in the pserver case 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.

With sserver, the client/server connection is encrypted using SSL, and the risk of the password being sniffed 'on the wire' is very low.

With sspi, if cvsnt login is used to gain access to a remote server, the passwords are stored on the client side in the same manner as pserver. However the passwords are never transmitted insecurely over the internet.

With pserver and sserver, the separate cvsnt password file (Section 2.9.4.1) allows people to use a different password for repository access than for login access. With other protocols the system passwords are used and the password field in the passwd file is ignored.

Once a user has non-read-only 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 cvsnt to prevent that, but no one has done so as of this writing.

Note that because the **$CVSROOT/CVSROOT** directory contains **passwd** and other files which are used to check security, you must control the permissions on this directory as tightly as the permissions on **/etc**. The same applies to the **$CVSROOT** directory itself and any directory above it in the tree. Anyone who has write access to such a directory will have the ability to become any user on the system. Note that these permissions are typically tighter than you would use if you are not using pserver.

In summary, with a password server anyone who gets the password gets repository access (which may imply some measure of general system access as well).

With pserver, the password is available to anyone who can sniff network packets or read a protected (i.e., user read-only) file. Other protocols do not have this problem.

2.9.5 Direct connection with GSSAPI

GSSAPI is a generic interface to network security systems such as Kerberos 5. If you have a working GSSAPI library, you can have cvsnt connect via a direct tcp connection, authenticating with GSSAPI.

To do this, cvsnt needs to be compiled with GSSAPI support; when configuring cvsnt it tries to detect whether GSSAPI libraries using kerberos version 5 are present. You can also use the **-with-gssapi** flag to configure.

The connection is authenticated using GSSAPI, but the message stream is *not* authenticated by default. You must use the **-a** global option to request stream authentication.

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

GSSAPI connections are handled on the server side by the same server which handles the password authentication server; see Section 2.9.4.1. If you are using a GSSAPI mechanism such as Kerberos which provides for strong authentication, you will probably want to disable the ability to authenticate via cleartext passwords. To do so, create an empty **CVSROOT/passwd** password file, and set **SystemAuth=no** in the config file (Section B.25).

The GSSAPI server uses a principal name of cvs/`hostname`, where `hostname` is the canonical name of the server host. You will have to set this up as required by your GSSAPI mechanism.

To connect using GSSAPI, use **:gserver:**. For example,

```
cvs -d :gserver:faun.example.org:/usr/local/cvsroot checkout foo
```

2.9.6 Connecting with fork

This access method allows you to connect to a repository on your local disk via the remote protocol. In other words it does pretty much the same thing as **:local:**, but various quirks, bugs and the like are those of the remote cvsnt rather than the local cvsnt.

For day-to-day operations you might prefer either **:local:** or **:fork:**, depending on your preferences. Of course **:fork:** comes in particularly handy in testing or debugging **cvsnt** and the remote protocol. Specifically, we avoid all of the network-related setup/configuration, timeouts, and authentication inherent in the other remote access methods but still create a connection which uses the remote protocol.

To connect using the **fork** method, use **:fork:** and the pathname to your local repository. For example:

```
cvs -d :fork:/usr/local/cvsroot checkout foo
```

2.9.7 Using repository aliases

Repository aliases hide the real paths to the repositories on the server behind virtual names. The server information is hidden to clients which increases security and means the cvs root strings are independent of the server architecture.

Aliases are normally specified in the **/etc/cvsnt/PServer** file on Unix, or in the CVSNT Server Control Panel on Windows. Especially on Windows is is recommended that aliases are used to avoid exposing Windows drive letters (C:, D:) to the clients.

Chapter 3

Security

A remote cvsnt *repository* can be set up to have its own security system outside of the standard security provided by the system. See also information about the chacl and chown commands, and the CVSROOT/admin file.

3.1 How to set up security

First setup the server normally. Changing the base path as described in Section 2.9.7 can be very convenient. The command should run as the user that owns the repository (not root). Use the RunAsUser setting for this.

On Unix systems setting a the Chroot variable is recommended also.

To lock down the access to the repository by default set the AclMode setting in the CVSROOT/config to 'normal'. This will stop anyone accessing the any file unless they are specifically granted access by an access control entry

On a secure system it is recommended that pserver is not used, as it sends its passwords in a trivially decryptable form. On Windows systems use encrypted SSPI, and on Unix ssh is recommended.

3.2 How to add and delete users

The cvs passwd command can be used to add or delete new users. Only an administrator can do this.

Note that deleting a user does not remove them from any user permissions.

3.3 Setting permissions for files and directories

CVSNT has its own access control mechanism that is aware of branches and other CVSNT features. There are currently 5 access that can be set, and 3 ways of matching the access entry.

The access permissions are as follows:

read User is able to read the file, or for a directory access files within that directory

write User is able to commit a new revision to the file or directory

create User is able to add new files to the directory.

tag User is able to tag the file or files within the directory.

control User is able to modify the access controls for the file or directory. This right is granted automatically to the file owner and to repository administrators.

Each access entry has 3 attributes which define which situations it applies to.

Username (-u) Defines that this access entry applies to a single user or group. Where this is specified it is the most significant attribute.

Branch (-r) Defines that this access entry applies to a single branch or tag. Where this is specified it is the second most significant attribute.

Merge (-j) Defines that this access entry applies when a merge is attempted from the specified branch.

There are also 3 optional attributes that may be specified for each access entry.

Message (-m) Define a custom message displayed to the user when an action fails due to this entry.

Priority (-p) Normally CVSNT prioritises access entries using a 'best fit' match, with ambguities solved as described above. In exceptional cases it may be necessary to override this behaviour. Specifying a priority over 100 is guaranteed to be higher than the caculated priorities, and will ensure that this ACL entry overrides all others.

Inheritance (-n) Normally directory access control entries automatically inherit, which means setting an access control entry on the root of a module affects all directories below it, unless overriden by an entry further down the tree. This option supresses that behaviour.

Access permissions are modified using the **cvs chacl** command. For example:

```
cvs chacl -a read,write,create -u theuser dir1 dir2 dir3
```

Those permissions will be inherited by a sub-diretory, so to stop commits on a sub-directory you would need to explicitly deny it. For example:

```
cvs chacl -a nowrite,nocreate -u theuser dir1/dir4
```

will grant the user named **theuser** read and write access to the three specified directories.

To view the current permissions the cvsnt lsacl command can be used. It will show the owner and all the users that have permissions in the given directories.

If the user name is not specified, those permissions will be given to all users of the directory, if not overriden by other entries. This is an easy way to give everyone read access to a directory, for instance.

For a user to have access to a directory, they must have at least read access to all the directories above it. If a user has a 'no access' ACL on a parent directory they cannot be granted access to directories below it.

The owner or a directory can be reassigned using the cvsnt chown command.

See also Section A.9, Section A.25 and Section A.11.

3.4 Groups of users can be assigned permissions

Sometimes administrators find it easier to maintain permissions on groups of users instead of on individual users. That way, if a group of people have access to a directory, the group can be assigned rights to the directories and the administrator only needs to modify the members of the group to maintain the permissions.

If **SystemAuth** is enabled CVSNT will automatically add all the system groups for the user to the list of available groups. If you don't require other groups then editing the **group** file is unnecessary.

The **group** file in the **CVSROOT** directory holds a list of groups. The file has two fields seperated by a colon, the first is the group name, the second is a list of group members, separated by white space, such as:

```
group1: user1 user2 user3
group2: me you dognamedblue
group3: peter paul mary
```

To set up groups, edit the **group** file in the CVSROOT directory in the repository and set up the permissions for the groups.

Repository administrators are automatically made a member of the group 'admin'. Don't list this group in the group file.

3.5 Running CVSNT as a nonprivileged user

Under the traditional CVS execution model, the server runs as the user checking in the file. For some security requirements this is inadequate, so CVSNT also provides a RunAsUser parameter (in the **/etc/cvsnt/PServer** or the in the registry under Windows). If this is set, the server always runs as the specified user, who should be a nonprivileged user who has read/write access only to the repository files. See also Section 3.6.

3.6 Running within a chroot jail

On operating systems that support this operation, cvsnt provides the Chroot parameter (in the **/etc/cvsnt/PServer** file). After CVSNT has loaded it will perform the chroot just prior to dropping privileges and before any filesystem operations.

The chroot jail must contain a /tmp directory for use by the server but does not need any binary or library directories. In the minimal (most secure) configuration it is impossible to run scripts of any kind. Adding binaries/libraries to allow script execution should be done with care. Never add setuid binaries to a chroot jail as it may allow an attacker an avenue to break out of it.

3.7 Setting and changing passwords

Users can use the cvs passwd command with no parameters to modify their passwords. The administrator can specify a user on the command line to change their password.

3.8 Repository administrators

If SystemAuth = Yes the user is considered to be an administrator of they are listed in the CVSROOT/admin file or if they are in the 'Administrators' group (Windows) or 'cvsadmin' group (Unix).

If SystemAuth = No only the CVSROOT/admin file is checked.

The CVSROOT/admin file contains a list of usernames who are designated repository administrators, one per line. This file should *not* be put under cvsnt control, as that would be a security risk.

Repository administrators are automatically made members of the group 'admin'.

3.9 Read-only repository access

It is possible to grant read-only repository access to people using the password-authenticated server (Section 2.9.4). (The other access methods do not have explicit support for read-only users because those methods all assume login access to the repository machine anyway, and therefore the user can do whatever local file permissions allow her to do.)

A user who has read-only access can do only those cvsnt operations which do not modify the repository, except for certain "administrative" files (such as lock files and the history file). It may be desirable to use this feature in conjunction with user-aliasing (Section 2.9.4.1).

Unlike with previous versions of cvsnt, read-only users should be able merely to read the repository, and not to execute programs on the server or otherwise gain unexpected levels of access. Or to be more accurate, the *known* holes have been plugged. Because this feature is new and has not received a comprehensive security audit, you should use whatever level of caution seems warranted given your attitude concerning security.

There are two ways to specify read-only access for a user: by inclusion, and by exclusion.

"Inclusion" means listing that user specifically in the **$CVSROOT/CVSROOT/readers** file, which is simply a newline-separated list of users. Here is a sample **readers** file:

```
melissa
splotnik
jrandom
```

(Don't forget the newline after the last user.)

"Exclusion" means explicitly listing everyone who has *write* access--if the file

```
$CVSROOT/CVSROOT/writers
```

exists, then only those users listed in it have write access, and everyone else has read-only access (of course, even the read-only users still need to be listed in the cvsnt **passwd** file). The **writers** file has the same format as the **readers** file.

Note: if your cvsnt **passwd** file maps cvs users onto system users (Section 2.9.4.1), make sure you deny or grant read-only access using the *cvsnt* usernames, not the system usernames. That is, the **readers** and **writers** files contain cvs usernames, which may or may not be the same as system usernames.

Here is a complete description of the server's behavior in deciding whether to grant read-only or read-write access:

If **readers** exists, and this user is listed in it, then she gets read-only access. Or if **writers** exists, and this user is NOT listed in it, then she also gets read-only access (this is true even if **readers** exists but she is not listed there). Otherwise, she gets full read-write access.

Of course there is a conflict if the user is listed in both files. This is resolved in the more conservative way, it being better to protect the repository too much than too little: such a user gets read-only access.

3.10 Temporary directories for the server

While running, the cvsnt server creates temporary directories. They are named

```
cvs-servpid
```

where `pid` is the process identification number of the server. They are located in the directory specified by the **TMPDIR** environment variable (Appendix C), the **-T** global option (Section A.4), or failing that **/tmp**.

In most cases the server will remove the temporary directory when it is done, whether it finishes normally or abnormally. However, there are a few cases in which the server does not or cannot remove the temporary directory, for example:

- If the server aborts due to an internal server error, it may preserve the directory to aid in debugging
- If the server is killed in a way that it has no way of cleaning up (most notably, **kill -KILL** on unix).
- If the system shuts down without an orderly shutdown, which tells the server to clean up.

In cases such as this, you will need to manually remove the **cvs-servpid** directories. As long as there is no server running with process identification number `pid`, it is safe to do so.

3.11 The CVSNT lockserver

In all recent versions of CVSNT the lockserver is the primary means of handling file locking. There should normally only be one lockserver, which may be shared by multiple repositories. Once running it should require little or no maintenence.

The lockserver provides file-level locking for the server, which allows much greater concurrency than previous versions of CVS. It also provides checkout atomicity which ensures that you always get a coherent view of the repository. The previous method of locking using directory locks on the filesystem is now depreciated and should not be used as it does not have these advantages.

Setting up the lockserver under Windows is handled by the setup program and happens automatically. Under Unix you need to arrange to run the **cvslockd** on startup - this varies between versions.

Chapter 4

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 cvsnt does have some quirks particularly in the area of renaming directories. Section 8.4.

What to do next depends on the situation at hand.

4.1 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.

4.1.1 Creating a directory tree from a number of files

When you begin using cvsnt, you will probably already have several projects that can be put under cvsnt 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 cvsnt 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, cvsnt 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 cvsnt requires them they must be present. Chapter 14, for more information about them.

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

```
$ cd ..
$ cvs checkout yoyodyne/rdir       # Explanation below
$ diff -r wdir yoyodyne/rdir
$ rm -r wdir
```

Erasing the original sources is a good idea, to make sure that you do not accidentally edit them in `wdir`, bypassing cvsnt. 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 cvsnt sets on the directories inside **$CVSROOT** are reasonable, and that they belong to the proper groups. Section 2.2.2.

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. Section B.3.

4.1.2 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 cvsnt, 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 cvsnt if they do not already exist. Then copy the files into the appropriate directories in the cvsnt repository (the name in the repository must be the name of the source file with **,v** added; the files go directly in the appropriate 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 cvsnt repository directly, rather than using cvsnt commands. Then you are ready to check out a new working directory.

The rcs file should not be locked when you move it into cvsnt; if it is, cvsnt 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.

Failing that, probably your best bet is to write a script that will check out the files one revision at a time using the command line interface to the other system, and then check the revisions into cvsnt. The **sccs2rcs** script mentioned below may be a useful example to follow.

From SCCS There is a script in the **contrib** directory of the cvsnt 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).

From PVCS There is a script in the **contrib** directory of the cvsnt source distribution called **pvcs_to_rcs** which converts pvcs archives to rcs files. You must run it on a machine which has both pvcs and rcs installed, and like everything else in contrib it is unsupported (your mileage may vary). See the comments in the script for details.

4.1.3 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 cvsnt sets on the directories inside **$CVSROOT** are reasonable.

4.2 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. Section 2.4, for an introduction. Section B.1, 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
```

Chapter 5

Revisions

For many uses of cvsnt, one doesn't need to worry too much about revision numbers; cvsnt assigns numbers such as **1.1**, **1.2**, and so on, and that is all one needs to know. However, some people prefer to have more knowledge and control concerning how cvsnt assigns revision numbers.

If one wants to keep track of a set of revisions involving more than one file, such as which revisions went into a particular release, one uses a *tag*, which is a symbolic revision which can be assigned to a numeric revision in each file.

5.1 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 !
+-----+    +-----+    +-----+    +-----+    +-----+
```

It is also possible to end up with numbers containing more than one period, for example **1.3.2.2**. Such revisions represent revisions on branches (Chapter 6); such revision numbers are explained in detail in Section 6.4.

5.2 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.

5.3 Assigning revisions

By default, cvsnt will assign numeric revisions by leaving the first number the same and incrementing the second number. For example, **1.1**, **1.2**, **1.3**, etc.

When adding a new file, the second number will always be one and the first number will equal the highest first number of any file in that directory. For example, the current directory contains files whose highest numbered revisions are **1.7**, **3.1**, and **4.12**, then an added file will be given the numeric revision **4.1**.

There is no reason to care about the revision numbers--it is easier to treat them as internal numbers that cvsnt maintains, and tags provide a better way to distinguish between things like release 1 versus release 2 of your product (Section 5.4).

5.4 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 cvsnt 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 must start with an uppercase or lowercase letter and can contain uppercase and lowercase letters, digits, **-**, and _. The two tag names **BASE** and **HEAD** are reserved for use by cvsnt. It is expected that future names which are special to cvsnt will be specially named, for example by starting with **.**, rather than being named analogously to **BASE** and **HEAD**, to avoid conflicts with actual tag names.

You'll want to choose some convention for naming tags, based on information such as the name of the program and the version number of the release. For example, one might take the name of the program, immediately followed by the version number with **.** changed to **-**, so that cvsnt 1.9 would be tagged with the name **cvs1-9**. If you choose a consistent convention, then you won't constantly be guessing whether a tag is **cvs-1-9** or **cvs1_9** or what. You might even want to consider enforcing your convention in the taginfo file (Section 9.3).

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

```
$ cvs tag rel-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     /u/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         (none)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        rel-0-4                     (revision: 1.4)
```

For a complete summary of the syntax of **cvs tag**, including the various options, see Section A.38.

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

(When you give cvsnt 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. Chapter 7.)

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 rel-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. Section A.10.1. When specifying **-r** to any of these commands, you will need beware of sticky tags; see Section 5.11.

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
```

5.5 Specifying what to tag from the working directory

The example in the previous section demonstrates one of the most common ways to choose which revisions to tag. Namely, running the **cvs tag** command without arguments causes cvsnt to select the revisions which are checked out in the current working directory. For example, if the copy of **backend.c** in working directory was checked out from revision 1.4, then cvsnt will tag revision 1.4. Note that the tag is applied immediately to revision 1.4 in the repository; tagging is not like modifying a file, or other operations in which one first modifies the working directory and then runs **cvs commit** to transfer that modification to the repository.

One potentially surprising aspect of the fact that **cvs tag** operates on the repository is that you are tagging the checked-in revisions, which may differ from locally modified files in your working directory. If you want to avoid doing this by mistake, specify the **-c** option to **cvs tag**. If there are any locally modified files, cvsnt will abort with an error before it tags any files:

```
$ cvs tag -c rel-0-4
cvs tag: backend.c is locally modified
cvs [tag aborted]: correct the above errors first!
```

5.6 Specifying what to tag by date or revision

The **cvs rtag** command tags the repository as of a certain date or time (or can be used to tag the latest revision). **rtag** works directly on the repository contents (it requires no prior checkout and does not look for a working directory).

The following options specify which date or revision to tag. See Section A.5, 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).

-r `tag` Only tag those files that contain existing tag `tag`.

The **cvs tag** command also allows one to specify files by revision or date, using the same **-r**, **-D**, and **-f** options. However, this feature is probably not what you want. The reason is that **cvs tag** chooses which files to tag based on the files that exist in the working directory, rather than the files which existed as of the given tag/date. Therefore, you are generally better off using **cvs rtag**. The exceptions might be cases like:

```
cvs tag -r 1.4 backend.c
```

5.7 Deleting, moving, and renaming tags

Normally one does not modify tags. They exist in order to record the history of the repository and so deleting them or changing their meaning would, generally, not be what you want.

However, there might be cases in which one uses a tag temporarily or accidentally puts one in the wrong place. Therefore, one might delete, move, or rename a tag. Warning: the commands in this section are dangerous; they permanently discard historical information and it can difficult or impossible to recover from errors. If you are a cvsnt administrator, you may consider restricting these commands with taginfo (Section 9.3).

To delete a tag, specify the **-d** option to either **cvs tag** or **cvs rtag**. For example:

```
cvs rtag -d rel-0-4 tc
```

deletes the tag **rel-0-4** from the module **tc**.

When we say *move* a tag, we mean to make the same name point to different revisions. For example, the **stable** tag may currently point to revision 1.4 of **backend.c** and perhaps we want to make it point to revision 1.6. To move a tag, specify the **-F** option to either **cvs tag** or **cvs rtag**. For example, the task just mentioned might be accomplished as:

```
cvs tag -r 1.6 -F stable backend.c
```

By default CVS doesn't allow moving and deleting branch tags, as this should not be done without understanding the issues that this raises. To override this, specify the **-B** option on the command line.

When we say *rename* a tag, we mean to make a different name point to the same revisions as the old tag. For example, one may have misspelled the tag name and want to correct it (hopefully before others are relying on the old spelling). To rename a tag, first create a new tag using the **-r** option to **cvs rtag**, and then delete the old name. This leaves the new tag on exactly the same files as the old tag. For example:

```
cvs rtag -r old-name-0-4 rel-0-4 tc
cvs rtag -d old-name-0-4 tc
```

5.8 Tagging and adding and removing files

The subject of exactly how tagging interacts with adding and removing files is somewhat obscure; for the most part cvsnt will keep track of whether files exist or not without too much fussing. By default, tags are applied to only files which have a revision corresponding to what is being tagged. Files which did not exist yet, or which were already removed, simply omit the tag, and cvsnt knows to treat the absence of a tag as meaning that the file didn't exist as of that tag.

However, this can lose a small amount of information. For example, suppose a file was added and then removed. Then, if the tag is missing for that file, there is no way to know whether the tag refers to the time before the file was added, or the time after it was removed. If you specify the **-r** option to **cvs rtag**, then cvsnt tags the files which have been removed, and thereby avoids this problem. For example, one might specify **-r HEAD** to tag the head.

5.9 Alias tags

Normally setting a tag equal to a branch with the **-r** causes the tag to be set to the revison at the head of the branch at that point. The **-A** option to tag and rtag changes this behaviour so that the new tag becomes an alias name for the existing branch. This allows you to switch active branches without having to change the clients.

5.10 Commit identifiers

A special 'meta-tag' is applied to each committed change to the repository, which uniquely identifies that commit. This is randomly generated string, and has no significance except that it is unique. It is used to group files from a single commit into a changeset.

To operate on a single the command the @ and @< prefixes define the committed revision, and the revision before the commit respectively.

For example:

```
$ cvs diff -r "@<91c41475ddc4ad2" -r @91c41475ddc4ad2 foo.c
Index: a.txt
===================================================================
RCS file: d:/repo/test/foo.c,v
retrieving revision 1.2
retrieving revision 1.3
diff -r1.2 -r1.3
...
```

Note that many shells treat the **<** symbol as special, so that part of the command will need to be quoted.

5.11 Sticky tags

Sometimes a working copy's revision has extra data associated with it, for example it might be on a branch (Chapter 6), or restricted to versions prior to a certain date by **checkout -D** or **update -D**. Because this data persists - that is, it applies to subsequent commands in the working copy - we refer to it as *sticky*.

Most of the time, stickiness is an obscure aspect of cvsnt that you don't need to think about. However, even if you don't want to use the feature, you may need to know *something* about sticky tags (for example, how to avoid them!).

You can use the **status** command to see if any sticky tags or dates are set:

```
$ cvs status 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 /u/cvsroot/yoyodyne/tc/driver.c,v
    Sticky Tag:         rel-1-0-patches (branch: 1.7.2)
    Sticky Date:        (none)
    Sticky Options:     (none)
```

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.

The most common use of sticky tags is to identify which branch one is working on, as described in Section 6.3. However, non-branch sticky tags have uses as well. 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** commands 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.

People often want to retrieve an old version of a file without setting a sticky tag. This can be done with the **-p** option to **checkout** or **update**, which sends the contents of the file to standard output. For example:

```
$ cvs update -p -r 1.1 file1 >file1
===================================================================
Checking out file1
rcs:  /tmp/cvs-sanity/cvsroot/first-dir/Attic/file1,v
VERS: 1.1
***************
$
```

However, this isn't the easiest way, if you are asking how to undo a previous checkin (in this example, put **file1** back to the way it was as of revision 1.1). In that case you are better off using the **-j** option to **update**; for further discussion see Section 6.8.

Chapter 6

Branching and merging

cvsnt allows you to isolate changes onto a separate line of development, known as a *branch*. When you change files on a branch, those changes do not appear on the main trunk or other branches.

Later you can move changes from one branch to another branch (or the main trunk) by *merging*. Merging involves first running **cvs update -j**, to merge the changes into the working directory. You can then commit that revision, and thus effectively copy the changes onto another branch.

6.1 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 (Section 5.4) 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 elect to either incorporate them on the main trunk, or leave them on the branch.

6.2 Creating a branch

You can create a branch with **tag -b**; for example, assuming you're in a working copy:

```
$ cvs tag -b rel-1-0-patches
```

This splits off a branch based on the current revisions in the working copy, assigning that branch the name **rel-1-0-patches**.

It is important to understand that branches get created in the repository, not in the working copy. Creating a branch based on current revisions, as the above example does, will *not* automatically switch the working copy to be on the new branch. For information on how to do that, see Section 6.3.

You can also create a branch without reference to any working copy, by using **rtag**:

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

-r rel-1-0 says that this branch should be rooted at the revision that corresponds to the tag **rel-1-0**. It need not be the most recent revision - it's often useful to split a branch off an old revision (for example, when fixing a bug in a past release otherwise known to be stable).

As with **tag**, the **-b** flag tells **rtag** to create a branch (rather than just a symbolic revision name). Note that the numeric revision number that matches **rel-1-0** will probably be different from file to file.

So, the full effect of the command is to create a new branch - named **rel-1-0-patches** - in module **tc**, rooted in the revision tree at the point tagged by **rel-1-0**.

6.3 Accessing branches

You can retrieve a branch in one of two ways: by checking it out fresh from the repository, or by switching an existing working copy over to the branch.

To check out a branch from the repository, invoke **checkout** with the **-r** flag, followed by the tag name of the branch (Section 6.2):

```
$ cvs checkout -r rel-1-0-patches tc
```

Or, if you already have a working copy, you can switch it to a given branch with **update -r**:

```
$ cvs update -r rel-1-0-patches tc
```

or equivalently:

```
$ cd tc
$ cvs update -r rel-1-0-patches
```

It does not matter if the working copy was originally on the main trunk or on some other branch - the above command will switch it to the named branch. And similarly to a regular **update** command, **update -r** merges any changes you have made, notifying you of conflicts where they occur.

Once you have a working copy tied to a particular branch, it remains there until you tell it otherwise. This means that changes checked in from the working copy will add new revisions on that branch, while leaving the main trunk and other branches unaffected.

To find out what branch a working copy is on, you can use the **status** command. In its output, look for the field named **Sticky tag** (Section 5.11) - that's cvsnt's way of telling you the branch, if any, of the current working files:

```
$ 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     /u/cvsroot/yoyodyne/tc/driver.c,v
    Sticky Tag:         rel-1-0-patches (branch: 1.7.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

    Existing Tags:
        rel-1-0-patches             (branch: 1.7.2)
        rel-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     /u/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         rel-1-0-patches (branch: 1.4.2)
    Sticky Date:        (none)
    Sticky Options:     (none)

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

Don't be confused by the fact that the branch numbers for each file are different (**1.7.2** and **1.4.2** respectively). The branch tag is the same, **rel-1-0-patches**, and the files are indeed on the same branch. The numbers simply reflect the point in each file's revision history at which the branch was made. In the above example, one can deduce that **driver.c** had been through more changes than **backend.c** before this branch was created.

See Section 6.4 for details about how branch numbers are constructed.

6.4 Branches and revisions

Ordinarily, a file's revision history is a linear series of increments (Section 5.1):

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

However, cvsnt 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 !
               / +---------+    +---------+    +---------+
              /
             /
+-----+    +-----+    +-----+    +-----+    +-----+
! 1.1 !----! 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 cvsnt 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 cvsnt (Section 6.5). The branch 1.1.1 has a special meaning. Chapter 14.

6.5 Magic branch numbers

This section describes a cvsnt feature called *magic branches*. For most purposes, you need not worry about magic branches; cvsnt handles them for you. However, they are visible to you in certain circumstances, so it may be useful to have some idea of how it works.

Externally, branch numbers consist of an odd number of dot-separated decimal integers. Section 5.1. That is not the whole truth, however. For efficiency reasons cvsnt sometimes inserts an extra 0 in the second rightmost position (1.2.4 becomes 1.2.0.4, 8.9.10.11.12 becomes 8.9.10.11.0.12 and so on).

cvsnt 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).

6.6 Merging an entire branch

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

If you wish to revert to the older CVS behaviour of merging from the point the branch forked, specify the **-b** option.

If you are updating from an Unix CVS server of older cvsnt server that doesn't support merge points, then the merge will always be done from the branch point.

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. Section 11.3.

If your source files contain keywords (Chapter 13), you might be getting more conflicts than strictly necessary. See Section 6.10, for information on how to avoid this.

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

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

It should be noted that **update -j tagname** will also work but may not produce the desired result. Section 6.9, for more.

6.7 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, cvsnt will remember that you have previously merged any only merge the new changes. You can override this by using the **cvs update -b -j** command, which will attempt to merge again the changes which you have already merged, which can have undesirable side effects.

6.8 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 undo 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. You almost always use symbolic tags rather than revision numbers when operating on multiple files.

Specifying two **-j** options can also undo file removals or additions. 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 -j 1.2 -j 1.1 file1
U file1
$ 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
$
```

6.9 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
```

After these commands are executed and a **cvs commit** is done, file **a** will be removed and file **d** added in the main branch.

Note that using a single static tag (**-j `tagname`**) rather than a dynamic tag (**-j `branchname`**) to merge changes from a branch will usually not remove files which were removed on the branch since cvsnt does not automatically add static tags to dead revisions. The exception to this rule occurs when a static tag has been attached to a dead revision manually. Use the branch tag to merge all changes from the branch or use two static tags as merge endpoints to be sure that all intended changes are propogated in the merge.

6.10 Merging and keywords

If you merge files containing keywords (Chapter 13), you will normally get numerous conflicts during the merge, because the keywords are expanded differently in the revisions which you are merging.

Chapter 7

Recursive behavior

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

```
$HOME
  |
  +--tc
  |   |
      +--CVS
      |      (internal cvsnt files)
      +--Makefile
      +--backend.c
      +--driver.c
      +--frontend.c
      +--parser.c
      +--man
      |    |
      |    +--CVS
      |    |  (internal cvsnt files)
      |    +--tc.1
      |
      +--testing
           |
           +--CVS
           |  (internal cvsnt 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** directory

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 cvsnt subcommands, not only the **update** command.

The recursive behavior of the cvsnt subcommands can be turned off with the **-l** option. Conversely, the **-R** option can be used to force recursion if **-l** is specified in **~/.cvsrc** (Section A.3).

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

Chapter 8

Adding, removing, and renaming files and directories

In the course of a project, one will often add new files. Likewise with removing or renaming, or with directories. The general concept to keep in mind in all these cases is that instead of making an irreversible change you want cvsnt to record the fact that a change has taken place, just as with modifying an existing file. The exact mechanisms to do this in cvsnt vary depending on the situation.

8.1 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. Section 1.3.1.
- Create the new file inside your working copy of the directory.
- Use **cvs add** `filename` to tell cvsnt that you want to version control the file. If the file contains binary data, specify **-kb** or **-kB** (Chapter 10).
- 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 ... 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. Section A.18.

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. Section 8.2, for an example.

The **-k** option specifies the default way that this file will be checked out; for more information see Section 13.4.

The **-m** option specifies a description for the file. This description appears in the history log (if it is enabled, Section B.21). 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**. Section A.7. 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 (Chapter 6). You can later merge the additions to another branch if you want (Section 6.9).

8.2 Removing files

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

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

- Make sure that you have not made any uncommitted modifications to the file. Section 1.3.4, 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 directory. You can for instance use **rm**.
- Use **cvs remove `filename`** to tell cvsnt 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, cvsnt 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. cvsnt will correctly create or not create the file, based on the **-r** and **-D** options specified to **checkout** or **update**.

`cvs remove` [options] files ... 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. For a full list of options, see Section A.33.

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 .
```

As a convenience you can remove the file and **cvs remove** it in one step, by specifying the **-f** option. For example, the above example could also be done like this:

```
$ cd test
$ cvs remove -f *.c
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 execute **remove** for a file, and then change your mind before you commit, you can undo the **remove** with an **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
```

If you realize your mistake after running the **commit** command you can use **add** to resurrect the file:

```
$ cvs remove foo.c
cvs remove: scheduling foo.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 .
$ cvs add foo.c
cvs add: foo.c, version 1.5, resurrected
U foo.c
cvs add: use 'cvs commit' to add this file permanently
$cvs ci -m "Oops shouldn't have deleted that..."
cvs commit: Examining .
cvs commit: Committing .
```

When you remove a file it is removed only on the branch which you are working on (Chapter 6). You can later merge the removals to another branch if you want (Section 6.9).

8.3 Removing directories

In concept removing directories is somewhat similar to removing files--you want the directory to not exist in your current working directories, but you also want to be able to retrieve old releases in which the directory existed.

The way that you remove a directory is to remove all the files in it. You don't remove the directory itself; there is no way to do that. Instead you specify the **-P** option to **cvs update** or **cvs checkout**, which will cause cvsnt to remove empty directories from working directories. (Note that **cvs export** always removes empty directories.) Probably the best way to do this is to always specify **-P**; if you want an empty directory then put a dummy file (for example **.keepme**) in it to prevent **-P** from removing it.

Note that **-P** is implied by the **-r** or **-D** options of **checkout**. This way cvsnt will be able to correctly create the directory or not depending on whether the particular version you are checking out contains any files in that directory.

8.4 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. Section 8.5.).

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

8.4.1 The Normal way to Rename

The normal way to move a file is to issue a **cvs rename** command.

```
$ cvs rename old new
$ cvs commit -m "Renamed old to new"
```

This is the simplest way to move a file. It is not error prone, and it preserves the history of what was done. CVSNT clients can retrieve the original name by checking out an older version of the repository.

This feature is only supported on CVSNT servers 2.0.55 and later.

Note that rename is still in testing at the time of writing, so if unsure use use one of the other methods below.

Note that rename information is a property of the directory, not the file. This behaviour is slightly non-obvious when you first encounter it. For a rename to be stored in the repository a **cvs commit** must be issued at the directory level, and for a rename to be picked up by other clients a **cvs update** must be issued at the directory level.

You can move a file to a different directory within a sandbox provided the destination directory is within the same server. In this case to avoid confusion it is recommended to commit both directories at the same time (by committing from a common parent directory).

8.4.2 The old way to Rename

If you are connected to a server that does not support versioned-renames, the way to move a file is to copy `old` to `new`, and then issue the normal cvsnt commands to remove `old` from the repository, and add `new` to it.

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

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 again, usually at 1.1, so if that bothers you, use the **-r rev** option to commit. For more information see Section 5.3.

8.4.3 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/dir
$ mv old,v new,v
```

Advantages:

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

Disadvantages:

- Old releases 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 cvsnt commands while you move it.

8.4.4 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/dir
$ cp old,v new,v
# Remove the old file
$ cd ~/dir
$ 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.

Advantages:

- Checking out old revisions works correctly, as long as you use **-rtag** and not **-Ddate** 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.

8.5 Moving and renaming directories

The normal way to rename or move a directory is to rename or move each file within it as described in Section 8.4.2. Then check out with the **-P** option, as described in Section 8.3.

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

1. Inform everyone who has a checked out copy of the directory that the directory will be renamed. They should commit all their changes, and remove their working copies, before you take the steps below.
2. Rename the directory inside the repository.

   ```
   $ cd $CVSROOT/parent-dir
   $ mv old-dir new-dir
   ```

3. Fix the cvsnt administrative files, if necessary (for instance if you renamed an entire module).
4. Tell everyone that they can check out again and continue working.

If someone had a working copy the cvsnt 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.

Chapter 9

History browsing

Once you have used cvsnt 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.

9.1 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 (Section A.21).

9.2 The history database

You can use the history file (Section B.21) to log various cvsnt actions. To retrieve the information from the history file, use the **cvs history** command (Section A.17).

Note: you can control what is logged to this file by using the **LogHistory** keyword in the **CVSROOT/config** file (Section B.25).

9.3 User-defined logging

You can customize cvsnt 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 (Section B.8). 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 (Section 11.6.2); this command is useful even if you are not using **cvs watch on**.

9.3.1 The taginfo file

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 (Appendix B), 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`. The standard input contains pairs of `filename revision`. A non-zero exit of the filter program will cause the tag to be aborted.

Each line in the **taginfo** file consists of a regular expression and a command-line template. Each line can have any combination of the following, in addition to those listed in the common syntax (Section B.4.1.)

%p Directory name relative to the current root.

%m Message supplied on the command line.

%s List of files being tagged

%v List of versions being tagged

%b tag type

%o tag operation

%t tag name

If no other options are specified, the default format string is **%t %o %r/%p %<{%s %v}**

Here is an example of using taginfo to log tag and rtag commands. In the taginfo file put:

```
ALL /usr/local/cvsroot/CVSROOT/loggit
```

Where **/usr/local/cvsroot/CVSROOT/loggit** contains the following script:

```
#!/bin/sh
echo "$@" >>/home/kingdon/cvsroot/CVSROOT/taglog
```

9.4 Annotate command

`cvs annotate` [**-flR**] [**-r rev**|**-D date**] files ... 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 (Section A.13).

The options to **cvs annotate** are listed in Section A.8, and can be used to select the files and revisions to annotate. The options are described in more detail in Section A.5.

Chapter 10

Handling binary files

The most common use for cvsnt is to store text files. With text files, cvsnt can merge revisions, display the differences between revisions in a human-visible fashion, and other such operations. However, if you are willing to give up a few of these abilities, cvsnt can store binary files. For example, one might store a web site in cvsnt including both text files and binary images.

10.1 The issues with binary files

While the need to manage binary files may seem obvious if the files that you customarily work with are binary, putting them into version control does present some additional issues.

One basic function of version control is to show the differences between two revisions. For example, if someone else checked in a new version of a file, you may wish to look at what they changed and determine whether their changes are good. For text files, cvsnt provides this functionality via the **cvs diff** command. For binary files, it may be possible to extract the two revisions and then compare them with a tool external to cvsnt (for example, word processing software often has such a feature). If there is no such tool, one must track changes via other mechanisms, such as urging people to write good log messages, and hoping that the changes they actually made were the changes that they intended to make.

Another ability of a version control system is the ability to merge two revisions. For cvsnt this happens in two contexts. The first is when users make changes in separate working directories (Chapter 11). The second is when one merges explicitly with the **update -j** command (Chapter 6).

In the case of text files, cvsnt can merge changes made independently, and signal a conflict if the changes conflict. With binary files, the best that cvsnt can do is present the two different copies of the file, and leave it to the user to resolve the conflict. The user may choose one copy or the other, or may run an external merge tool which knows about that particular file format, if one exists. Note that having the user merge relies primarily on the user to not accidentally omit some changes, and thus is potentially error prone.

If this process is thought to be undesirable, the best choice may be to avoid merging. To avoid the merges that result from separate working directories, see the discussion of reserved checkouts (file locking) in Chapter 11. To avoid the merges resulting from branches, restrict use of branches.

10.2 How to store binary files

There are two issues with using cvsnt to store binary files. The first is that cvsnt by default converts 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).

The second is that a binary file might happen to contain data which looks like a keyword (Chapter 13), so keyword expansion must be turned off.

The third is that storing differences (deltas) between binary files can be very inefficient.

The **-kb** option available with some cvsnt commands insures that neither line ending conversion nor keyword expansion will be done.

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 update -kb kotest
# For non-unix systems:
# Copy in a good copy of the file from outside CVS
$ cvs commit -fm "make it binary" kotest
```

When you check in the file **kotest** the file is not preserved as a binary file, because you did not check it in as a binary file. The **cvs update -kb** command sets the current keyword substitution method for this file, but it does not alter the working copy of the file that you have. If you need to cope with line endings (that is, you are using cvsnt 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.

Older versions of CVS/CVSNT didn't properly version control the expansion option, and had an **admin -kb** option.

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. Section B.3. There is currently no way to have cvsnt detect whether a file is binary based on its contents. The main difficulty with designing such a feature is that it is not clear how to distinguish between binary and non-binary files, and the rules to apply would vary considerably with the operating system.

The **-kB** solves the inefficiency problem by using a special binary delta algorithm to store the files. The function is similar to **-kb** except it is more efficient, and some functions that rely on text deltas, such as **cvs annotate** do not work with it.

Normally CVS assumes that every file (whether binary or not) is a text file of some sort. This makes sense for most files that you would normally use during development, and the storage of such files is highly efficient.

However, if you are storing pure binary files (libraries, or perhaps Word documents) it is very inefficient to treat them as text files. CVSNT solves this problem with the -kB option. This tells CVSNT to switch to an alternate algorithm to store such files.

Chapter 11

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, sccs, visual studio, pvcs etc. Currently the usual way to get reserved checkouts with cvsnt is the setting the file to have the cvsnt expansion keyword **-kx** or **-kc** (Section 13.4) and using the command **cvs edit** (Section A.14.1). This is very nicely integrated into cvsnt and can work with the watch features, described below.

The default model with cvsnt 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 cvsnt commands to bring their working copy up to date with the repository revision. This process is almost automatic.

cvsnt also supports mechanisms which facilitate various kinds of communication, 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.

11.1 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 cvsnt 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.

File had conflicts on merge This is like Locally Modified, except that a previous **update** command gave a conflict. If you have not already done so, you need to resolve the conflict as described in Section 11.3.

Unknown cvsnt 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.

The options to **status** are listed in Section A.37. For information on its **Sticky tag** and **Sticky date** output, see Section 5.11. For information on its **Sticky options** output, see the **-k** option in Section A.40.1.

You can think of the **status** and **update** commands as somewhat complementary. You use **update** to bring your files up to date, and you can use **status** to give you some idea of what an **update** would do (of course, the state of the repository might change before you actually run **update**). In fact, if you want a command to display file status in a more brief format than is displayed by the **status** command, you can invoke

```
$ cvs -n -q update
```

The **-n** option means to not actually do the update, but merely to display statuses; the **-q** option avoids printing the name of each directory. For more information on the **update** command, and these options, see Section A.40.

11.2 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 directory.

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, cvsnt 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, cvsnt 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. Section A.40, for a complete description of the **update** command.

11.3 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
```

cvsnt 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, cvsnt 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. In previous versions of cvsnt, you also needed to insure that the file contains no conflict markers. Because your file may legitimately contain conflict markers (that is, occurrences of **>>>>>>>** at the start of a line that don't mark a conflict), the current version of cvsnt will print a warning and proceed to check in the file.

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

11.4 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. Section B.1. Section B.8. You can use these features of cvsnt to, for instance, instruct cvsnt to mail a message to all developers, or post a message to a local newsgroup.

11.5 Several developers simultaneously attempting to run CVS

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

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

cvsnt will try again every second, 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, and you are not running the lockserver (see Section 3.11, look in the repository directory mentioned in the message and remove files which they own whose names start with **#cvs.rfl**, **#cvs.wfl**, or **#cvs.lock**.

Note that these locks are to protect cvsnt's internal data structures and have no relationship to the word *lock* in the sense used by rcs--which refers to reserved checkouts (Chapter 11).

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

Checkouts on cvsnt are atomic which means:

```
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.
```

By default atomic checkouts occur across an entire checkout, (provided the lockserver is running), but atomic commits are only atomic at the file level. 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** will either get all of the changes, or none of them. If however, the person running **cvs commit** suffers a power failure in the middle of the commit, it is possible that only one of the files will be updated.

11.6 Mechanisms to track who is editing files

For many groups, use of cvsnt 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 cvsnt is not able to enforce this behavior.

11.6.1 Setting up cooperative edits

`cvs watch ro` [**-lR**] files ... Specify that developers should run **cvs edit** before editing `files`. cvsnt 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, cvsnt sets up cooperative edits for 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. The **-R** option can be used to force recursion if the **-l** option is set in **~/.cvsrc** (Section A.3).

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

`cvs watch rw` [**-lR**] files ... Do not create `files` read-only on checkout; thus, developers will not be reminded to use **cvs edit** and **cvs unedit**.

The `files` and options are processed as for **cvs watch ro**.

11.6.2 Telling CVS to notify you when someone modifies a file

You can tell cvsnt that you want to receive notifications about various actions taken on a file. You can do this without using **cvs watch ro** for the file, however generally you will want to use **cvs watch ro**, so that developers are reminded to use the **cvs edit** command.

`cvs watch add` [**-a** action] [**-lR**] files ... 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 cvsnt 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 options are processed as for the **cvs watch** commands.

`cvs watch remove` [**-a** action] [**-lR**] files ... 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, cvsnt calls the **notify** administrative file. Edit **notify** as one edits the other administrative files (Section 2.4). This file follows the usual conventions for administrative files (Section B.4.1), where each line is a regular expression followed by a command to execute. The command should contain a single occurrence 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. A common thing to put in the **notify** file is the single line:

```
ALL mail %s -s "CVS notification for bug %b"
```

This causes users to be notified by electronic mail (assuming the command 'mail' exists).

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, cvsnt 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**, cvsnt will pass the `value` (normally an email address on some other machine).

CVSNT does not notify you for your own changes. Currently this check is done based on whether the user name of the person taking the action which triggers notification matches the user name of the person getting notification. In fact, in general, the watches features only track one edit by each user. It probably would be more useful if watches tracked each working directory separately, so this behavior might be worth changing.

You can also pass information about the notification to the script, using the following substitution variables:

%s user being notified, or email (see above).

%b bug being edited.

%m reason for edit as supplied by user.

%d date and time of edit.

%u username of user performing the unedit (may not be the same as %s).

%t tag or branch being edited.

11.6.3 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 cvsnt uses that term for obtaining a copy of the sources (Section 1.3.1), an operation which those systems call a *get* or a *fetch*.

`cvs edit` [options] files ... Prepare to edit the working files `files`. cvsnt 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`; cvsnt 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 options are processed as for the **cvs watch** commands.

If the **-c** option is given, then the file is checked for existing editors before the edit can proceed. In this way a reasonable facsimile of 'reserved edits' can be achieved (note however this is a suboptimal way to use cvsnt).

If the **-x** is given, then the edit is marked as exclusive on the server, and other users will be prevented from creating further edits. This option will also prevent commits to the file by their users. (Implies -c).

The **-z** option stores the edited base revision in a compressed form. This is useful if you are using an IDE which tends to pick up the base revisions while searching files. Its also saves quite a bit of disk space on large edits.

Normally edits apply only to the branch that you are currently using. The **-w** causes the edit to apply to all branches within the file (which is the behaviour for pre-2.0.55 CVSNT).

The **-b** marks this edit as belonging to a specific bug. This information is stored and also sent to the **CVSROOT/notify** script.

The **-m** sets a reason message for this edit. This can be processed by the **CVSROOT/notify** script.

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` [-lR] [-r] [-u user] [-w] [-b bug] [-m message] [files...]

Abandon work on the working files `files`, and revert them to the repository versions on which they are based. CVSNT by default also makes the files read only. Use the -w option to override this.. CVSNT then notifies users who have requested **unedit** notification for any of `files`.

The `files` and options 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.

The reason message passed to this function is passed directly to the **CVSROOT/Notify** script. If the **-b** option is used then only those files edited under a specific bug are unedited.

Repository administrators can use the 'unedit others' option **-u**. Only use this as a last resort as it only does the server side of the unedit.

11.6.4 Information about who is watching and editing

`cvs watchers` [**-lR**] files ... 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 options are processed as for the **cvs watch** commands.

`cvs editors` [**-lR**] [**-a**] files ... 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 options are processed as for the **cvs watch** commands.

11.6.5 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 cvsnt 1.6 or earlier with the repository, you get an error message such as the following (all on one line):

```
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 cvsnt 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 cvsnt 1.6 can cope with.

11.7 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 a brief description of some of the issues. There are many ways to organize a team of developers. cvsnt 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 11.6 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.

Chapter 12

Revision management

If you have read this far, you probably have a pretty good grasp on what cvsnt 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 cvsnt 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.

12.1 When to commit?

Your group should decide which policy to use regarding commits. Several policies are possible, and as your experience with cvsnt 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 (Section B.6), 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.

Chapter 13

Keyword substitution

As long as you edit source files inside a working directory 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.

cvsnt can use 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.

13.1 Keyword List

This is a list of the keywords. This list may be altered or augmented by the use of the keywords control file (seeSection B.17).

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

$Branch$ The name of the branch that the revision is a member of.

$CommitId$ The Commit (or Session) identifier of the commit that generated this 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 cvsnt.

$RCSHeader$ A standard header containing the relative 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 cvsnt.

Id Same as **$Header$**, except that the rcs filename is without a path.

$Name$ Tag name used to check out this file. The keyword is expanded only if one checks out with an explicit tag name. For example, when running the command **cvs co -r first**, the keyword expands to **Name: first**.

$Locker:$ The login name of the user who locked the revision (empty if not locked, which is the normal case unless **cvs admin -l** is in use). This keyword has little meaning under cvsnt.

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 the same string which precedes the **$Log** keyword. For example, if the file contains

```
/* Here is what people have been up to:
 *
 * $Log: foo.txt,v $
 * Revision 1.1  1997/01/03 14:23:51  joe
 * Add the superfrobnicate option
 *
 */
```

then additional lines which are added when expanding the **$Log** keyword will be preceded by * . Unlike previous versions of cvsnt and rcs, the *comment leader* from the rcs file is not used. The **$Log** keyword is useful for accumulating a complete change log in a source file, but for several reasons it can be problematic. Section 13.5.

$rcsfile$ 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.7.1.

13.2 Using keywords

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

It is common to embed the **Id** string in the source files so that it gets passed through to generated files. For example, if you are managing computer program source code, you might include a variable which is initialized to contain that string. Or some C compilers may provide a **#pragma ident** directive. Or a document management system might provide a way to pass a string through to generated files.

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:
     $Id$
$ gcc samp.c
$ ident a.out
a.out:
     $Id: cvs.dbk,v 1.1.2.1 2004/04/16 14:14:42 tmh 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 keyword with the magic sccs phrase, like this:

```
static char *id="@(#) $Id$";
```

13.3 Avoiding substitution

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

There is unfortunately no way to selectively turn off keyword substitution. You can use **-ko** (Section 13.4) to turn off keyword substitution entirely.

In many cases you can avoid using 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.

13.4 Substitution modes

Keyword substitution modes are stored for each version of a file. When you commit a new revision that version will be stored exactly as it is in the sandbox, including the substitution mode.

You can override the substitution mode by using the **-k** option to **cvs add**, **-k** or **-A** options to **cvs checkout** or **cvs update**. **cvs diff** also has a **-k** option. For some examples, see Chapter 10, and Section 6.10.

The **cvs update** and **cvs checkout** commands allow you to modify existing substitution modes without overwriting the existing ones. This is done by prefixing the mode with '+' (for add) or '-' (for remove). For example to switch off keyword substitution for all files in a subtree:

```
$ cvs update -k+o
$ cvs commit -fm "Change substitution mode"
```

The modes available are defined by combining an optional encoding with a series of options.

Some combinations were not available on older CVS versions so be careful if you want to access your repository from older clients. The CVSNT server will automatically downgrade some of these options if an older client fetches a file.

Encodings:

t Treat the file as a text file. This is the default setting if no encoding is specified.

MBCS character sets that don't change the behaviour of CR/LF and NULL should also work in this mode. eg. Shift-JIS and EUC.

b Treat the file as binary. No interpretation is done of the contents and they are stored verbatim. Be default no keyword expansion is done. Binary files are considered non-mergable by CVS.

B Treat the file as binary. No interpretation is done of the contents and they are stored verbatim. Be default no keyword expansion is done. Binary files are considered non-mergable by CVS. In addition, an alternate storage algorithm is used that is optimised for storage of binary files.

u Treat the file as Unicode. The file will be checked in/out in UCS-2 (or UTF-16) encoding and internally stored as UTF-8 by the server.

{...} Use an extended encoding. Any encoding supported by the client-side iconv library can be used, however beware of mismatches between clients (the Win32 version does not currently support EBCIDIC encodings for example). The following list will work on all platforms that are using Unicode-capable CVSNT:

ucs2le, utf16le Little-endian UCS-2 without BOM.

ucs2be, utf16be Big-endian UCS-2 without BOM.

ucs2le_bom, utf16le_bom Little-endian UCS-2 with BOM.

ucs2be_bom, utf16be_bom Big-endian UCS-2 with BOM.

Flags:

c Enforce cooperative edits (edit -c) for the file.

x Enforce reserved (ie: exclusive) edit (edit -x) for the file.

k Preserve the keyword string (default). When combined with the v flag this generates results using the default form, e.g. **$Revision$** for the **Revision** keyword. On its own it produces output with no keywords expanded.

v Generate keyword values for keyword strings. Normally paired with the k option. If it is used on its own the effect is to strip keywords from the output - for example, for the **Revision** keyword, generate the string **5.7** instead of **$Revision: 1.1.2.1$**. 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.

l Insert the lockers name if the given revision is currently locked. The locker's name is only relevant if **cvs admin -l** is in use.

s File is never considered modified on the client. A normal commit will never commit this file, unless -f is used to force it. Use with care, and for files that change infrequently, since local changes will be lost on update.

1 Unversioned. Only ever keep a single revision in the repository. History is lost, and the file is merely kept as the latest copy. Only one branch (HEAD) ever exists and an update of any revision will return the single revision.

L When checking out, always create a file with Unix line endings (LF).

D Whcn checking out, always create a file with Dos/Windows line endings (CR/LF).

M When checking out, always create a file with Mac line endings (CR)

o 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 **$Revision: 1.1.2.1;$** instead of **$Revision$** if that is how the string appeared when the file was checked in.

z Compress the files and deltas when they are stored. This sacrifices speed for disk space - only use if disk space is at a premium.

13.5 Log

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 cvsnt 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.

Chapter 14

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. cvsnt can help you with this task.

In the terminology used in cvsnt, 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*. cvsnt 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. When you import a new file, 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.

14.1 Importing 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 14.6.). The *release tags* are symbolic names for a particular release, such as **FSF_0_04**.

Suppose you have the sources to a program called **wdiff** in a directory **wdiff-0.04**, 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:

```
$ 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**.

14.2 Updating 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 (Section 11.3). 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.

14.3 Reverting to the latest vendor release

You can also revert local changes completely and return to the latest vendor release by changing the `head' revision back to the vendor branch on all files. For example, if you have a checked-out copy of the sources in **~/work.d/wdiff**, and you want to revert to the vendor's version for all the files in that directory, you would type:

```
$ cd ~/work.d/wdiff
$ cvs admin -bWDIFF .
```

You must specify the **-bWDIFF** without any space after the **-b**. Section A.7.1.

14.4 How to handle binary files with cvs import

Use the **-k** wrapper option to tell import which files are binary. Section B.3.

14.5 How to handle keyword substitution with cvs import

The sources which you are importing may contain keywords (Chapter 13). For example, the vendor may use cvsnt or some other system which uses similar keyword expansion syntax. If you just import the files in the default fashion, then the keyword expansions supplied by the vendor will be replaced by keyword expansions supplied by your own copy of cvsnt. It may be more convenient to maintain the expansions supplied by the vendor, so that this information can supply information about the sources that you imported from the vendor.

To maintain the keyword expansions supplied by the vendor, supply the **-ko** option to **cvs import** the first time you import the file. This will turn off keyword expansion for that file entirely, so if you want to be more selective you'll have to think about what you want and use the **-k** option to **cvs update** or **cvs admin** as appropriate.

14.6 Multiple vendor branches

All the examples so far assume that there is only one vendor from which you are getting sources. In some situations you might get sources from a variety of places. For example, suppose that you are dealing with a project where many different people and teams are modifying the software. There are a variety of ways to handle this, but in some cases you have a bunch of source trees lying around and what you want to do more than anything else is just to all put them in cvsnt so that you at least have them in one place.

For handling situations in which there may be more than one vendor, you may specify the **-b** option to **cvs import**. It takes as an argument the vendor branch to import to. The default is **-b 1.1.1**.

For example, suppose that there are two teams, the red team and the blue team, that are sending you sources. You want to import the red team's efforts to branch 1.1.1 and use the vendor tag RED. You want to import the blue team's efforts to branch 1.1.3 and use the vendor tag BLUE. So the commands you might use are:

```
$ cvs import dir RED RED_1-0
$ cvs import -b 1.1.3 dir BLUE BLUE_1-5
```

Note that if your vendor tag does not match your **-b** option, cvsnt will not detect this case! For example,

```
$ cvs import -b 1.1.3 dir RED RED_1-0
```

Be careful; this kind of mismatch is sure to sow confusion or worse. I can't think of a useful purpose for the ability to specify a mismatch here, but if you discover such a use, don't. cvsnt is likely to make this an error in some future release.

Chapter 15

How your build system interacts with CVS

As mentioned in the introduction, cvsnt does not contain software for building your software from source code. This section describes how various aspects of your build system might interact with cvsnt.

One common question, especially from people who are accustomed to rcs, is how to make their build get an up to date copy of the sources. The answer to this with cvsnt is two-fold. First of all, since cvsnt itself can recurse through directories, there is no need to modify your **Makefile** (or whatever configuration file your build tool uses) to make sure each file is up to date. Instead, just use two commands, first **cvs -q update** and then **make** or whatever the command is to invoke your build tool. Secondly, you do not necessarily *want* to get a copy of a change someone else made until you have finished your own work. One suggested approach is to first update your sources, then implement, build and test the change you were thinking of, and then commit your sources (updating first if necessary). By periodically (in between changes, using the approach just described) updating your entire tree, you ensure that your sources are sufficiently up to date.

One common need is to record which versions of which source files went into a particular build. This kind of functionality is sometimes called *bill of materials* or something similar. The best way to do this with cvsnt is to use the **tag** command to record which versions went into a given build (Section 5.4).

Using cvsnt in the most straightforward manner possible, each developer will have a copy of the entire source tree which is used in a particular build. If the source tree is small, or if developers are geographically dispersed, this is the preferred solution. In fact one approach for larger projects is to break a project down into smaller separately-compiled subsystems, and arrange a way of releasing them internally so that each developer need check out only those subsystems which are they are actively working on.

Another approach is to set up a structure which allows developers to have their own copies of some files, and for other files to access source files from a central location. Many people have come up with some such a system using features such as the symbolic link feature found in many operating systems, or the **VPATH** feature found in many versions of **make**. One build tool which is designed to help with this kind of thing is Odin (see **ftp://ftp.cs.colorado.edu/pub/distribs/odin**).

Chapter 16

Special Files

In normal circumstances, cvsnt works only with regular files. Every file in a project is assumed to be persistent; it must be possible to open, read and close them; and so on. cvsnt also ignores file permissions and ownerships, leaving such issues to be resolved by the developer at installation time. In other words, it is not possible to "check in" a device into a repository; if the device file cannot be opened, cvsnt will refuse to handle it. Files also lose their ownerships and permissions during repository transactions.

Appendix A

Guide to CVS commands

This appendix describes the overall structure of cvsnt commands, and describes some commands in detail

A.1 Overall structure of CVS commands

The overall format of all cvsnt commands is:

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

cvs The name of the cvsnt program.

cvs_options Some options that affect all sub-commands of cvsnt. 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 cvsnt 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 CVS's exit status

cvsnt can indicate to the calling environment whether it succeeded or failed by setting its *exit status*. The exact way of testing the exit status will vary from one operating system to another. For example in a unix shell script the **$?** variable will be 0 if the last command returned a successful exit status, or greater than 0 if the exit status indicated failure.

If cvsnt is successful, it returns a successful status; if there is an error, it prints an error message and returns a failure status. The one exception to this is the **cvs diff** command. It will return a successful status if it found no differences, or a failure status if there were differences or if there was an error. Because this behavior provides no good way to detect errors, in the future it is possible that **cvs diff** will be changed to behave like the other cvsnt commands.

A.3 Default options and the ~/.cvsrc and CVSROOT/cvsrc files

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
checkout -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 **cvsnt** to specify global options (Section A.4). For example the following line in **.cvsrc**

```
cvs -z6
```

causes cvsnt to use compression level 6.

The **CVSROOT/cvsrc** file on the server contains the default .cvsrc file that is used by all compatible clients. This is merged with the local .cvsrc file and the result behaves as normal.

The CVSROOT/cvsrc file cannot contain global options (Section A.4) as it is parsed after the server has started.

Older **cvsnt** clients and Unix cvs clients will not use the global cvsrc.

A.4 Global options

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

-allow-root=`rootdir` Specify legal cvsroot directory. Obsolete. See Section 2.9.4.1.

-a, --authenticate Authenticate all communication between the client and the server. Only has an effect on the cvsnt client. Authentication prevents certain sorts of attacks involving hijacking the active tcp connection. Enabling authentication does not enable encryption.

-b `bindir` In CVS 1.9.18 and older, this specified that rcs programs are in the `bindir` directory. Current versions of cvsnt do not run rcs programs; for compatibility this option is accepted, but it does nothing.

-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. Chapter 2.

-e editor Use `editor` to enter revision log information. Overrides the setting of the **$CVSEDITOR** and **$EDITOR** environment variables. For more information, see Section 1.3.2.

-f Do not read the **~/.cvsrc** file. This option is most often used because of the non-orthogonality of the cvsnt 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.

-F file Read the contents of `file` and append it to the supplied command line. Arguments are separated by whitespace, and follow normal quoting rules.

-H, -help 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 overall help for cvsnt, including a list of other help options.

-l Do not log the **cvs_command** in the command history (but execute it anyway). Section A.17, 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.

This option has a long history and is not guaranteed to actually leave the sandbox in the same state that it started with. It is supported only for the **checkout** command to an empty directory, which is required by certain frontends.

cvsnt has other commands which replace the functionality of this option - see **status -q** and **ls** commands.

-N Enable :local: access to a network share. Normally this is explicitly prohibited to discourage its use. It is recommended that you setup a proper server instead, as problems encountered using network shares will not normally be supported.

-o[locale] Where supported by the server (CVSNT 2.0.57+), try to convert the character set of the server to that of the client. This allows you to store exended characters such an umlauts in the repository even if your machine is set to a different codepage/language to the server.

For Win32, the codepage used is the current ANSI codepage. This may not render correctly in the OEM codepage used by the command line processor. To verify that CVSNT is doing the correct conversion look at the filename in Windows Explorer.

As of CVSNT 2.0.59 this is the default behaviour.

-O Disable client/server locale translation. If the client and server are not in the same locale then care must be taken not to use characters outside US-ASCII codepage if this option is used.

-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 read-only. Same effect as if the **$CVSREAD** environment variable is set (Appendix C). The default is to make working files writable, unless watches are on (Section 11.6).

--readonly Make all users readonly. This is used for read only mirror servers.

-s variable=value Set a user variable (Section B.24).

-t Trace program execution; display messages showing the steps of cvsnt activity. Particularly useful with **-n** to explore the potential impact of an unfamiliar command. More instances of -t increase verbosity.

-v, --version Display version and copyright information for cvsnt.

-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 Encrypt all communication between the client and the server. Only has an effect on the cvsnt client. Enabling encryption implies that message traffic is also authenticated.

-z gzip-level Set the compression level. Valid levels are 1 (high speed, low compression) to 9 (low speed, high compression), or 0 to disable compression (the default). Only has an effect on the cvsnt client.

A.5 Common command options

This section describes the **command_options** that are available across several cvsnt 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 cvsnt 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**, cvsnt 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, Section 5.11).

-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; Section A.17.1).

A wide variety of date formats are supported by cvsnt. The most standard ones are ISO8601 (from the International Standards Organization) and the Internet e-mail standard (specified in RFC822 as amended by RFC1123).

ISO8601 dates have many variants but a few examples are:

```
1972-09-24
1972-09-24 20:05
```

There are a lot more ISO8601 date formats, and cvsnt accepts many of them, but you probably don't want to hear the *whole* long story :-).

In addition to the dates allowed in Internet e-mail itself, cvsnt also allows some of the fields to be omitted. For example:

```
24 Sep 1972 20:05
24 Sep
```

The date is interpreted as being in the local timezone, unless a specific timezone is specified.

These two date formats are preferred. However, cvsnt currently accepts a wide variety of other date formats. They are intentionally not documented here in any detail, and future versions of cvsnt might not accept all of them.

One such format is **`month/day/year`**. This may confuse people who are accustomed to having the month and day in the other order; **1/4/96** is January 4, not April 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 cvsnt 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).

Note that even with **-f**, a tag that you specify must exist (that is, in some file, not necessary in every file). This is so that cvsnt will continue to give an error if you mistype a tag name.

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

Warning: The **commit** and **remove** commands also have a **-f** option, but it has a different behavior for those commands. See Section A.12.1, and Section 8.2.

-k `kflag` Alter the default processing of keywords. Chapter 13, 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, cvsnt 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**, **import** 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: **annotate**, **checkout**, **commit**, **diff**, **edit**, **editors**, **export**, **log**, **rdiff**, **remove**, **rtag**, **status**, **tag**, **unedit**, **update**, **watch**, and **watchers**.

-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 (Section B.1); 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 empty directories. See Section 8.3.

-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.

-R Process directories recursively. This is on by default.

Available with the following commands: **annotate**, **checkout**, **commit**, **diff**, **edit**, **editors**, **export**, **rdiff**, **remove**, **rtag**, **status**, **tag**, **unedit**, **update**, **watch**, and **watchers**.

-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: cvsnt remembers the tag and continues to use it on future update commands, until you specify otherwise (for more information on sticky tags/dates, Section 5.11).

The tag can be either a symbolic or numeric tag, as described in Section 5.4, or the name of a branch, as described in Chapter 6.

Specifying the **-q** global option along with the **-r** command option is often useful, to suppress the warning messages when the rcs 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 cvsnt command!

-r is available with the **checkout**, **commit**, **diff**, **history**, **export**, **rdiff**, **rtag**, 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. Available with the following commands: **import**, and **update**.

A.6 add--Add files to repository

- Requires: repository, working directory.
- Changes: repository.
- Synonym: ad,new

This adds new files to the existing working directory. Before any commands which operate on sandbox files can be used, they must be added to the list of cvs controlled files using this command.

Directories are added immediately, and exist on all branches. Ordinary files must be committed before other users are able to use them.

A.6.1 add options

-b bugid Mark the newly added file with a bug identifier.

-k kflag Override the default expansion option for the file.

-m message Use "message" for the message creation log.

-r branch Add the new file onto a different branch (default is the same branch as the directory).

A.7 admin--Administration

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

This is the cvsnt interface to assorted administrative facilities. Some of them have questionable usefulness for cvsnt but exist for historical purposes. Some of the questionable options are likely to disappear in the future. This command *does* work recursively, so extreme care should be used.

Do not use this command unless you know what you are doing. Some of the admin commands can have unexpected consequences.

On unix, if there is a group named **cvsadmin**, only members of that group can run **cvs admin**. This group should exist on the server, or any system running the non-client/server cvsnt. To disallow **cvs admin** for all users, create a group with no users in it. On Windows, server administrators are able to use the admin command.

A.7.1 admin options

-ksubst This option is provided as legacy support for older servers and has no function under CVSNT.

-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. There can be no space between **-l** and its argument.

This command is depreciated in favour of the 'edit -c' and 'edit -x' commands, which gives pseudo and genuine reserved checkouts.

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

-orange Deletes (*outdates*) the revisions given by `range`.

Note that this command can be quite dangerous unless you know *exactly* what you are doing (for example see the warnings below about how the `rev1:rev2` syntax is confusing).

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! If you delete different revisions than you planned, either due to carelessness or (heaven forbid) a cvsnt bug, there is no opportunity to correct the error before the revisions are deleted. It probably would be a good idea to experiment on a copy of the repository first.

Specify `range` in one of the following ways:

rev1::rev2 Collapse all revisions between rev1 and rev2, so that cvsnt only stores the differences associated with going from rev1 to rev2, not intermediate steps. For example, after **-o 1.3::1.5** one can retrieve revision 1.3, revision 1.5, or the differences to get from 1.3 to 1.5, but not the revision 1.4, or the differences between 1.3 and 1.4. Other examples: **-o 1.3::1.4** and **-o 1.3::1.3** have no effect, because there are no intermediate revisions to remove.

::rev Collapse revisions between the beginning of the branch containing `rev` and `rev` itself. The branchpoint and `rev` are left intact. For example, **-o ::1.3.2.6** deletes revision 1.3.2.1, revision 1.3.2.5, and everything in between, but leaves 1.3 and 1.3.2.6 intact.

rev:: Collapse revisions between `rev` and the end of the branch containing `rev`. Revision `rev` is left intact but the head revision is deleted.

rev Delete the revision `rev`. For example, **-o 1.3** is equivalent to **-o 1.2::1.4**.

rev1:rev2 Delete the revisions from `rev1` to `rev2`, inclusive, on the same branch. One will not be able to retrieve `rev1` or `rev2` or any of the revisions in between. For example, the command **cvs admin -oR_1_01:R_1_02 .** is rarely useful. It means to delete revisions up to, and including, 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! In most cases you want to specify `rev1::rev2` instead.

:rev Delete revisions from the beginning of the branch containing `rev` up to and including `rev`.

rev: Delete revisions from revision `rev`, including `rev` itself, to the end of the branch containing `rev`.

None of the revisions to be deleted may have branches or locks.

If any of the revisions to be deleted have symbolic names, and one specifies one of the **::** syntaxes, then cvsnt will give an error and not delete any revisions. If you really want to delete both the symbolic names and the revisions, first delete the symbolic names with **cvs tag -d**, then run **cvs admin -o**. If one specifies the non-**::** syntaxes, then cvsnt will delete the revisions but leave the symbolic names pointing to nonexistent revisions. This behavior is preserved for compatibility with previous versions of cvsnt, but because it isn't very useful, in the future it may change to be like the **::** case.

Due to the way cvsnt handles branches `rev` cannot be specified symbolically if it is a branch. Section 6.5, 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 (Section 6.8).

-q Run quietly; do not print diagnostics.

-t[file] Useful with cvsnt. 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 **-**. The descriptive text can be seen in the output from **cvs log** (Section A.21). There can be no space between **-t** and its argument.

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**.

-t-string Similar to **-tfile**. Write descriptive text from the `string` into the rcs file, deleting the existing text. There can be no space between **-t** and its argument.

A.8 annotate--find out who made changes to the files

- Requires: repository, working directory.
- Changes: nothing.
- Synonyms: ann

Annotate is used to discover who made changes to specific lines within files. The output to annotate gives the username, date and version number of the change.

The output to annotate is similar to checkout, for example:

```
1.54          (tmh      28-Aug-02):      char host[NI_MAXHOST];
1.54          (tmh      28-Aug-02):
1.71          (tmh      26-Mar-03):      if(!getnameinfo((struct sockaddr*)&ss,ss_len,host, ↩
    NI_MAXHOST,NULL,0,flags))
1.54          (tmh      28-Aug-02):          remote_host_name = xstrdup(host);
```

This information is usually enough to assign blame (or credit!) when tracing bugs.

A.8.1 annotate options

-l Local directory only, no recursion

-R Process directories recursively (default).

-f Use head revision if tag is not found.

-r rev Annotate files for specific revision or tag.

-D date Annotate files for specific date

A.9 chacl--Change access control lists

- Synopsis: chacl [-R] [-r branch] [-u user] [-j branch] [-n] [-p priority] [-m message] [-a [no]{read|write|create|tag|control|all|none}[,...]] [-d] [file or directory...]
- Requires: repository, working directory.
- Changes: repository.
- Synonyms: setacl, setperm

Modify the access control list for a file or directory. See also Section 3.3 for more details.

A.9.1 chacl options

-a Add access control entry - any combination of read,write,create,tag,control. Any of these may be prefixed by 'no' to deny access. Also special access all or none for setting all permissions.

-d Delete access control entry

-j branch Entry applies when merging from branch.

-m message Show customised error message when access is blocked due to this entry.

-n Stop entry from being inherited by subdirectories.

-p priority Modify the priority that this entry has. This is an advanced option - the internal prioritisation is designed to work correctly in most circumstances.

-r branch This entry applies to a single branch only.

-R Recurse into subdirectories. Note that becuase entries are by default recursive this option is not normally required.

-u user This entry applies to a single user (or group) only.

A.10 checkout--Check out sources for editing

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

Create or update a working directory containing copies of the source files specified by `modules`. You must execute **checkout** before using most of the other cvsnt commands, since most of them operate on your working directory.

The `modules` 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. Section B.1.

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 cvsnt (Section A.4) is specified, the **CVSREAD** environment variable is specified (Appendix C), or a watch is in effect for that file (Section 11.6).

Note that running **checkout** on a directory that was already built by a prior **checkout** is also permitted. This is similar to specifying the **-d** option to the **update** command in the sense that new directories that have been created in the repository will appear in your work area. However, **checkout** takes a module name whereas **update** takes a directory name. Also to use **checkout** this way it must be run from the top level directory (where you originally ran **checkout** from), so before you run **checkout** to update an existing directory, don't forget to change your directory to the top level directory.

For the output produced by the **checkout** command see Section A.40.2.

A.10.1 checkout options

These standard options are supported by **checkout** (Section A.5, 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 5.11, 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 keywords according to `kflag`. See Chapter 13. 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.37, for more information on the **status** command.

-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; Section B.1).

-P Prune empty directories. See Section 8.5.

-p Pipe files to the standard output.

-R Checkout directories recursively. This option is on by default.

-r `tag` Use revision `tag`. This option is sticky, and implies **-P**. See Section 5.11, 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 5.11, 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. In general, using this flag is equivalent to using **mkdir dir; cd dir** followed by the checkout command without the **-d** flag.

There is an important exception, however. It is very convenient when checking out a single item to have the output appear in a directory that doesn't contain empty intermediate directories. In this case *only*, cvsnt tries to "shorten" pathnames to avoid those empty directories.

For example, given a module **foo** that contains the file **bar.c**, the command **cvs co -d dir foo** will create directory **dir** and place **bar.c** inside. Similarly, given a module **bar** which has subdirectory **baz** wherein there is a file **quux.c**, the command **cvs -d dir co bar/baz** will create directory **dir** and place **quux.c** inside.

Using the **-N** flag will defeat this behavior. Given the same module definitions above, **cvs co -N -d dir foo** will create directories **dir/foo** and place **bar.c** inside, while **cvs co -N -d dir bar/baz** will create directories **dir/bar/baz** and place **quux.c** inside.

-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: **-jSymbolic_Tag:Date_Specifier**.

Chapter 6.

-b Perform the -j merge from the branchpoint not the last mergepoint. This can be useful to re-merge changes that have been merged before, however it likely to produce a lot of conflicts.

-m Perform the -j merge from the last recorded mergepoint. This is the default.

-N Only useful together with **-d dir**. With this option, cvsnt will not "shorten" module paths in your working directory when you check out a single module. See the **-d** flag for examples and a discussion.

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

-3 Produce 3-way conflict differences, containing the old and new files from the server and the edited files.

-S Select between conflicting case-sensitive names on a case-insensitive client. This provides limited support for checking out repositories with such conflicts - the problem should really be fixed in the repository.

-t Update using the last checkin time not the current time. This can cause issues with build systems so it is not recommended that this be used unless you are fully aware of the side-effects.

A.10.2 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.11 chown--Change directory owner

- Synopsis: chown [-R] user directory...
- Requires: working directory, repository
- Changes: repository.
- Synonyms: setowner

Change the owner of a directory. The owner has control access permissions (sometimes referred to as admin rights) over files within that directory. CVSNT administrators are always an implied owner.

A.11.1 chown options

-R Change directory owner recursively.

A.12 commit--Check files into the repository

- Synopsis: commit [-lnRf] [-m 'log_message' | -F file] [-r revision] [files...]
- Requires: working directory, repository.
- Changes: repository.
- Synonym: ci

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** (Section A.40). **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 (Section B.1, and Section B.8) and placed in the rcs file inside the repository. This log message can be retrieved with the **log** command; see Section A.21. 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.12.1 commit options

These standard options are supported by **commit** (Section A.5, for a complete description of them):

-D Assume all datestamps are different and send all files to the server for checking.

-l Local; run only in current working directory.

-n Do not run any module program.

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

commit also supports these options:

-F file Read the log message from `file`, instead of invoking an editor.

-f Note that this is not the standard behavior of the **-f** option as defined in Section A.5.

Force cvsnt 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
```

The **-f** option disables recursion (i.e., it implies **-l**). To force cvsnt to commit a new revision for all files in all subdirectories, you must use **-f -R**.

This command is also used when changing **-k** expansion options. Unless **-f** is specified modified options will not be propogated back to the server.

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

-c Check for a valid edit on the file before committing. See 'cvs edit'.

-b bugid Only commit files that have been edited and marked with bug `bugid`.

-B bugid Mark committed files as belonging to bug `bugid`.

-e Keep files edited after commit - supresses the automatic unedit that normally follows a commit.

-T Attempt to move branches rather than create branch revisions where possible. Where a branch has no revisions, this option will compare what you are trying to commit with the parent branch head, and if they are identical it will move the branch rather than create a new revision.

This option has detrimental affects on reproducability, for example:

In branch A, a developer is working with files a (version 1.1) and b (version 1.1.2.1). He merges from HEAD, then commits using this option, moving the branch A in file a to version 1.2, and committing a new file b version 1.1.2.2

Meanwhile manager B notices that that branch A is now broken, and asks the developer to fix it. The developer knows that the revsion 1.1.2.1 file works, so he wants to test with the older version. He can't. Since the environment that the older file was written in no longer exists (as the branchpoint has moved), it's impossible to roll back.

For this reason it is recommended that this option only be used where absolutely necessary, and on unrelated files only.

A.12.2 commit examples

A.12.2.1 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 (Chapter 6). 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.12.2.2 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 cvsnt 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.

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

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

A.13 diff--Show differences between revisions

- Synopsis: diff [-lR] [format_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 for diff is different than for other cvsnt commands; for details Section A.2.

A.13.1 diff options

These standard options are supported by **diff** (Section A.5, 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.

-k `kflag` Process keywords according to `kflag`. See Chapter 13.

This option is for use when diffing two repository revisions - it will probably not do what you expect when diffing against a sandbox file.

-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.

The following options specify the format of the output. They have the same meaning as in GNU diff.

```
-0 -1 -2 -3 -4 -5 -6 -7 -8 -9
--binary
--brief
--changed-group-format=arg
-c
  -C nlines
  --context[=lines]
-e --ed
-t --expand-tabs
-f --forward-ed
--horizon-lines=arg
--ifdef=arg
-w --ignore-all-space
-B --ignore-blank-lines
-i --ignore-case
-I regexp
   --ignore-matching-lines=regexp
-h
-b --ignore-space-change
-T --initial-tab
-L label
  --label=label
--left-column
-d --minimal
-N --new-file
--new-line-format=arg
--old-line-format=arg
--paginate
-n --rcs
-s --report-identical-files
-p
--show-c-function
-y --side-by-side
-F regexp
--show-function-line=regexp
-H --speed-large-files
--suppress-common-lines
-a --text
--unchanged-group-format=arg
-u
  -U nlines
  --unified[=lines]
-V arg
-W columns
  --width=columns
```

A.13.2 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.14 edit--Mark files for editing

- Requires: repository, sandbox.
- Changes: Current directory.
- Synonyms:

This is command is used to mark files for editing in a reserved or semi-reserved scenario. When used with bug identifiers it also marks which users are currently working on which bugs, and which files are affected by those bugs.

cvs is primarily designed as a cooperative system, as experience has shown that this is the most productive way for teams of developers to work. The reserved models implemented by this command do not replace proper configuration management processes - the correct model to use should be decided after due consideration of the advantages and disadvantages of each. See also Section 11.6

The default working model is a cooperative multiple-editors model. Any number of people may be editing a file at any time and anyone may commit changes. This is very similar to the standard cooperative model except that the server keeps track of the editors.

Using the **edit -c** option creates a semi-reserved or cooperative reserved system. The edit command checks that there are no editors before editing the file, however its cooperative nature does not prevent other users editing if they wish to.

For a stricter model the **-kc** and **-kx** expansion options create a mandatory reserved system on individual files. Users are prevented from editing or committing a file unless they are the only editor of that file.

A.14.1 edit options

-A Modify filesystem ACL on edited file (Win32 only, Experimental).

In shared sandbox scenarios this command is designed to stop other users from being able to modify the edited file - it create an access control list that disallows write access to anyone but the editor.

-a Specify actions for the temporary watch that is created during an edit. This may be one of edit,unedit,commit,all,none.

-b bugid Mark the edited file with a bug identifier. This marks the editor as not only using the file, but also working on a particular bug on that file. Specify multiple -b options for multiple bugs.

-c Check that working files are unedited before appying the edit.

-C Check that working file is unmodified.

-d Check that working files are up to date.

-f Edit even if working files are already edited by someone else (default).

-l Do not recurse into subdirectories.

-m message Specify a reason for this edit. This message is made available as an option to the trigger libraries and notify script.

-R Process directories recursively (default).

-w Edit the whole file, not just the current branch. Normally edits only apply to the branch that is current at the time of the edit. If you wish to stop anyone changing other branches then this option allow this.

-x Exclusive edit. Attempt to stop other users editing the file even if they do not use the **-c** option.

-z Edit creates copies of the original files in the `CVS/Base` directory. With this option those copies are gzip compressed, which saves disk space and stops the copies being found by text searches of the sandbox.

A.15 editors--Find out who is editing a file

- Requires: repository, sandbox.
- Changes: nothing.
- Synonyms:

This command queries the server for everyone editing a file or group of files. It can also optionally show which bugs are being worked on.

The server only knows the last reported state of each client. In a controlled environment this is very likely to be accurate, however it is possible to leave edits on the server and not on the client (for example my deleting the sandbox without using commit/unedit).

The normal editors output is as follows:

```
components.dir/TEST.xml   tmh   Fri Dec  3 16:15:00 2004 GMT   tucker   c:\temp\repos\ ↩
    components.dir
rep/version_no.h          tmh   Thu Nov  4 17:37:43 2004 GMT   tucker   D:\t\test\rep
```

If you list edits for all branches and bug identifiers you get an extra columns. The full output is as follows:

```
components.dir/TEST.xml   tmh   Fri Dec  3 16:15:00 2004 GMT   tucker   c:\temp\repos\ ↩
    components.dir   3465   HEAD
rep/version_no.h          tmh   Thu Nov  4 17:37:43 2004 GMT   tucker   D:\t\test\rep  ↩
                            HEAD
```

The first output is designed to be compatible with older cvs versions that did not support the full cvsnt feature set.

A.15.1 editors options

-a Show all branches, not just the current branch.

-c Check whether edit on the selected files would actually succeed. This can be used by frontends to verify an edit without actually performing it.

-l Process this directory only

-R Process directories recursively (default).

-v Show active bugs within the output.

A.16 export--Export sources from CVS, similar to checkout

- Synopsis: export [-flNnR] [-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 cvsnt 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 (and thus it always prunes empty directories).

One often would like to use **-kv** with **cvs export**. This causes any 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 keyword strings. If you want to be able to use **ident** you must not use **-kv**.

A.16.1 export options

These standard options are supported by **export** (Section A.5, 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. Section A.10.1, for complete details on how cvsnt handles this flag.

-k `subst` Set keyword expansion mode (Section 13.4).

-N Only useful together with **-d `dir`**. Section A.10.1, for complete details on how cvsnt handles this flag.

A.17 history--Show status of files and users

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

cvsnt 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 cvsnt (Section A.5).

A.17.1 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 **-x** with all record types. Of course, **-e** will also include record types which are added in a future version of cvsnt; if you are writing a script which can only handle certain record types, you'll want to specify **-x**.

-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. This is the default report type.

-T Report on all tags.

-x type Extract a particular set of record types `type` from the cvsnt 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

E export

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`.

-f file Show data for a particular file (you can specify several **-f** options on the same command line). This is equivalent to specifying the file on the command line.

-n module Show data for a particular module (you can specify several **-n** options on the same command line).

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

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

-t tag Show records since tag `tag` was last added to the history file. This differs from the **-r** 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`.

-z timezone Show times in the selected records using the specified time zone instead of UTC.

A.18 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. Chapter 14, for a discussion on this subject.

The `repository` argument gives a directory name (or a path to a directory) under the cvsnt 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 cvsnt decides a file should be ignored (Section B.19), it does not import it and prints **I** followed by the filename (Section A.18.2, 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. Section B.3.

The outside source is saved in a first-level 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 one argument is required. `repository` is needed to identify the collection of source. Normally also two other arguments are supplied - `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**.

Note that by default **import** does *not* change the directory in which you invoke it. In particular, it does not set up that directory as a cvsnt working directory. For initial imports the **-C** option will achieve this, but for vendor source imports you need to import them first and then check them out into a different directory (Section 1.3.1).

A.18.1 import options

This standard option is supported by **import** (Section A.5, for a complete description):

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

There are the following additional special options.

-b `branch` See Section 14.6.

-k `subst` Indicate the 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 13.4, 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. Section B.19.

If you specify '-I @' the contents of .cvsigore files are ignored for the import.

-W `spec` Specify file names that should be filtered during import. You can use this option repeatedly. To override all the default wrappers specify '-W !'.

`spec` can be a file name pattern of the same type that you can specify in the **.cvswrappers** file. Section B.3.

-C Create CVS directories during initial import. This provides simplified setup of a sandbox, however as it does not contact the server is not suitable for vendor updates - for this a proper import/checkout sequence should be used.

-i Ignore files with names that are illegal on windows.

-d Use the modification time of the file as the import time instead of the current time.

-f Overwrite any duplicate release tags within imported files.

-n Do not require vendor or release tags. This is used for initial imports only, and creates a repository without a vendor branch. If you are not planning to use vendor source imports then using this option simplifies the import process.

A.18.2 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 (Section B.19).

L `file` The file is a symbolic link; **cvs import** ignores symbolic links. People periodically suggest that this behavior should be changed, but if there is a consensus on what it should be changed to, it doesn't seem to be apparent. (Various options in the **modules** file can be used to recreate symbolic links on checkout, update, etc.; Section B.1.)

A.18.3 import examples

See Chapter 14, and Section 4.1.1.

A.19 init--Initialise a new repository

- Requires: local access.
- Changes: repository.
- Synonyms:

Initialises a new repository for use. This command can only be issued locally on the server, not remotely. See also Section 2.6

A.19.1 init options

Init can be called succesfully without any options.

-a alias Define the repository alias for the new repository.

-d description Set the repository description

-f Force overwrite of an existing repository. This doesn't normally make sense, so be careful with this option.

-r repository Remote repository creation. To successfully use this you must have an admin account on an existing repository, remote init must be initialized on the server, and the server needs the access rights to modify its global configuration.

-n Do not attempt repository registration

-u Unregister an existing repository. For this to succeed remotely the same conditions must exist as in -r.

A.20 info--Get information about the client and server

- Requires: nothing. (Repository required for server functions).
- Changes: nothing.
- Synonyms: inf

Return information about available protocols on the client or server. Also list the current cvsignore and cvswrappers settings.

Without any parameters this lists available protocols:

```
$ cvs info
Available protocols:

local               (internal)
ext                 ext 2.0.62.1872
fork                fork 2.0.62.1872
gserver             gserver 2.0.62.1872 (Active Directory)
ntserver            ntserver 2.0.62.1872
pserver             pserver 2.0.62.1872
server              server 2.0.62.1872
ssh                 ssh 2.0.62.1872
sspi                sspi 2.0.62.1872
```

For information about an individual protocol specify the prototocol name on the command line.

```
$ cvs info pserver
Name:               pserver
Version:            pserver 2.0.62.1872 (Debug)
Syntax:             :pserver[;keyword=value...]:[username[:password]@]host[:port][:]/path
  Username:         Optional
  Password:         Optional
  Hostname:         Required
  Port:             Optional
Client:             Yes
Server:             Yes
Login:              Yes
Encryption:         No
Impersonation:      CVS Builtin

Keywords available:

username            Username (alias: user)
password            Password (alias: pass)
hostname            Hostname (alias: host)
port                Port
proxy               Proxy server
proxyport           Proxy server port (alias: proxy_port)
tunnel              Proxy protocol (aliases: proxyprotocol,proxy_protocol)
proxyuser           Proxy user (alias: proxy_user)
proxypassword       Proxy passsord (alias: proxy_password)
```

The format is designed to be easily parsed by frontends. Its layout does not change between cvs versions, however lines may be added or deleted from the output.

Specifying **cvswrappers** or **cvsignore** dumps out the internal state of these files. It is possible to have duplicates, as the list is built up of both the client and server contents. When parsed however the client always takes precedence over the server setting.

A.20.1 info options

-c Return client-side information. Returns all protocols available to the client

-s Return server-side information. Returns protocols that a client can use to communicate with the server. This does not include local or external protocols.

-b (Where supported) list available cvsnt servers on the local network. This currently requires mdns support on the client.

-r server Find out as much as possible about a remote cvsnt server. For this command to succeed the remote server must support the cvsnt enumeration protocol.

```
$ cvs info -r cvs.cvsnt.org
Server: CVSNT Public Repository
Version: Concurrent Versions System (CVSNT) 2.5.01 (Travis) Build 2010

Protocols:
  gserver
  pserver
  sserver
  sspi

Repositories:
  /usr/local/cvs                CVSNT Main repository

Anonymous username: cvs
Anonymous protocol: pserver
Default repository: /usr/local/cvs

Anonymous login: :pserver:cvs@cvs.cvsnt.org:/usr/local/cvs
Recommended login: :sspi:cvs.cvsnt.org:/usr/local/cvs
```

The layout will remain the same as much as possible in future revisions to facilitate automatic parsing. Parsers should ignore elements that they do not understand.

A.21 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 cvsnt 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 cvsnt print times in the local timezone).

Warning: **log** uses **-R** in a way that conflicts with the normal use inside cvsnt (Section A.5).

A.21.1 log options

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

-B bugid Only select revisions which are related to a single bug.

-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 cvsnt commands (Section A.5). 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 name of the rcs file, name of the file in the working directory, 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 file.

-rrevisions 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).

rev1::rev2 Revisions between, but not including, `rev1` and `rev2`.

:rev Revisions from the beginning of the branch up to and including `rev`.

::rev Revisions from the beginning of the branch up to, but not including, `rev`.

rev: Revisions starting with `rev` to the end of the branch containing `rev`.

rev:: Revisions starting just after `rev` to the end of the branch containing `rev`.

branch An argument that is a branch means all revisions on that branch.

branch1:branch2, 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 Supress log output when no revisions are selected within a file.

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

-T Display dates and times in the log output in Local time rather than GMT.

-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.

-X Supress extended information generated only by CVSNT servers. This can be useful with some frontends that cannot parse the extra output.

-x Generate full output. This is the default unless configured otherwise on the server.

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

A.21.2 log examples

Contributed examples are gratefully accepted.

A.22 login--Cache a client password locally

- Requires: repository.
- Changes: local password cache.
- Synonyms: logon, lgn

Cache the password required for the client locally. Not all protocols require this, and some do not even support it. If you are using a protocol that does not require this then for security reasons it is better not to use it, since the local cache is relatively easy to find and decrypt if your local account/machine is compromised.

Do not make any assumptions about the storage of passwords in the local cache. In particular do not attempt to manipulate it manually - its format may change without warning.

See also Section 2.9

A.22.1 login options

-p password Specify the password to use (the default is to prompt).

A.23 logout--Remove the cached entry for a password

- Requires: repository.
- Changes: local password cache.
- Synonyms:

Destroy the password cache entry for the current connection. See also Section A.22

A.23.1 logout options

none.

A.24 ls--list modules, files and directories in the repository

- Requires: repository.
- Changes: nothing.
- Synonyms: dir,list,rls

Lists the contents of the repository, and optionally the latest version information from files within the repository.

Used without any parameters, it lists the toplevel directories (modules) in the repository. This includes directories created using the **modules2** file.

The list is followed by the contents of the **modules** file, if available.

A.24.1 ls options

-D date Show files current on a particular date.

-e Display in CVS/Entries format:

```
$ cvs ls -e CVSROOT
Listing module: CVSROOT

/checkoutlist/1.9/Wed Jan 26 19:08:06 2005/-kkv/
/commitinfo/1.10/Tue Jan 11 01:25:34 2005/-kkv/
/config/1.15/Sun Jan 23 02:15:57 2005/-kkv/
```

-l Display all details. Note that the usage of the -l option differs from other cvs commands. This is for consistency with the unix-style ls command.

```
$ cvs ls -l CVSROOT
Listing module: CVSROOT

checkoutlist                    1.9     Wed Jan 26 19:08:06 2005 -kkv
commitinfo                      1.10    Tue Jan 11 01:25:34 2005 -kkv
config                          1.15    Sun Jan 23 02:15:57 2005 -kkv
```

-P Ignore (Prune) empty directories.

-q Quieter output. Do not print extraneous human-readable prompts.

-R Recurse into subdirectories.

-r tag Show files with the specified revision, tag or branch.

-T Show timestamps in local time instead of GMT.

A.25 lsacl--Show file/directory permissions

- Synopsis: lsacl [-d] [-R] [file or directory...]
- Requires: repository, sandbox.
- Changes: nothing.
- Synonyms: lsattr,listperm

List the access control lists entries for files and directories within the current sandbox. For directories also shows the owner.

Permissions on a file or directory are also supplemented by parent directories, so the lack of mention of a user does not imply or deny access. For compatibility with older cvs versions the default is to grant permissions unless explicitly denied. This can be changed by putting an inheritable default deny permission in the repository root.

A.25.1 lsacl options

-d List only directories, not files. By default both file and directory permisisons are listed in the output.

-R Recursively list permissions in all subdirectories.

A.26 rlsacl--Show remote file/directory permissions

- Synopsis: rlsacl [-d] [-R] [module...]
- Requires: repository
- Changes: nothing.
- Synonyms: rlsattr,rlistperm

List permissions remotely, without reference to a local sandbox. See Section A.25

A.27 passwd--Modify a user's password or create a user

- Requires: repository.
- Changes: remote password file.
- Synonyms: password, setpass.

Change the username/password information for a user. This command is only useful for those protocols which do not use system passwords (such as pserver). It does not affect the real system password of the user.

Ordninary users are only able to change their own cvs password. Repository administrators can use the full funcitonality of this command.

If invoked without a username, the current username is used. If invoked with a username, the repository administrator can change the details of another user.

A.27.1 passwd options

-a Add user. Adds a new user entry to the password file.

-x Disable user. Changes the password so that the user cannot log in.

-X Delete user. Remove the user entry from the password file.

-r user Alias username to real system user. Before a virtual (pserver) user can log in the system needs to know which user account to use for that user.

-R Remove system alias for user.

-D domain (Win32 only) Use the users' domain password instead of a separate password. For security reasons this is not recommended.

A.28 rannotate--Show who made changes to remote files

- Requires: repository.
- Changes: nothing.
- Synonyms: rann, ra

Show changes on remote files within a repository. Does not require a sandbox. See also Section A.8

A.29 rchacl--Change remote access control lists

- Synopsis: rchacl [-R] [-r branch] [-u user] [-j branch] [-n] [-p priority] [-m message] [-a [no]{read|write|create|tag|control|all|none}[,...] [-d] [module...]
- Requires: repository.
- Changes: repository.
- Synonyms: rsetacl, rsetperm

Change an access control list on a remote file or directory. Does not require a sandbox. See also Section A.9

A.30 rchown--Change owner of a remote directory

- Synopsis: rchown [-R] user module...
- Requires: repository.
- Changes: repository.
- Synonyms: rsetowner

Change the owner of a remote directory. Does not require a sandbox. See also Section A.11

A.31 rdiff--'patch' format diffs between releases

- Synopsis: rdiff [-flags] [-V vn] [-r t|-D d [-r t2|-D d2]] modules...
- Requires: repository.
- Changes: nothing.
- Synonyms: patch, pa

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 cvsnt 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.31.1 rdiff options

These standard options are supported by **rdiff** (Section A.5, 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 Examine directories recursively. This option is on by default.

-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. 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 keywords according to the rules current in rcs version `vn` (the expansion format changed with rcs version 5). Note that this option is no longer accepted. cvsnt will always expand keywords the way that rcs version 5 does.

A.31.2 rdiff examples

Suppose you receive mail from *foo@example.net* asking for an update from release 1.2 to 1.4 of the tc compiler. You have no such patches on hand, but with cvsnt 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@example.net
```

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.32 release--Indicate that a Module is no longer in use

- Synopsis: release [-d [-f]] [-e] [-y] directories...
- Requires: Working directory.
- Changes: Working directory, history log.
- Synonyms: re, rel

This command is meant to safely cancel the effect of **cvs checkout**. Since cvsnt 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 cvsnt history file (Section B.21) 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 cvsnt 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 cvsnt history log.

A.32.1 release options

The **release** command supports the following command options:

-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.

-f Must be specified with -f, above. Force the the deletion of the directory even if non-cvs files are present.

-e Don't delete any files, just delete the cvsnt administrative directories. The directory is then left in a state as if it had just been **export**ed.

-y Automatically assume 'yes' to any confirmation prompts.

A.32.2 release output

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

A.32.3 release examples

Release the **tc** directory, 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 [5] altered files in this repository.
Are you sure you want to release (and delete) directory 'tc': y
$
```

A.33 remove--Remove files from the working directory

- Requires: working directory, repository.
- Changes: working directory.
- Synonyms: rm, delete

Remove a file from the working directory, marking the file as 'dead' which comes into effect after the next commit.

Files are never actually removed from the repository, only ever flagged as deleted. You can recover such a removed file by using a combinaiton of add and commit. See also Section 8.2

As a safety measure this command will not do anything unless the physical file is already deleted or you use the -f option.

A.33.1 remove options

-f Delete the physical file as well. Remove will not complete unless the file has already been deleted or this option is given.

-l Process this directory only.

-R Process directories recursively.

A.34 rename--Rename files in the repository

- Synopsis: rename [-q] source target
- Requires: working directory, repository.
- Changes: repository.
- Synonyms: ren,mv

Use the **rename** to rename or move a file within the sandbox, whilst keeping the history intact.

Rename functions depend on the capabilities of the server:

- CVS Suite 2008: limited support for rename, move not supported
- CVS Suite 2009R2: rename supported, move not supported
- CVS Suite 2010: rename supported, move available but not supported
- CM Suite 2008: rename and move supported
- CVSNT Community Edition 2.5.03, 2.0.51 etc: due to serious bugs, rename and move not supported
- CVS 1.11.x, 1.12.x etc: rename and move not available

Complex renames/move (across directories) should be avoided. If you must rename across directories then ensure that the 'destination' directory gets committed at the same time as the 'source' directory - otherwise the file can vanish completely.

Rename information is held at the directory level, so the rename/move is not committed to the repository until **cvs commit** is called on the directory containing the file.

If another user has the file checked out they will continue to use the file under its old name until they issue a **cvs update** at the directory level. CVSNT has no problems with this and both users can continue to merge each others' changes.

A.35 rlog--Return log history of remote file

- Requires: repository.
- Changes: nothing.
- Synonyms: rl

Return the log history of a remote file or group of files. Does not require a sandbox. See Section A.21

A.36 rtag--Mark a single revision over multiple files

- Requires: repository.
- Changes: nothing.
- Synonyms: rfreeze

Set a tag on a group of files in a repository. Does not require a sandbox. See Section A.38, also Section 5.4

A.37 status--Display the state of a file in the working directory

- Requires: repository, working directory.
- Changes: nothing.
- Synonyms: st,stat

Display the status of a file within the working directory. This includes any expansion options, its version, and whether it is modified or may require updating.

The normal output from the status command is as follows:

```
$ cvs status cvs.dbk
===================================================================
File: cvs.dbk           Status: Up-to-date

   Working revision:    1.1.2.36
   Repository revision: 1.1.2.36        /usr/local/cvs/cvsnt/doc/cvs.dbk,v
   Expansion option:    o
   Commit Identifier:   75a042064840566c
   Sticky Tag:          CVSNT_2_0_x (branch: 1.1.2)
   Sticky Date:         (none)
   Sticky Options:      -ko
   Merge From:          (none)
```

The layout of this output will remain the same across versions, although information may be added or removed.

A more terse form of status is produced by using the -q option, in which case only the checkout status is displayed:

```
$ cvs status -q cvs.dbk
File: cvs.dbk           Status: Up-to-date
```

A.37.1 status options

-v Verbose format. Append the tag information for each selected file.

-l Process this directory only.

-R Process directories recursively.

-q Display only a quick summary of the status of each file. Specifying a second **-q** option reduces the output still further, by supressing output for up to date files.

-X Display shorter output produced by cvs 1.x. This output may be required for parsing with older tools.

-x Display full cvsnt status details. Default, unless overridden on the server.

A.38 tag--Create a tag or branch

- Requires: repository, working directory.
- Changes: repository.
- Synonyms: ta,freeze

Create or modify a tag in the repository.

A tag is a snapshot of a single moment in time in the repository. Normally a tag would be applied to entire directories, although it is possible to tag individual files if required. See also Section 5.4

A branch is a unit of parallel development, which may or may not be kepy in sync with the main trunk. See also Chapter 6

Creating a tag or branch does not change the working directory. To create and work with a branch it is also necessary to use the cvs update command to move your working directory onto that branch.

A.38.1 tag options

-A Make an alias of an existing branch (requires -r). See Section 5.9

-b Make a branch tag.

-c Check that the working files are unmodified before tagging.

-d Delete the named tag. Deletion of branches is not recommended.

-F Move the tag if it already exists. Not recommended for branches.

-B Allow **-d** and **-F** to be applied to branch tags. Use of this option is not recommended as it does not affect the revisions within the branch and can result in them being orphaned.

-f Force a head revision match if the existing branch is not found.

-l Process local directory only.

-M Create a floating, or 'magic' branch. A floating branch always points to the head of its parent branch, unless a revision is checked into it. Once a revision is added it becomes a normal fixed branch.

-R Process directories recursively.

-r rev Select files based on existing tag/branch/revision.

-D date Select files current on a specific date.

A.39 unedit--Mark edit as finished without committing

- Requires: repository, working directory.
- Changes: working directory.
- Synonyms:

Discard any changes made and finish editing a file without committing. It may also be necessary to run an **update** command to retrieve the latest version of the file.

Unediting also sends out a notification to other users if the server is configured to do this. It will mark the working directory file as read only. See also Section 11.6

A.39.1 unedit options

-b bugid Unedit only files marked as edited with **bugid**.

-l Process local directory only.

-m message Specify the reason for this unedit. The message is sent to the **trigger** and **notify** programs on the server.

-r Revert file only. Do not perform unedit. This merely copies the unedited copy back onto the working copy.

-R Process directories recursively.

-u username (repository administrators only) perform an unedit for another user.

-w Leave working directory file writable after the unedit.

A.40 update--Bring work tree in sync with repository

- Requires: repository, working directory.
- Changes: working directory.
- Synonyms: up,upd

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.

It is unwise to let your local working directory become out of sync with others for too long. Depending on your working model it may be necessary to run updates daily or even hourly to keep in step. On the other hand if you are the only developer on a project it may not be necessary to update at all.

If updating is left too long, then conflicts that arise get progressively harder to fix over time as the code diverges. On the other hand frequent updating may mean that there are no conflicts to deal with at all.

A.40.1 update options

These standard options are available with **update** (Section A.5, for a complete description of them):

-e [bugid] Automatically edit modified/merged files.

-E [bugid] Automatically edit modified/merged files and unmodified files.

-D date Use the most recent revision no later than `date`. This option is sticky, and implies **-P**. See Section 5.11, 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).

-F `filename` Write clean repository copy to this filename. This is most often used when you need to perfom a visual side-by-side diff - for which you need two complete copies of the file, but where the 'name' of the file for that point in time may not be known. ie: you haVe the file with `filename` 'blat.htm' in your current directory and you want to compare it to the HEAD revision: but at HEAD it may have a different name (someone has used the **rename** command). You can use the **diff** without knowing the revisions filename, but you can't use the **checkout** unless you know the actual filename for that point in time. So this is the situation where this flag of the udpate command can be used.

-k `kflag` Process keywords according to `kflag`. See Chapter 13. 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.37, for more information on the **status** command.

-l Local; run only in current working directory. Chapter 7.

-P Prune empty directories. See Section 8.5.

-p Pipe files to the standard output.

-R Update directories recursively (default). Chapter 7.

-r rev Retrieve revision/tag `rev`. This option is sticky, and implies **-P**. See Section 5.11, for more information on sticky tags/-dates.

These special options are also available with **update**.

-3 Provide 3-way conflicts.

-A Reset any sticky tags, dates, or **-k** options. See Section 5.11, for more information on sticky tags/dates.

-B bugid Set the boundary of a -j merge to revisions marked with a particular bug. This is used to extract individual bug fixes from one branch to another.

If there are other revisions unrelated to the bug required to merge all the differences, these will also be merged. This option is much more useful in quiet or controlled repositories where this happens infrequently.

-b Perform the **-j** merge from the branchpoint, ignoring mergepoints.

-C Overwrite locally modified files with clean copies from the repository (the modified file is saved in **.#file.revision**, however).

-c If the file is edited, update the base revision copy to the latest revision. If this option is not used an unedit will always revert to the same revision that is edited, not the latest revision in the repository.

-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. Section B.19, for other ways to make cvsnt ignore some files.

-m Perform the **-j** merge based on the last mergepoint. This is the default.

-S Perform limited selection between conflicting case sensitive names on a case insensitive system. This option can be used to checkout files with conflicting names however it is not a solution to the problem - the conflict should be fixed in the repository.

-t Update using the last checkin time of the file not the current time. Do not use this option if you are using a makefile based system as it will cause problems with the build process. On other systems be aware of any side effects before using this option.

-Wspec Specify file names that should be filtered during update. You can use this option repeatedly. Use **-W !** avoid using the default wrappers. `spec` can be a file name pattern of the same type that you can specify in the **.cvswrappers** file. Section B.3.

-jrevision 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.

Note that using a single **-j tagname** option rather than **-j branchname** to merge changes from a branch will often not remove files which were removed on the branch. Section 6.9, for more.

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: **-jSymbolic_Tag:Date_Specifier**.

Chapter 6.

A.40.2 update output

update and **checkout** keep you informed of their 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.

P file Like **U**, but the cvsnt server sends a patch instead of an entire file. These two things accomplish the same thing.

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.

cvsnt 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 result of attempting to merge the two revisions; an unmodified copy of your file is also in your working directory, with the name **.#file.revision** where `revision` is the revision that your modified file started from. Resolve the conflict as described in Section 11.3. (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 cvsnt to ignore (see the description of the **-I** option, and Section B.19).

A.41 version--Display client and server versions.

- Requires: nothing.
- Changes: nothing.
- Synonyms: ve,ver

Display the version of the client in use. Also displays the version of the remote server if that information is available.

A.41.1 version options

-q Display only the version number of the local client, not other information.

A.42 watch--Watch for changes in a file

- Requires: repository, working directory.
- Changes: repository.
- Synonyms:

Add yourself to the list of watchers for a file. Watchers are notified via the **notify** script whenever an action they are interested in happens. See Section 11.6

A.42.1 watch options

-l Process local directory only.

-R Process directories recursively.

-a Specify what actions to watch. One of edit,unedit,commit,all,none.

A.43 watchers--list watched files

- Requires: repository, working directory.
- Changes: nothing.
- Synonyms:

Display the list of files that are being watched, and what is being watched about them. See Section 11.6

A.43.1 watchers options

-l Process local directory only.

-R Process directories recursively.

A.44 xdiff--External diff

- Requires: repository, working directory.
- Changes: nothing.
- Sysnonyms: xd

Run an external diff defined by the **cvswrappers** file on the server. The output and options for this option vary depending on what is run on the server-side diff, however the common options are listed below.

A.44.1 xdiff options

-D date Diff revision for date against working file. Specifying the **-D** option twice causes the diff to be against the two dated revisions instead of the working file.

-N Also diff added and removed files.

-R Process directories recursively.

-l Process local directory only.

-o xdiff-options Pass extra arguments and options to the external xdiff program.

-r rev Diff revision or tag against working file. Specifying a second **-r** option causes the diff to be against two specified revisions instead of the working file.

Appendix B

Reference manual for Administrative files

Inside the repository, in the directory **$REAL_CVSROOT/CVSROOT**, there are a number of supportive files for cvsnt. You can use cvsnt 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 2.4.

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. cvsnt will use these definitions if you use cvsnt 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.

There are three basic types of modules: alias modules, regular modules, and ampersand modules. The difference between them is the way that they map files in the repository to files in the working directory. In all of the following examples, the top-level repository contains a directory called **first-dir**, which contains two files, **file1** and **file2**, and a directory **sdir**. **first-dir/sdir** contains a file **sfile**.

B.1.1 Alias modules

Alias modules are the simplest kind of module:

`mname -a aliases...` This represents the simplest way of defining a module `mname`. The **-a** flags the definition as a simple alias: cvsnt 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 cvsnt arguments.

For example, if the modules file contains:

```
amodule -a first-dir
```

then the following two commands are equivalent:

```
$ cvs co amodule
$ cvs co first-dir
```

and they each would provide output such as:

```
cvs checkout: Updating first-dir
U first-dir/file1
U first-dir/file2
cvs checkout: Updating first-dir/sdir
U first-dir/sdir/sfile
```

B.1.2 Regular modules

mname [options] dir [files...] 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.

For example, if a module is defined by:

```
regmodule first-dir
```

then regmodule will contain the files from first-dir:

```
$ cvs co regmodule
cvs checkout: Updating regmodule
U regmodule/file1
U regmodule/file2
cvs checkout: Updating regmodule/sdir
U regmodule/sdir/sfile
$
```

By explicitly specifying files in the module definition after `dir`, you can select particular files from directory `dir`. Here is an example:

```
regfiles first-dir/sdir sfile
```

With this definition, getting the regfiles module will create a single working directory **regfiles** containing the file listed, which comes from a directory deeper in the cvsnt source repository:

```
$ cvs co regfiles
U regfiles/sfile
$
```

B.1.3 Ampersand modules

A module definition can refer to other modules by including **&module** in its definition.

```
mname [ options ] &module...
```

Then getting the module creates a subdirectory for each such module, in the directory containing the module. For example, if modules contains

```
ampermod &first-dir
```

then a checkout will create an **ampermod** directory which contains a directory called **first-dir**, which in turns contains all the directories and files which live there. For example, the command

```
$ cvs co ampermod
```

will create the following files:

```
ampermod/first-dir/file1
ampermod/first-dir/file2
ampermod/first-dir/sdir/sfile
```

There is one quirk/bug: the messages that cvsnt prints omit the **ampermod**, and thus do not correctly display the location to which it is checking out the files:

```
$ cvs co ampermod
cvs checkout: Updating first-dir
U first-dir/file1
U first-dir/file2
cvs checkout: Updating first-dir/sdir
U first-dir/sdir/sfile
$
```

Do not rely on this buggy behavior; it may get fixed in a future release of cvsnt.

B.1.4 Excluding directories

An alias module may exclude particular directories from other modules by using an exclamation mark (**!**) before the name of each directory to be excluded.

For example, if the modules file contains:

```
exmodule -a !first-dir/sdir first-dir
```

then checking out the module **exmodule** will check out everything in **first-dir** except any files in the subdirectory **first-dir/sdir**.

B.1.5 Module options

Either regular modules or ampersand modules can contain options, which supply additional information concerning the module.

-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 **verifymsg** 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**. It is not run when **tag** is executed. Generally you will find that taginfo is a better solution (Section 9.3).

B.1.6 How the modules file "program options" programs are run

For checkout, rtag, and export, the program is server-based, and as such the following applies:-

If using remote access methods (pserver, ext, etc.), cvsnt will execute this program on the server from a temporary directory. The path is searched for this program.

If using "local access" (on a local or remote NFS filesystem, i.e. repository set just to a path), the program will be executed from the newly checked-out tree, if found there, or alternatively searched for in the path if not.

The programs are all run after the operation has effectively completed.

B.2 The modules2 file

The **modules2** file provides a lower level definition of modules than the **modules** file. Clients see the **modules2** structure as if it existed physically on the server.

B.2.1 How the modules2 file differs from the modules file

The **modules** provides different types of module, which are 'high level', in that checking out a module is equivalent to calling checkout multiple times on different directories. This approach works well for simple cases, but breaks down in the more complex cases, causing unwanted interactions with the **update** command for example.

The **modules2** has only one way of describing a module, but operates on a much lower level. Clients are unaware that the directory structure that they are checking out does not actually exist, and all cvs commands behave as normal. A file or directory defined by modules2 may have a completely different name to its real name, and updates/merging will be handled correctly even if multiple clients checkout under different names.

Which file you choose depends on your requirements. It isn't recommend that usage is mixed between the two files as they both serve a similar function and it would get confusing.

B.2.2 Modules2 syntax

The **modules2** file is structured in a similar way to the familiar Windows .ini file. Each section defines a module, and within each section is a description of the files and directories within that module.

An example modules2 file is:

```
[pets]
dog
cat

[people]
brother
sister

[household]
pets
people
```

Checking out 'household' will create the directory structure:

```
household
  pets
    dog
    cat
  people
    brother
    sister
```

In this example the 'household', 'pets', and 'people' directories don't have any files in them - they're just containers. However let's say we want to put the files listing pet food in the pets directory, above all the pet specific directories.

Modules2 lets you override what goes in the root of a module, to overlay another module in it:

```
[pets]
/ = !petfood
dog
cat

[people]
brother
sister

[household]
pets
people
```

Now when we checkout we get the same directory structure as above, but the pets directory contains the contents of 'petfood'. Note also we said that we don't want any subdirectories of petfood, using the '!' prefix. This makes sure that the directory is never recursed into, even during an update -d. We still get the 'dog' and 'cat' directory of course.

You can simply rename an entire directory tree using this method. The following:

```
[project1]
/ = myproject

[project2]
/ = myproject
junk =
total_junk =
project/old_project = myproject/junk
```

project1 will checkout the entire myproject tree. project2 is the same, except the 'junk' directory is removed, and moved to project/oldproject. The total_junk directory is hidden completely.

You can also mask certain files within a directory, or certain subdirectories using an extended regular expression.

```
[project1]
/ = myproject

[project2]
/ = myproject (*\.cpp$|*\.[ch]$|*/$)
junk =
total_junk =
project/old_project = myproject/junk
```

Directories are subject to the same filtering, except they have a '/' directory separator after their name. If you just want to filter some files and allow subdirectories then add '|*/$' as an option.

(need to be more verbose here: FIXME)

The '+' prefix stops processing, so that entries that would be potentially recursive can be defined to be nonrecursive.

Spaces can be used in the file, delimited by quotes or using backslash escapes. File separators must always be forward slashes.

Comments are on a line beginning '#'

B.3 The cvswrappers file

Wrappers refers to a cvsnt feature which lets you control certain settings based on the name of the file which is being operated on. The settings are **-k** for binary files, **-x** to define xdiff wrappers, **-t** to override default mime types, and **-m** for nonmergeable text files.

The basic format of the file **cvswrappers** is:

```
wildcard     [option value][option value]...

where option is one of
-m           update methodology      value: MERGE or COPY
-k           keyword expansion       value: expansion mode
-x           xdiff specification     value: name of xdiff DLL, plus options.
-t           mime type               value: new mime type

and value is a single-quote delimited value.
```

For example, the following command imports a directory, treating files whose name ends in **.exe** as binary:

```
cvs import -I ! -W "*.exe -kb" first-dir vendortag reltag
```

The **-m** option specifies the merge methodology that should be used when a non-binary file is updated. **MERGE** means the usual cvsnt behavior: try to merge the files. **COPY** means that **cvs update** will refuse to merge files, as it also does for files specified as binary with **-kb** (but if the file is specified as binary, there is no need to specify **-m 'COPY'**). cvsnt will provide the user with the two versions of the files, and require the user using mechanisms outside cvsnt, to insert any necessary changes. *WARNING*: do not use **COPY** with cvsnt 1.9 or earlier-such versions of cvsnt will copy one version of your file over the other, wiping out the previous contents. The **-m** wrapper option only affects behavior when merging is done on update; it does not affect how files are stored. See Chapter 10, for more on binary files.

The -x option specifies the external diff program used when the cvs xdiff command is used. It is followed by the name of the xdiff library (which must always be relative to the predefined library root), and any optional parameters that the xdiff library requires. For example, to pass all .txt files through GNU diff:

```
*.txt -x "ext_xdiff diff -u --label \"%label1%\" \"%file1%\" \"%file2%\""
```

B.3.1 default wrappers

There are some kinds of files which are nearly always binary, and these have been given default wrappers of **-kb** within cvs. You can override these wrappers using **-W !** on the command line or specifying **!** in the first line of your cvswrappers files. If you do override these be sure to warn your users, who may be expecting default behaviour.

```
*.a   *.avi  *.bin  *.bmp  *.bz2  *.class  *.dll  *.exe  *.gif
*.gz  *.hqx  *.ilk  *.lib  *.jar  *.jpg    *.jpeg *.mpg  *.mpeg
*.mov *.mp3  *.ncb  *.o    *.ogg  *.obj    *.pdb  *.pdf  *.png
*.ppt *.res  *.rpm  *.sit  *.so   *.tar    *.tgz  *.tif  *.tiff
*.wmv *.xls  *.zip
```

In addition, a file pattern of ***.*** or just ***** will be used as a default where no wrappers exist. This pattern may also contain addive or subtractive wrapper options (eg -k+x), in which case it will always be applied.

B.4 The commit support files

The **-i** flag in the **modules** file can be used to run a certain program whenever files are committed (Section B.1). 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.

triggers This is the main control file that cvs uses. Most of the other commit support files are implemented through a trigger (known as the default trigger).

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.

verifymsg The specified program is used to evaluate 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 (Section B.15).

The verifymsg script may or may be able to change the log message depending on the value of the ReredLogAfterVerify setting in the **config** file.

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!

precommand The specified program is called prior to command execution, and is passed the list of arguments supplied to the command. Returning an error from this script will terminate command execution.

postcommand The specified program is called when the command is complete, and all locks have been released from the repository prior to returning to the user. This is useful to maintain checked-out copies of repositories and to perform cvs actions after a commit.

premodule The specified program is called prior to entering a module. It is passed the repository, command and module name. Returning an error from this script will terminate command execution.

Not all commands use the premodule/postmodule scripts, only those which take modules as arguments, eg. checkout, repository commands such as rlog, rtag, etc.

postmodule The specified program is called when module processing is complete. It is passed the repository, command and module name.

Not all commands use the premodule/postmodule scripts, only those which take modules as arguments, eg. checkout, repository commands such as rlog, rtag, etc.

postcommit The specified program is called when a commit is complete, and all locks have been released from the repository prior to returning to the user. This is useful to maintain checked-out copies of repositories and to perform cvs actions after a commit.

This script has been largely superceded by the **postcommand** script.

historyinfo This is called when any action which causes an entry in the history file is initiated. Its standard input receives the line to be written to the history log, in a semi-compressed format.

notify When required this file is called durin a commit, and also during edit/unedit.

keywords Define non-standard or user defined keyword mappings

script.vbs, script.js, script.pl, script.py, script.rb (Win32 only) ActiveScript trigger file. Implemented through the script trigger library (script_trigger.dll).

shadow Define shadow or checked-out workspaces on the server. Implemented by the checkout trigger library.

commit_email Define templates for automatic sending of emails during commit.

tag_email Define templates for automatic sending of emails during tag

notify_email Define templates for automatic sending of emails during notify

B.4.1 The common syntax

The administrative files such as **commitinfo**, **loginfo**, **rcsinfo**, **verifymsg**, etc., 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. This uses Perl-Compatible regular expression syntax (PCRE). On case insensitive servers this expression is also case insensitive.
- A whitespace separator--one or more spaces and/or tabs.

- **shell command** the commonest form, invokes a shell script using the parameters described below.

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.

Each regular expression that matches the current directory name in the repository is considered for use. The first match will be used, followed by any matching lines that are prefixed by '+'. The rest of the line after the regular expression is used as a file name or command-line as appropriate.

Directory and filename separators must use Unix conventions (forward slashes) on all platforms.

All files have a set of default parameters (see the description of each file for details). In addition a number of common parameters are available to be passed to each file.

%c Command currently being executed.

%d Date and time of executing command.

%h Remote host name, on client/server connections.

%u Current cvs username.

%r virtual repository name (repository alias, to be displayed to the user).

%R Physical repository name (for file access).

%S Session ID or commit ID.

%e Current value of $CVSEDITOR.

%H Local host name.

%P Local directory (temporary directory in client/server connections).

%i Client version string if supplied by the client.

%n An empty string.

%% % sign

$... CVSNT environment variable (see Appendix C), System environment variable, or User variable (see Section B.24).

You can group parameters into a single script argument using the **%{}** syntax.

Parameters can be sent either to the script command line, or its standard input. Options for standard input begin with **%<**, as below.

%<option Send value of option to standard input.

%<{options} Send value of options to standard input on a single line.

%<< text for standard input\nhello %u!\n Send the following string to standard input. This must be the last item on the line.

%<<TAG Send everything in the file until the next occurence of **TAG** on its own to standard input. This must be the last item on the line.

B.5 Triggers

The triggers file defines a binary interface to the cvsnt server. The calling parameters to this interface are listed in a separate document (The CVS Library Interface).

Each line of the file lists the location of the a trigger library to be loaded on server startup. On Win32 this line may also be the Class ID of a COM object resident on the server.

As the server goes through each phase of operation, each trigger is called in turn. In addition a default library (called default_trigger) is called, which dispatches scripts contained in each of the files mentioned below.

B.6 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. A number of extra options are available in the commitinfo file beyond those in the generic list.

%s File name(s) that this invocation is using.

%m Commit message supplied by the user

%p Current directory name, relative to the root.

The default pattern used if none is specified is **%r/%p %<s**, which causes the full path to the current source repository to be 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 directory within the repository 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 occurrences 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 cvsnt is accessing a remote repository, **commitinfo** will be run on the *remote* (i.e., server) side, not the client side (Section 2.9).

B.7 Verifying

Once you have entered a log message, you can evaluate that message to check for specific content, such as a bug ID. Use the **verifymsg** file to specify a program that is used to verify the log message. This program could be a simple script that checks that the entered message contains the required fields.

The **verifymsg** file is often most useful together with the **rcsinfo** file, which can be used to specify a log message template.

Each line in the **verifymsg** file consists of a regular expression and a command-line template. Each line can have any combination of the following, in addition to those listed in the common syntax.

%p Directory name relative to the current root.

%l Full path to file containing the log message.

The template must include a program name, and can include any number of arguments. If no other formatting is used **%l** is automatically added which appends the full path to the current log message file file to the 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.

If the verification script exits with a non-zero exit status, the commit is aborted.

Note that the verification script cannot change the log message; it can merely accept it or reject it.

The following is a little silly example of a **verifymsg** file, together with the corresponding **rcsinfo** file, the log message template and an verification 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.verify** is used to evaluate the log message.

```
#!/bin/sh
#
#       bugid.verify filename
#
#  Verify that the log message contains a valid bugid
#  on the first line.
#
if head -1 < $1 | grep '^BugId:[ ]*[0-9][0-9]*$' > /dev/null; then
    exit 0
else
    echo "No BugId found."
    exit 1
fi
```

The **verifymsg** file contains this line:

```
^tc     /usr/cvssupport/bugid.verify
```

The **rcsinfo** file contains this line:

```
^tc     /usr/cvssupport/tc.template
```

B.8 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.

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 occurrences 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.

Section B.4, for a description of the syntax of the **loginfo** file.

The user may specify a format string as part of the filter. The string is composed of a **%** followed by a space, or followed by a single format character, or followed by a set of format characters surrounded by **{** and **}** as separators. The format characters are those in the common syntax plus:

%m Message supplied by user

%T Status string

%p Directory name relative to the current root

%s Module name, followed by the list of filenames. When used in a group this option has a special action which is designed to mimic previous versions of cvs when the standard **%{sVv}** is used.

%V Current version, pre-checkin.

%v Current version, post-checkin.

%b Bug identifier

%t Tag

%y Type

All other characters that appear in a format string expand to an empty field (commas separating fields are still provided).

For example, some valid format strings are **%**, **%s**, **%{s}**, and **%{sVv}**.

By default the standard input is a formatted string which mimics the behaviour of older CVS versions (see below).

For a commit the command line will be a string of tokens separated by spaces. For backwards compatibility, the first token will be the repository subdirectory. The rest of the tokens will be comma-delimited lists of the information requested in the format string. For example, if **/u/src/master/yoyodyne/tc** is the repository, **%{sVv}** is the format string, and three files (*ChangeLog*, *Makefile*, *foo.c*) were modified, the output might be:

```
yoyodyne/tc ChangeLog,1.1,1.2 Makefile,1.3,1.4 foo.c,1.12,1.13
```

As another example, **%{}** means that only the name of the repository will be generated.

When run as part of an import or add directory, the command line the repository subdirectory followed by the text

- New Directory

or

- Imported Sources

Note: when cvsnt is accessing a remote repository, **loginfo** will be run on the *remote* (i.e., server) side, not the client side (Section 2.9).

B.8.1 Loginfo example

The following **loginfo** file, together with the tiny shell-script below, appends all log messages to the file **$REAL_CVSROOT/CVSROOT** and any commits to the administrative files (inside the **CVSROOT** directory) are also logged in **/usr/adm/cvsroot-log**. Commits to the **prog1** directory are mailed to *ceder*.

```
ALL             /usr/local/bin/cvs-log $REAL_CVSROOT/CVSROOT/commitlog $USER
^CVSROOT        /usr/local/bin/cvs-log /usr/adm/cvsroot-log
^prog1          Mail -s %s ceder
```

The shell-script **/usr/local/bin/cvs-log** looks like this:

```
#!/bin/sh
(echo "-------------------------------------------------------";
 echo -n $2"  ";
 date;
 echo;
 cat) >> $1
```

B.8.2 Loginfo default standard input format

For both commit and import the first two lines are the following:

```
Update of %r/%p
In directory %H:%P
```

Next part is different for import and commit. For commits there comes line with the current operation/operations, namely "Added Files:", "Removed Files:" or "Modified Files:". In the next lines there are, indented with TAB, space separated list of added, removed or modified files. There is no such section for added directories (because one can remove empty directories only with checkout/update with -P option, not by commit). There is instead "Directory $CVSROOT/subdirectory added to repository" log message. For import next part (separated by empty line) is the log message:

```
Log Message:
%m
```

This part is also after commit, but for commit it is at the very end of input, and is _not_ separated by an empty line. Further parts are for import solely. After log message, separated by empty line comes:

```
Status:
```

Next is the information about release and vendor tag (see 'cvs import' syntax), separated of course from log message by an empty line, namely

```
Vendor Tag: vendor_tag
Release Tags: release_tag
```

Next, separated by an empty line, is the output of the import command. The format is

```
X module_dir/subdir/file
```

where X is one letter indicator of status. The last line is the status of import command, e.g.

```
No conflicts created by this import
```

B.9 Precommand

Prior to each cvs command, this file is called to validate the command arguments. The following formatting strings are available in addition to those in the common syntax:

%a List of arguments passed on the command line

By default **%r %c %<a** is used, which passes the repository, command name, and all the command arguments in the standard input. An error (non-zero) return from this script will abort the command.

B.10 postcommand

After a command has completed, the actions in this file are executed so that you can perform operations on the repository before returning. The following formatting strings are available in addition to those in the common syntax:

%p Last directory affected, relative to repository root

By default the string **%r/%p %c** is used, which passes the current directory and command on the command line.

If a command uses multiple modules, the module name used to select the line in the postcommand script is undefined. For this reason it is recommended to use only the **DEFAULT** and **ALL** lines.

Typical uses of this would be to keep a checked-out copy of a repository.

B.11 premodule

Before parsing a module, this script is called with the command name and module, to validate or log the use of the module. Returning nonzero from this script will terminate the operation. The following formatting strings are available in addition to those in the common syntax:

%o Logical module name

By default the string **%r/%p %c %o** is used, which passes the current directory, command and module on the command line.

Not all command pass through premodule. In particular sandbox-related commands use the information in CVS/Repository and do not use the module system.

B.12 postmodule

This script is called after module processing is completed. It is passed the command name and module. The following strings are available in addition to those in the common syntax:

%o Logical module name

By default the string **%r/%p %c %o** is used, which passes the current directory, command and module on the command line.

As modules in CVSROOT/modules may be defined recursively, it is possible that many premodule calls will be made in sequence before the first postmodule call. Scripts must be written to handle this when it occurs.

B.13 postcommit

This script is called after a commit has completed. Its use has largely been superceded by the use of the postcommand script. The following formatting strings are available in addition to those in the common syntax:

%p Directory relative to root of last directory committed.

By default the string **%r/%p** is used, which passes the last directory committed.

If multiple modules or repositories are committed the module in effect when this file is parsed is undetermined, so it is recommended that only the DEFAULT and ALL lines are used.

B.14 historyinfo

This script is called whenever a new line is to be written to the history file. As this can happen frequently it is not recommended that the script interface for this command be used as it will slow down server operations. The following formatting strings are available in addition to those in the common syntax:

%t History entry type

%w Working directory, truncated to history format

%v Affected revisions

%s Name of affected file

By default the string **%t|%d|%u|%w|%s|%v** is used, which is the same as the line written to the history file.

B.15 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 **verifymsg**, **commitinfo** and **loginfo** files. Section B.4.1. 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 occurrences 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.

Section B.7, for an example **rcsinfo** file.

When cvsnt is accessing a remote repository, the contents of **rcsinfo** at the time of the last update are used. If you edit **rcsinfo** or its templates, you may need to update your working directory.

B.16 notify

The notify file is called whenever a watched file is changed during commit, edit or unedit. See also Section 11.6 for more details.

In addition to the formatting strings in the common syntax, the following formatting strings are available:

%m Message supplied by user.

%b Bug identifier

%p Directory name relative to current root

%s User being notified

%t Tag or branch of file being notified

%y Type of notification

%f File being notified about

By default the notify file has a string designed to be compatible with older versions of cvs passed in its standard input:

```
%<< %p %f
---
Triggered %y watch on %r
By %u
```

B.17 keywords

The keywords file contains user defined mappings of the standard rcs keywords (see Chapter 13).

For each module listed the first line defines the module(s) that the keyword list applies to, then on subsequent lines the keywords are listed, indented by at least one space. The special module name **ALL** refers to all modules, and is used by default.

```
ALL
  Maintainer   Joe Bloggs
^Foo$
  Maintainer   Fred Bloggs
```

In the example above, Joe Bloggs maintains the repository and his brother Fred maintains the Foo module on his own. The rcs tag $Maintainer$ will be expanded differently depending on the location of any source files.

You can also redefine or remove RCS tags, for example when tracking third party sources it may be desirable for the RCS tags from the imported sources to remain intact.

```
ALL
  Id
  LocalId    %f %v %d %a %s
```

Listing a keyword with no definition disables its standard usage. In the example above Id will no longer be considered as an RCS keyword, and $LocalId$ has been defined with the pattern previously assigned to Id.

When redefining keywords you can use any variable listed in the common syntax, plus the following:

%p Path relative to repository root

%f Filename

%a Author

%d Date formatted in human readable format

%D Date formatted in RCS format

%l File locker (not normally useful in cvsnt)

%s State (normally one of **Exp** or **Dead**)

%v Version number

%N RCS $Name$ value

%b Bug identifier(s) identified with file

%C Commit/Session identifier

%t Branch that revision is a member of

The standard RCS keywords are defined as follows:

```
ALL
  Author     %a
  Date       %d
  Header     %r/%p/%f %v %d %a %s
  CVSHeader  %p/%f %v %d %a %s
  Id         %f %v %d %a %s
  Locker     %l
  Log        %f
  Name       %N
  RCSfile    %f
  Revision   %v
  Source     %r/%p/%f
  State      %s
  CommitId   %C
  Branch     %t
```

B.17.1 Storing user defined information using keywords

The keywords file is parsed on checkout, which means it isn't useful for storing environment variables or other dynamic data. To store such data user defined variables (see Section B.24) can be used.

For example, using the following keywords file

```
ALL
  Weather    $WEATHER
```

the definition of WEATHER be defined during commit as a user defined variable

```
$ cvs -s WEATHER=Sunny commit -m "Fix stuff" foo.c
```

This is then associated with that revision for future checkouts.

B.18 Email notification

CVSNT contains a trigger library which is capable of sending notification emails on commit, tag or notify. It allows you to put any contents in the emails, but the output format is fairly simple - it is no substitute for a purpose designed notification program.

Email sending is disabled by default. To configure it for use you must do the following.

B.18.1 Configure the commit support files

The commit support files **commit_email**, **tag_email** and **notify_email** contain the names of the template files to use for commit, tag and notify respectively. Each line in these files is a regular expression followed by a filename. The filename is always relative to the CVSROOT directory and may not be an absolute path for security reasons.

The first matching line for each directory committed is used. If there is no match the DEFAULT line is used.

The name of the template file should also be **listed in the checkoutlist file** so that it is available for the script to use.

The **CVSROOT/users** file is used to lookup the username -> email mapping. This file is a list of colon separated username/email pairs. If this file does not exist or the username is not listed the default domain name set in the global configuration is used.

B.18.2 Write the template

The template file is a simple text file listing the exact text of the email to send including headers. The To:, From:, Cc: and Bcc: lines are used by the sending software to determing the addresses to use.

An example commit template is:

```
From: [email]
To: cvsnt_users@mycompany.com
Subject: Commit to [module]

CVSROOT:     [repository]
Module name: [module]
Changes by:  [email]           [date]
On host:     [hostname]

[begin_directory]
Directory: [directory]

[begin_file]
[change_type] [filename]   [tag]   [old_revision] -> [new_revision]   [bugid]
[end_file]
[end_directory]

Log message:
[message]
```

A number of replacements are done on the file to format it for final sending. This differs for each file, and is listed below.

B.18.3 Configure the server

There are two ways that CVSNT can send email. The simplest is to set the SMTP Server and default domain in the global configuration (Control Panel in win32, /etc/cvsnt/PServer in Unix) and let the internal SMTP client send the emails.

This will not work in the case where authenticaton is required or the server is not capable of SMTP. In these cases you instead set the Email Command. This command should take a list of 'to' addresses as parameters, and a raw RFC822 email as its standard input.

A suitable configuration for Unix systems is

```
/usr/sbin/sendmail -i
```

Similar programs exist for Win32.

B.18.4 Keywords used in template files

The following are the global keywords used in the email template files.

[hostname] Client hostname, it known

[repository] Repository name

[commitid] or [sessionid] Session identifier

[server_hostname] Local hostname of server

[user] User who is performing the action

[email] Email address, as looked up from **CVSROOT/users**

[date] Date/time of action

[message] Message associated with action, if any

[module] Module associated with action

[begin_directory] .. [end_directory] Lines between these tags (which must be on their own on the line) are repeated for each directory referenced by the operation.

[begin_file] .. [end_file] Lines between these tags (which must be on their own on the line) are repeated for each file referenced by the operation. These tags can only exist inside begin/end directory tags.

[directory] Current directory

[filename] Curent file

Each type of template also has its own keywords that is uses:

B.18.4.1 commit emails

[old_revision] Revision number of previous revision.

[new_revision] Revision number of new revision.

[tag] Tag for file.

[change_type] Code for change made by this commit.. 'M', 'A', etc.

B.18.4.2 tag emails

[tag_type] Type of tag operation.

[action] What is being done with the tag.

[tag] Tag for file

[revision] File revision that is being tagged.

B.18.4.3 notify emails

[bugid] Bug identifier(s) associated with this notification.

[tag] Tag name of file

[notify_type] Notification type

[to_user] Mapped user/e-mail to notify from CVSROOT/users file

B.19 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 cvsnt 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 (Section A.40.2).

cvsnt 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 cvsnt 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:

```
. .. core RCSLOG tags TAGS RCS SCCS .make.state .nse_depinfo #* .#* cvslog.* ,*
CVS CVS.adm .del-* *.a *.olb *.o *.obj *.so *.Z *~ *.old *.elc *.ln *.bak *.BAK
*.orig *.rej *.exe *.dll *.pdb *.lib *.ncb *.ilk *.exp *.suo .DS_Store _$* *$
*.lo *.pch *.idb *.class ~*
```

- The per-repository list in **$REAL_CVSROOT/CVSROOT/cvsignore** is appended to the list, if that file exists.
- The per-user list in **.cvsignore** in your home directory is appended to the list, if it exists.
- Any entries in the environment variable **$CVSIGNORE** is appended to the list.
- Any **-I** options given to cvsnt is appended.
- As cvsnt 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 cvsnt.

Specifying **-I !** to **cvs import** will import everything, which is generally what you want to do if you are importing files from a pristine distribution or any other source which is known to not contain any extraneous files. However, looking at the rules above you will see there is a fly in the ointment; if the distribution contains any **.cvsignore** files, then the patterns from those files will be processed even if **-I !** is specified. The only workaround is to remove the **.cvsignore** files in order to do the import. Because this is awkward, in the future **-I !** might be modified to override **.cvsignore** files in each directory.

Note that the syntax of the ignore files consists of a series of lines, each of which contains a space separated list of filenames. This offers no clean way to specify filenames which contain spaces, but you can use a workaround like **foo?bar** to match a file named **foo bar** (it also matches **fooxbar** and the like). Also note that there is currently no way to specify comments.

B.20 The checkoutlist file

It may be helpful to use cvsnt to maintain your own files in the **CVSROOT** directory. For example, suppose that you have a script **logcommit.pl** which you run by including the following line in the **commitinfo** administrative file:

```
ALL   $REAL_CVSROOT/CVSROOT/logcommit.pl
```

To maintain **logcommit.pl** with cvsnt you would add the following line to the **checkoutlist** administrative file:

```
logcommit.pl
```

The format of **checkoutlist** is one line for each file that you want to maintain using cvsnt, giving the name of the file.

Files in checkoutlist must always be relative to and below CVSROOT. Attempting to checkout files outside these constraints is an error.

After setting up **checkoutlist** in this fashion, the files listed there will function just like cvsnt's built-in administrative files. For example, when checking in one of the files you should get a message such as:

```
cvs commit: Rebuilding administrative file database
```

and the checked out copy in the **CVSROOT** directory should be updated.

Note that listing **passwd** (Section 2.9.4.1) in **checkoutlist** is not allowed for security reasons.

For information about keeping a checkout out copy in a more general context than the one provided by **checkoutlist**, see Section B.22.1.

B.21 The history file

The file **$REAL_CVSROOT/CVSROOT/history** is used to log information for the **history** command (Section A.17). This file must be created to turn on logging.

The file format of the **history** file is documented only in comments in the cvsnt source code, but generally programs should use the **cvs history** command to access it anyway, in case the format changes with future releases of cvsnt.

For new installations it is preferred to setup auditiing rather than use the history file. In a future release the history file will be deprecated.

B.22 The shadow file

The CVSROOT/shadow file is used by the checkout plugin to specify directories that will be automatically updated on checkout or tag.

In order for the shadow file to have any effect the "Automatic checkout extension" plugin must be enabled in the CVSNT Server control panel.

B.22.1 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 cvsnt and want every checkin to cause the files used by the web server to be updated.

The way to do this is by having a line in the **CVSROOT/shadow**. Here is an example (this should all be on one line):

```
^cyclic-pages HEAD /u/www/local-docs
```

This will cause checkins to repository directories starting with **cyclic-pages** to update the checked out tree in **/u/www/local-docs**.

Note that if the shadow copy does not exist already it will be created by the execution of the shadow command. Likewise if a shadow copy exists and new directories have been added to the module then these directories and files will also be checked out into the shadow copy. So it will always be a true representation of the current state of the module.

The shadow file works only on the physical file system level (inside the repository). This means that a module specified in the regular expression must match a physical module name in order to be recognized.

For example if you have created virtual modules inside the CVSROOT/modules file or CVSROOT/modules2 file you cannot specify such a module name in the shadow file.

B.23 ActiveScript support

On Windows platforms (Windows 2000 and later), cvsnt can use the builtin active scripting support to produce simple scripts.

To enable this functionality, enable the 'ActiveScript Plugin' using the CVSNT Server Control Panel, and **cvs add** a file to the checked out CVSROOT as follows:

- `script.vbs` for VBScript support
- `script.js` for JavaScript support
- `script.pl` for PerlScript support
- `script.rb` for RubyScript support

The scripting engine you intend to use must be installed on the server - only the VBScript engine is installed by default.

Available functions are as follows:

```
init(command, current_date, hostname, username, virtual_repository, physical_repository,  ↩
    session_id, editor, user_variable_array, client_version, character_set)
close
taginfo(message,directory, file_array, action, tag)
verifymsg(directory, filename)
loginfo(message, status, directory, change_array)
history(history_type, workdir, revs, name, bugid, message, dummy)
notify(message, bugid, directory, notify_user, tag, notify_type, file)
precommit(name_list, message, directory)
postcommit(directory)
precommand(argument_list)
postcommand(directory)
premodule(module)
postmodule(module)
get_template(directory)
parse_keyword(keyword, directory, file, branch, author, printable_date, rcs_date, locker,  ↩
    state, version, name, bugid, commitid, global_properties, local_properties)
prercsdiff(file, directory, oldfile, newfile, diff_type, options, oldversion, newversion)
rcsdiff(file, directory, oldfile, newfile, diff, diff_type, options, oldversion, newversion ↩
    , added, removed)
```

Functions should return 0 if successful. Arguments are strings, except where the name ends in _list which are arrays of strings, or _array, which are associative arrays (name=value).

The exceptions to this is `loginfo`, which is passed an array of structures containing `filename`, `rev_old` and `rev_new`. Also the `get_template` and `parse_keyword` functions, which are expected to return a string or `null`.

The script has access to a server object, called `Server`, which contains the following functions:

```
Trace(tracelevel,message)
Warning(message)
Error(message)
```

An example of all these functions is available in the cvsnt source tree, called `script.vbs` in the `contrib_nt` directory.

B.24 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 cvsnt is running in. There are several mechanisms to do that.

To find the home directory of the user running cvsnt (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 reasonable expansion if pserver (Section 2.9.4) is in use; therefore user variables (see below) may be a better choice to customize behavior based on the user running cvsnt.

One may want to know about various pieces of information internal to cvsnt. A cvsnt internal variable has the syntax **${variable}**, where `variable` starts with a letter and consists of alphanumeric characters and _. If the character following `variable` is a non-alphanumeric character other than _, the **{** and **}** can be omitted. The cvsnt internal variables are:

CVSROOT This is the name of the current cvsnt repository as the user sees it.

VIRTUAL_CVSROOT This is the name of the current cvsnt repository as the user sees it.

REAL_CVSROOT This is the physical location of the current cvsnt repository. Avoid displaying this value to users as it is an information leak.

CVSEDITOR

VISUAL

EDITOR These all expand to the same value, which is the editor that cvsnt is using. Section A.4, for how to specify this.

USER Username of the user running cvsnt (on the cvsnt server machine). When using pserver, this is the user specified in the repository specification which need not be the same as the username the server is running as (Section 2.9.4.1).

CVSPID Parent process ID of the cvsnt process.

SESSIONID

COMMITID Unique Session ID of cvsnt process. This is a random string of printable characters that may be up to 256 characters long.

If you want to pass a value to the administrative files which the user who is running cvsnt 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 cvsnt, with argument **variable=value**. It may be particularly useful to specify this option via **.cvsrc** (Section A.3).

For example, if you want the administrative file to refer to a test directory you might create a user variable **TESTDIR**. Then if cvsnt 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**.

Environment variables passed to administrative files are:

CVS_USER The cvsnt-specific username provided by the user, if it can be provided (currently just for the pserver access method), and to the empty string otherwise. (CVS_USER and USER may differ when **$REAL_CVSROOT/CVSROOT/passwd** is used to map cvs usernames to system usernames.)

B.25 The CVSROOT/config configuration file

The administrative file **config** contains various miscellaneous settings which affect the behavior of cvsnt. The syntax is slightly different from the other administrative files. Variables are not expanded. Lines which start with **#** are considered comments. Other lines consist of a keyword, **=**, and a value. Note that this syntax is very strict. Extraneous spaces or tabs are not permitted.

Currently defined keywords are:

SystemAuth=value If `value` is **yes**, then pserver should check for users in the system's user database if not found in **CVSROOT/passwd**. If it is **no**, then all pserver users must exist in **CVSROOT/passwd**. The default is **yes**. For more on pserver, see Section 2.9.4.

TopLevelAdmin=value Modify the **checkout** command to create a **CVS** directory at the top level of the new working directory, in addition to **CVS** directories created within checked-out directories. The default value is **no**.

This option is useful if you find yourself performing many commands at the top level of your working directory, rather than in one of the checked out subdirectories. The **CVS** directory created there will mean you don't have to specify **CVSROOT** for each command. It also provides a place for the **CVS/Template** file (Section 2.3).

AclMode=value Select the access control list mode. One of 3 values:

- **none** - No extra access control is done on this repository.

- **compat** (default) - Default access mode is to allow access.
- **normal** - Default access mode is to deny access.

LockDir=`directory` This option is ignored unless the lockserver is disabled. It is accepted only for compatibility with older systems. This option will be removed in the near future.

Put cvsnt lock files in `directory` rather than directly in the repository. This is useful if you want to let users read from the repository while giving them write access only to `directory`, not to the repository. You need to create `directory`, but cvsnt will create subdirectories of `directory` as it needs them. For information on cvsnt locks, see Section 11.5.

Before enabling the LockDir option, make sure that you have tracked down and removed any copies of cvsnt 1.9 or older. Such versions neither support LockDir, nor will give an error indicating that they don't support it. The result, if this is allowed to happen, is that some cvsnt users will put the locks one place, and others will put them another place, and therefore the repository could become corrupted. CVS 1.10 does not support LockDir but it will print a warning if run on a repository with LockDir enabled.

LockServer=`hostname`[:port] Uses the cvsnt lock server to handle locking rather than using files in the repository. This is useful if you want to let users read from the repository while giving them write access only to `directory`, not to the repository. For information on cvsnt locks, see Section 11.5. cvsnt 2.0.15 and above use the LockServer by default and other methods of locking are depreciated. You can override this behaviour by using the line **LockServer=none**. Note however that future versions may not allow this override. See also Section 3.11

LogHistory=`value` Control what is logged to the **CVSROOT/history** file. Default of **TOFEWGCMAR** (or simply **all**) will log all transactions. Any subset of the default is legal. (For example, to only log transactions that modify the ***,v** files, use **LogHistory=TMAR**.)

ReereadLogAfterVerify=`value` If enabled the log message parsed by **verifymsg** is reread after the script has run. The default behavoiur is to not reread this file.

Watcher=`name` Set a watcher who sees all edit/unedit/commit notifications via the **CVSROOT/notify** script. The watcher sees all notificaitons regardless of an existing edit/watch on the file, which for a large commit could be a lot of files. It is therefore recommended that the notify script completes as fast a possible. Using a custom trigger library or COM interface is recommended for best efficiency.

B.26 The server configuration files

The CVSNT global server configuration contains information with affects all repositories.

On Win32, this information is stored in the registry and is normaly only manipulated via the cvsnt server control panel. On Unix, a text file, normally **/etc/cvsnt/PServer** is used to store most of this information.

See the **PServer.example** file for the list of current available settings.

Appendix C

All environment variables which affect CVS

This is a complete list of all environment variables that affect cvsnt.

$CVSIGNORE A whitespace-separated list of file name patterns that cvsnt should ignore. Section B.19.

$CVSWRAPPERS A whitespace-separated list of file name patterns that cvsnt should treat as wrappers. Section B.3.

$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.

$CVSUMASK Controls permissions of files in the repository. See Section 2.2.2.

$CVSROOT Should contain the full pathname to the root of the cvsnt source repository (where the rcs files are kept). This information must be available to cvsnt for most commands to execute; if **$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, cvsnt 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, $VISUAL Specifies the program to use for recording log messages during commit. **$CVSEDITOR** overrides **$EDITOR**. See Section 1.3.2.

$PATH If **$rcsBIN** is not set, and no path is compiled into cvsnt, it will use **$PATH** to try to find all programs it uses.

$HOME

$HOMEPATH

$HOMEDRIVE Used to locate the directory where the **.cvsrc** file, and other such files, are searched. On Unix, cvsnt just checks for **HOME**. On Windows, the system will set **HOMEDRIVE**, for example to **d:** and **HOMEPATH**, for example to **\joe**. On Windows 95, you'll probably need to set **HOMEDRIVE** and **HOMEPATH** yourself.

$CVS_EXT

$CVS_RSH Specifies the external program which cvsnt connects with, when **:ext:** access method is specified. This replaces the **CVS_RSH** environment used in older implementations of cvs.

The **CVS_EXT** variable parsed as a formatting string specifying the command to pass to invoke the remote server. The default string is: **ssh -l %u %h** The **%u** parameter is replaced with the username specified in the CVSROOT (or the current username if none is specified) and the **%h** parameter is replaced with the hostname specified in the CVSROOT.

The CVS_EXT string has the string ' cvs server' appended to it, and this is then passed to the command processor for execution.

Section 2.9.2.

$CVS_CLIENT_PORT Used in client-server mode when accessing the server via Kerberos, GSSAPI, or cvsnt's password authentication if the port is not specified in $CVSROOT. Section 2.9

$CVS_CLIENT_LOG Used for debugging only in client-server mode. If set, everything sent to the server is logged into **$CVS_CLIENT_LOG.in** and everything sent from the server is logged into **$CVS_CLIENT_LOG.out**.

$CVS_SERVER_LOG Used for debugging only in client-server mode. If set, everything sent to the server is logged into **$CVS_SERVER_LOG.in** and everything sent from the server is logged into **$CVS_SERVER_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 specified amount of seconds so that you can attach to it with a debugger.

$CVS_DIR Used by the client to find the lockserver when automatically executing it. If not defined the client looks in the system path.

$CVSLIB Location of the libraries and protocol DLLs used by cvsnt. Not used on Win32.

$CVSCONF Location of the global configuration settings file. Not used on Win32.

$COMSPEC Used under DOS/Windows and 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. The cvsnt server uses **TMPDIR**. Section A.4, for a description of how to specify this. Some parts of cvsnt will always use **/tmp** (via the **tmpnam** function provided by the system).

On Windows, **TMP** is used (via the **_tempnam** function provided by the system).

The **patch** program which is used by the cvsnt client uses **TMPDIR**, and if it is not set, uses **/tmp** (at least with GNU patch 2.1). Note that if your server and client are both running cvsnt 1.9.10 or later, cvsnt will not invoke an external **patch** program.

Appendix D

Compatibility between CVS Versions

It is always possible to upgrade from an earlier version of CVS or CVSNT to a newer version. Downgrading however is not guaranteed to work. In particular downgrading from CVSNT 2.x to CVS 1.x will require some work on the repository format as many features are unsupported in the older version.

The working directory format is compatible going back to cvsnt 1.5. It did change between cvsnt 1.3 and cvsnt 1.5. If you run cvsnt 1.5 or newer on a working directory checked out with cvsnt 1.3, cvsnt will convert it, but to go back to cvsnt 1.3 you need to check out a new working directory with cvsnt 1.3.

Support for the Entries.Extra file varies between versions, however this should not normally affect client operations. Client versions of CVSNT before 2.0.55 used a **Baserev** file to store edit information. If downgrading a client existing edits may be lost.

The remote protocol is interoperable going back to cvsnt 1.5, but no further (1.5 was the first official release with the remote protocol, but some older versions might still be floating around). In many cases you need to upgrade both the client and the server to take advantage of new features and bugfixes, however.

When changing between platforms care should be taken to avoid platform-specific issues. RCS files are always in the same format and are interoperable, however the CVSROOT control files are often written specifically for the platform, and will need to be updated.

The Win32 port of CVSNT is a fully native application and does not require cygwin. It is not recommended that cygwin CVS and CVSNT are installed on the same machine as confusion and incompatibities may arise between versions.

CVSNT obeys the CYGWIN environment variable when deciding where to store extended permissions. However its default is ntea not ntsec, so this will be needed to be specified if using both environments. The recommended setting is CYGWIN="ntea nontsec tty" which will force both CYGWIN and CVSNT to use the same permissions structure.

Appendix E

Troubleshooting

If you are having trouble with cvsnt, this appendix may help. If there is a particular error message which you are seeing, then you can look up the message alphabetically. If not, you can look through the section on other problems to see if your problem is mentioned there.

E.1 Partial list of error messages

Here is a partial list of error messages that you may see from cvsnt. It is not a complete list--cvsnt is capable of printing many, many error messages, often with parts of them supplied by the operating system, but the intention is to list the common and/or potentially confusing error messages.

The messages are alphabetical, but introductory text such as **cvs update:** is not considered in ordering them.

In some cases the list includes messages printed by old versions of cvsnt (partly because users may not be sure which version of cvsnt they are using at any particular moment).

cvs `command`: authorization failed: server `host` rejected access This is a generic response when trying to connect to a pserver server which chooses not to provide a specific reason for denying authorization. Check that the username and password specified are correct and that the **CVSROOT** specified is allowed by **/etc/cvsnt/PServer**. See Section 2.9.4.

`file` 'text' failed The exact format of this message may vary depending on your system. It indicates a bug in cvsnt, which can be handled as described in Appendix G.

cvs `command`: conflict: removed `file` was modified by second party This message indicates that you removed a file, and someone else modified it. To resolve the conflict, first run **cvs add `file`**. If desired, look at the other party's modification to decide whether you still want to remove it. If you don't want to remove it, stop here. If you do want to remove it, proceed with **cvs remove `file`** and commit your removal.

cannot change permissions on temporary directory

```
Operation not permitted
```

This message has been happening in a non-reproducible, occasional way when we run the client/server testsuite, both on Red Hat Linux 3.0.3 and 4.1. We haven't been able to figure out what causes it, nor is it known whether it is specific to linux (or even to this particular machine!). If the problem does occur on other unices, **Operation not permitted** would be likely to read **Not owner** or whatever the system in question uses for the unix **EPERM** error. If you have any information to add, please let us know as described in Appendix G. If you experience this error while using cvsnt, retrying the operation which produced it should work fine.

cvs [server aborted]: Cannot check out files into the repository itself The obvious cause for this message (especially for non-client/server cvsnt) is that the cvsnt root is, for example, **/usr/local/cvsroot** and you try to check out files when you are in a subdirectory, such as **/usr/local/cvsroot/test**. However, there is a more subtle cause, which is that the temporary directory on the server is set to a subdirectory of the root (which is also not allowed). If this is the problem, set the temporary directory to somewhere else, for example **/var/tmp**; see **TMPDIR** in Appendix C, for how to set the temporary directory.

cannot open CVS/Entries for reading: No such file or directory This generally indicates a cvsnt internal error, and can be handled as with other cvsnt bugs (Appendix G). Usually there is a workaround--the exact nature of which would depend on the situation but which hopefully could be figured out.

cvs [init aborted]: cannot open CVS/Root: No such file or directory This message is harmless. Provided it is not accompanied by other errors, the operation has completed successfully. This message should not occur with current versions of cvsnt, but it is documented here for the benefit of cvsnt 1.9 and older.

cvs [checkout aborted]: cannot rename file `file` to CVS/,,`file`: Invalid argument This message has been reported as intermittently happening with cvsnt 1.9 on Solaris 2.5. The cause is unknown; if you know more about what causes it, let us know as described in Appendix G.

cvs [`command` aborted]: cannot start server via rcmd This, unfortunately, is a rather nonspecific error message which cvsnt 1.9 will print if you are running the cvsnt client and it is having trouble connecting to the server. Current versions of cvsnt should print a much more specific error message. If you get this message when you didn't mean to run the client at all, you probably forgot to specify **:local:**, as described in Chapter 2.

ci: `file`,v: bad diff output line: Binary files - and /tmp/T2a22651 differ cvsnt 1.9 and older will print this message when trying to check in a binary file if rcs is not correctly installed. Re-read the instructions that came with your rcs distribution and the install file in the cvsnt distribution. Alternately, upgrade to a current version of cvsnt, which checks in files itself rather than via rcs.

cvs checkout: could not check out `file` With cvsnt 1.9, this can mean that the **co** program (part of rcs) returned a failure. It should be preceded by another error message, however it has been observed without another error message and the cause is not well-understood. With the current version of cvsnt, which does not run **co**, if this message occurs without another error message, it is definitely a cvsnt bug (Appendix G).

cvs [login aborted]: could not find out home directory This means that you need to set the environment variables that cvsnt uses to locate your home directory. See the discussion of **HOME**, **HOMEDRIVE**, and **HOMEPATH** in Appendix C.

cvs update: could not merge revision `rev` of `file`: No such file or directory cvsnt 1.9 and older will print this message if there was a problem finding the **rcsmerge** program. Make sure that it is in your **PATH**, or upgrade to a current version of cvsnt, which does not require an external **rcsmerge** program.

cvs [update aborted]: could not patch `file`: No such file or directory This means that there was a problem finding the **patch** program. Make sure that it is in your **PATH**. Note that despite appearances the message is *not* referring to whether it can find `file`. If both the client and the server are running a current version of cvsnt, then there is no need for an external patch program and you should not see this message. But if either client or server is running cvsnt 1.9, then you need **patch**.

cvs update: could not patch `file`; will refetch This means that for whatever reason the client was unable to apply a patch that the server sent. The message is nothing to be concerned about, because inability to apply the patch only slows things down and has no effect on what cvsnt does.

dying gasps from `server` unexpected There is a known bug in the server for CVS 1.9.18 and older which can cause this. For me, this was reproducible if I used the **-t** global option. It was fixed by Andy Piper's 14 Nov 1997 change to src/filesubr.c, if anyone is curious. If you see the message, you probably can just retry the operation which failed, or if you have discovered information concerning its cause, please let us know as described in Appendix G.

end of file from server (consult above messages if any) The most common cause for this message is if you are using an external **rsh** program and it exited with an error. In this case the **rsh** program should have printed a message, which will appear before the above message. For more information on setting up a cvsnt client and server, see Section 2.9.

cvs [update aborted]: EOF in key in rcs file `file`,v, cvs [checkout aborted]: EOF while looking for end of string in rcs file `file` This means that there is a syntax error in the given rcs file. Note that this might be true even if rcs can read the file OK; cvsnt does more error checking of errors in the rcs file. That is why you may see this message when upgrading from CVS 1.9 to CVS 1.10. The likely cause for the original corruption is hardware, the operating system, or the like. Of course, if you find a case in which cvsnt seems to corrupting the file, by all means report it, (Appendix G). There are quite a few variations of this error message, depending on exactly where in the rcs file cvsnt finds the syntax error.

cvs commit: Executing 'mkmodules' This means that your repository is set up for a version of cvsnt prior to cvsnt 1.8. When using cvsnt 1.8 or later, the above message will be preceded by

```
cvs commit: Rebuilding administrative file database
```

If you see both messages, the database is being rebuilt twice, which is unnecessary but harmless. If you wish to avoid the duplication, and you have no versions of cvsnt 1.7 or earlier in use, remove **-i mkmodules** every place it appears in your **modules** file. For more information on the **modules** file, see Section B.1.

missing author Typically this can happen if you created an rcs file with your username set to empty. cvsnt will, bogusly, create an illegal rcs file with no value for the author field. The solution is to make sure your username is set to a non-empty value and re-create the rcs file.

cvs [checkout aborted]: no such tag `tag` This message means that cvsnt isn't familiar with the tag `tag`. Usually this means that you have mistyped a tag name; however there are (relatively obscure) cases in which cvsnt will require you to try a few other cvsnt commands involving that tag, before you find one which will cause cvsnt to update the **val-tags** file; see discussion of val-tags in Section 2.2.2. You only need to worry about this once for a given tag; when a tag is listed in **val-tags**, it stays there. Note that using **-f** to not require tag matches does not override this check; see Section A.5.

***PANIC* administration files missing** This typically means that there is a directory named cvsnt but it does not contain the administrative files which cvsnt puts in a CVS directory. If the problem is that you created a CVS directory via some mechanism other than cvsnt, then the answer is simple, use a name other than cvsnt. If not, it indicates a cvsnt bug (Appendix G).

rcs error: Unknown option: -x,v/ This message will be followed by a usage message for rcs. It means that you have an old version of rcs (probably supplied with your operating system), as well as an old version of cvsnt. CVS 1.9.18 and earlier only work with rcs version 5 and later; current versions of cvsnt do not run rcs programs.

cvs [server aborted]: received broken pipe signal This message seems to be caused by a hard-to-track-down bug in cvsnt or the systems it runs on (we don't know--we haven't tracked it down yet!). It seems to happen only after a cvsnt command has completed, and you should be able to just ignore the message. However, if you have discovered information concerning its cause, please let us know as described in Appendix G.

Too many arguments! This message is typically printed by the **log.pl** script which is in the **contrib** directory in the cvsnt source distribution. In some versions of cvsnt, **log.pl** has been part of the default cvsnt installation. The **log.pl** script gets called from the **loginfo** administrative file. Check that the arguments passed in **loginfo** match what your version of **log.pl** expects. In particular, the **log.pl** from cvsnt 1.3 and older expects the logfile as an argument whereas the **log.pl** from cvsnt 1.5 and newer expects the logfile to be specified with a **-f** option. Of course, if you don't need **log.pl** you can just comment it out of **loginfo**.

cvs [update aborted]: unexpected EOF reading `file,v` See **EOF in key in rcs file**.

cvs [login aborted]: unrecognized auth response from `server` This message typically means that the server is not set up properly. For example, if **cvsmanager** is running a bad cvs executable. To debug it further, find the log file which inetd writes (**/var/log/messages** or whatever inetd uses on your system). For details, see Section E.2, and Section 2.9.4.1.

cvs server: cannot open /root/.cvsignore: Permission denied, cvs [server aborted]: can't chdir(/root): Permission denied See Section E.2.

cvs commit: Up-to-date check failed for `file' This means that someone else has committed a change to that file since the last time that you did a **cvs update**. So before proceeding with your **cvs commit** you need to **cvs update**. cvsnt will merge the changes that you made and the changes that the other person made. If it does not detect any conflicts it will report **M `file`** and you are ready to **cvs commit**. If it detects conflicts it will print a message saying so, will report **C `file`**, and you need to manually resolve the conflict. For more details on this process see Section 11.3.

Usage: diff3 [-exEX3 [-i | -m] [-L label1 -L label3]] file1 file2 file3

```
Only one of [exEX3] allowed
```

This indicates a problem with the installation of **diff3** and **rcsmerge**. Specifically **rcsmerge** was compiled to look for GNU diff3, but it is finding unix diff3 instead. The exact text of the message will vary depending on the system. The simplest solution is to upgrade to a current version of cvsnt, which does not rely on external **rcsmerge** or **diff3** programs.

warning: unrecognized response `text' from cvs server** If `text` contains a valid response (such as **ok) followed by an extra carriage return character (on many systems this will cause the second part of the message to overwrite the first part), then it probably means that you are using the **:ext:** access method with a version of rsh, such as most non-unix rsh versions, which does not by default provide a transparent data stream. In such cases you probably want to try **:server:** instead of **:ext:**. If `text` is something else, this may signify a problem with your cvsnt server. Double-check your installation against the instructions for setting up the cvsnt server.

cvs commit: [time] waiting for user's lock in directory This is a normal message, not an error. See Section 11.5, for more details.

cvs commit: warning: editor session failed This means that the editor which cvsnt is using exits with a nonzero exit status. Some versions of vi will do this even when there was not a problem editing the file. If so, point the **CVSEDITOR** environment variable to a small script such as:

```
#!/bin/sh
vi $*
exit 0
```

E.2 Trouble making a connection to a CVS server

This section concerns what to do if you are having trouble making a connection to a cvsnt server. If you are running the cvsnt command line client running on Windows, first upgrade the client to cvsnt 1.9.12 or later. The error reporting in earlier versions provided much less information about what the problem was. If the client is non-Windows, cvsnt 1.9 should be fine.

If the error messages are not sufficient to track down the problem, the next steps depend largely on which access method you are using.

:ext: Try running the ssh program from the command line. For example: "ssh servername cvs -v" should print cvsnt version information. If this doesn't work, you need to fix it before you can worry about cvsnt problems.

:server: You don't need a command line rsh program to use this access method, but if you have an rsh program around, it may be useful as a debugging tool. Follow the directions given for :ext:.

:pserver: Errors along the lines of "connection refused" typically indicate that cvsmanager isn't even listening for connections on port 2401 whereas errors like "connection reset by peer" or "recv() from server: EOF" typically indicate that cvsmanager is listening for connections but is unable to start cvsnt . Another less common problem is invisible control characters that your editor "helpfully" added without you noticing.

One good debugging tool is to "telnet servername 2401". After connecting, send any text (for example "foo" followed by return). If cvsnt is working correctly, it will respond with

```
cvs [authserver aborted]: bad auth protocol start: foo
```

If it fails to work at all, then make sure cvsmanager is working right.

On AIX systems, the system will often have its own program trying to use port 2401. This is AIX's problem in the sense that port 2401 is registered for use with cvsnt. I hear that there is an AIX patch available to address this problem.

:gserver: If you cannot connect using gserver, ensure that your kerberos installation is correctly configured. You will need a working PAM configuraiton if your system uses that, and nsswitch.conf may need to be configured to recognise kerberos users.

Kerberos is rather difficult to configure, and it is beyond the scope of this manual. There are many resources on the internet to help you with this.

Appendix F

Credits

March Hare Software has been the vendor of CVSNT since 2004 and has provided hundreds of thousands of dollars worth of development resources, plus IT infrastruture (computers, internet, servers etc), office space and resources (power, internet etc.) to the project.

```
http://www.march-hare.com/cvspro/
```

In addition to the staff, there are many contributors to the open Source CVS, CVSNT, CVSWEB, CVSWEBNT, WINCVS, TORTOISECVS, and BUGZILLA projects whose tremendous effort and support has helped the development of CVSNT.

Tony Hoyle has been instrumental in the development of CVSNT and this manual, both as a valued member of staff and as a volunteer.

For a more complete list of who has contributed to this manual see the file **doc/ChangeLog** in the cvsnt source distribution.

Appendix G

Dealing with bugs in CVS or this manual

Neither cvsnt nor this manual is perfect, and they probably never will be. If you are having trouble using cvsnt, or think you have found a bug, there are a number of things you can do about it. Note that if the manual is unclear, that can be considered a bug in the manual, so these problems are often worth doing something about as well as problems with cvsnt itself.

- For support please contact: *sales@march-hare.com*.

```
March Hare Software UG (haftungsbeschränkt)
Europadamm 4,
Neuss,
20128
Germany
+49 (0) 2131 5952927
```

Or in North America please contact: *sales.us@march-hare.com*.

```
March Hare Software LLC
200 Broadhollow Rd. Ste 207,
Melville, NY
11747
USA
800.653.1501
```

Or in Australia please contact: *sales.au@march-hare.com*.

```
March Hare Pty Ltd
PO Box 139
Narrabeen, NSW
2101
Australia
+61 2 8212 4409
```

The mailing list is no longer used for support, but the history is maintained online as well as the current bug database:

```
http://www.cvsnt.org/pipermail/cvsnt/
http://www.cvsnt.org/tt/
http://www.march-hare.com/pipermail/cvsnt/
http://customer.march-hare.com/webtools/bugzilla/tt.htm
```

- March Hare Software provide worldwide support including telephone support phone numbers in the USA, Germany, UK and Australia. Security update notification, patches, installation and training are also included. March Hare can guarantee this support because the people who have been developing CVS since 1999 work for us.

  At the time of writing, support is available from US$27.88 per licensed user per year for CVS Suite customers (CVS Suite is available for $139.40 ea.).

- If you got cvsnt through a distributor, such as an operating system vendor or a vendor of freeware cd-roms, you may wish to see whether the distributor provides support. Often, they will provide no support or minimal support, but this may vary from distributor to distributor.

- If you have the skills and time to do so, you may wish to fix the bug yourself using the supplied source code. Please submit your fix to *support@march-hare.com* for inclusion in future releases of cvsnt.

- There may be resources on the net which can help. A good place to start is:

  ```
  http://www.cvsnt.org/
  ```

  or

  ```
  http://www.march-hare.com/cvspro/
  ```

  .

Chapter 17

Index

_
!, in modules file, 107
-a, in modules file, 105
-d, in modules file, 107
-e, in modules file, 107, 108
-i, in modules file, 107, 108
-j (merging branches), 37
-j (merging branches), and keyword substitution, 39
-k (keyword substitution), 60
-o, in modules file, 107, 108
-s, in modules file, 107
-t, in modules file, 107, 108
-u, in modules file, 108
.# files, 102
.bashrc, setting CVSROOT in, 6
.cshrc, setting CVSROOT in, 6
.cvsrc file, 69
.profile, setting CVSROOT in, 6
.tcshrc, setting CVSROOT in, 6
/etc/cvsnt/PServer, 126
/usr/local/cvsroot, as example repository, 6
:ext:, setting up, 15
:ext:, troubleshooting, 133
:fork:, setting up, 19
:gserver:, setting up, 19
:gserver:, troublshooting, 133
:local:, setting up, 6
:pserver:, setting up, 18
:pserver:, troubleshooting, 133
:server:, setting up, 15
:server:, troubleshooting, 133
:sserver:, setting up, 18
:ssh:, setting up, 15
:sspi:, setting up, 18
<<<<<<<, 53
=======, 53
>>>>>>>, 53
#cvs.lock, removing, 53
#cvs.lock, technical details, 9
#cvs.rfl, and backups, 14
#cvs.rfl, removing, 53
#cvs.rfl, technical details, 9
#cvs.tfl, 9
#cvs.wfl, removing, 53
#cvs.wfl, technical details, 9
&, in modules file, 106
__ files (VMS), 102

A
Abandoning work, 56
Access a branch, 35
Access control lists (ACLs), 21
AclMode, in CVSROOT/config, 125
ActiveScript support, 123
add, 41, 72
Adding a tag, 29
Adding files, 41
Admin, 73
Administrative files (intro), 12
Administrative files (reference), 105
Administrative files, editing them, 13
Alias modules, 105
Alias tags, 32
Alias, Repository, 20
ALL in commitinfo, 113
Ampersand modules, 106
annotate, 47, 74
Atomic transactions, 54
Atomicity, 54
Attic, 9
Authenticated client, using, 18
Authenticating server, setting up, 17
Authentication, stream, 69
Author keyword, 59
authserver, 17
authserver (client/server connection method), port specification, 17
Automatically ignored files, 122
Avoiding editor invocation, 72

B
Backing up, repository, 14
Base directory, in CVS directory, 12
BASE, as reserved tag name, 29
BASE, special tag, 72
Bill of materials, 66

Binary files, 48
Branch keyword, 59
Branch merge example, 37
Branch number, 28, 36
Branch, accessing, 35
Branch, check out, 35
Branch, creating a, 34
Branch, identifying, 35
Branch, retrieving, 35
Branch, vendor-, 63
Branches motivation, 34
Branches, copying changes between, 34
Branches, sticky, 35
Branching, 34
Bringing a file up to date, 51
Bugs in this manual or CVS, 135
Bugs, reporting, 135
Builds, 66

C

chacl, 75
Changes, copying between branches, 34
Changesets, 32
Changing a log message, 73
Changing passwords, 23
Check out a branch, 35
Checked out copy, keeping, 123
Checkin program, 107
Checking commits, 113
Checking out source, 3
checkout, 75
Checkout program, 107
Checkout, as term for getting ready to edit, 56
Checkout, example, 3
checkoutlist, 122
Choosing, reserved or unreserved checkouts, 57
chown, 78
Chroot, running within, 23
Cleaning up, 4
Client/Server Operation, 14
Client/Server Operation, port specification, 14, 17
co, 75
Command structure, 68
commit, 78
Commit files, 110
Commit identifiers, 32
Commit, when to, 58
commitid, 32
CommitId keyword, 59
COMMITID, internal variable, 125
Commitinfo, 113
Committing changes, 3
Common options, 71
Common syntax of info files, 111
Compatibility, between CVS versions, 129
Compression, 70
COMSPEC, environment variable, 128
config, in CVSROOT, 125
Configuration, Server, 126
Conflict markers, 53
Conflict resolution, 53
Conflicts (merge example), 52
Contributors (cvsnt program), 1
Contributors (manual), 134
Copying a repository, 14
Copying changes, 34
Correcting a log message, 73
Creating a branch, 34
Creating a project, 25
Creating a repository, 13
Credits (cvsnt program), 1
Credits (manual), 134
CVROOT/shadow, 123
CVS 1.6, and watches, 57
cvs add, 41
cvs annotate, 47
CVS command structure, 68
CVS directory, in repository, 9
CVS directory, in working directory, 10
cvs edit, 56
cvs editors, 56
CVS passwd file, 17
cvs remove, 42
cvs unedit, 56
cvs watch add, 55
cvs watch off, 54
cvs watch on, 54
cvs watch remove, 55
cvs watch ro, 54
cvs watchers, 56
CVS, introduction to, 1
CVS, versions of, 129
CVS/Base directory, 12
CVS/Entries file, 11
CVS/Entries.Backup file, 11
CVS/Entries.Extra file, 12
CVS/Entries.Extra.Old file, 12
CVS/Entries.Log file, 11
CVS/Entries.Old file, 11
CVS/Entries.Static file, 12
CVS/Notify file, 12
CVS/Notify.tmp file, 12
CVS/Rename file, 12
CVS/Repository file, 10
CVS/Root file, 6
CVS/Tag file, 12
CVS/Template file, 12
CVS_CLIENT_LOG, environment variable, 127
CVS_DIR, environment variable, 128
CVS_EXT, environment variable, 127
CVS_RSH, environment variable, 127
CVS_SERVER_LOB, environment variable, 128
CVS_SERVER_SLEEP, environment variable, 128
cvsadmin, 73

CVSCONF, environment variable, 128
CVSEDITOR, environment variable, 3
CVSEDITOR, internal variable, 125
cvsignore (admin file), global, 122
CVSIGNORE, environment variable, 127
CVSLIB, environment variable, 128
cvslockd, 24
cvsnt, history of, 1
CVSPID, internal variable, 125
cvsrc file, 69
CVSREAD, environment variable, 127
CVSREAD, overriding, 70
cvsroot, 6
CVSROOT (file), 105
CVSROOT, environment variable, 6
CVSROOT, internal variable, 125
CVSROOT, module name, 12
CVSROOT, multiple repositories, 13
CVSROOT, overriding, 69
CVSROOT, storage of files, 10
CVSROOT/admin, 23
CVSROOT/config, 125
CVSROOT/cvsrc file, 69
CVSROOT/keywords, 118
CVSUMASK, environment variable (Unix only), 8
cvswrappers, 109
CVSWRAPPERS, environment variable, 109, 127
cygwin, compatibility issues, 129

D
Date keyword, 59
Dates, 71
Dead state, 9
Decimal revision number, 28
DEFAULT in commitinfo, 113
DEFAULT in verifymsg, 113
Defining a module, 26
Defining modules (intro), 12
Defining modules (reference manual), 105
Deleting files, 42
Deleting revisions, 73
Deleting sticky tags, 33
Deleting tags, 31
Descending directories, 40
Device nodes, 67
Diff, 4
diff, 80
Differences, merging, 38
Directories, moving, 45
Directories, removing, 43
Directory, descending, 40
Disjoint repositories, 13
Distributing log messages, 114
driver.c (merge example), 51

E
edit, 56, 82
Editing administrative files, 13
Editing the modules file, 26
Editor, avoiding invocation of, 72
EDITOR, environment variable, 3
EDITOR, internal variable, 125
EDITOR, overriding, 69
editors, 56, 83
Email notification, 119
emerge, 53
Encryption, 70
Entries file, in CVS directory, 11
Entries.Backup file, in CVS directory, 11
Entries.Extra file, in CVS directory, 12
Entries.Extra.Old file, in CVS directory, 12
Entries.Log file, in CVS directory, 11
Entries.Old file, in CVS directory, 11
Entries.Static file, in CVS directory, 12
Environment variables, 127
environment variables, passed to administrative files, 125
Errors, reporting, 135
Example of a work-session, 3
Example of merge, 51
Example, branch merge, 37
Excluding directories, in modules file, 107
Exit status, of commitinfo, 113
Exit status, of CVS, 68
Exit status, of editor, 133
Exit status, of taginfo, 46
Exit status, of verifymsg, 113
export, 83
Export program, 107
extnt.exe, 16
extnt.ini, 16

F
Fetching source, 3
File had conflicts on merge, 50
File locking, 50
File permissions, general, 8
File status, 50
fileattr.xml, in repository, 9
Files, moving, 43
Files, reference manual, 105
Filesystem locks (obsolete), 9
Fixing a log message, 73
Forcing a tag match, 71
fork, access method, 19
Form for log message, 117
Format of CVS commands, 68

G
Getting started, 3
Getting the source, 3
Global cvsignore, 122
Global options, 69
Group, 8

gserver (client/server connection method), port specification, 14
GSSAPI, 19
Gzip, 70

H
Hard links, 67
HEAD, as reserved tag name, 29
HEAD, special tag, 72
Header keyword, 59
history, 84
History browsing, 46
History file, 123
History files, 8
History of cvsnt, 1
historyinfo, 117
HOME, environment variable, 127
HOMEDRIVE, environment variable, 127
HOMEPATH, environment variable, 127

I
Id keyword, 59
Ident (shell command), 60
Identifying a branch, 35
Identifying files, 59
Ignored files, 122
Ignoring files, 122
import, 86
Importing files, 25
Importing files, from other version control systems, 26
Importing modules, 63
info, 88
Info files (syntax), 111
Informing others, 53
init, 13, 87
Internal variables, 124
Introduction to CVS, 1
Isolation, 46

J
Join, 37

K
Keeping a checked out copy, 123
Kerberos, using :gserver:, 19
Kerberos, using kerberized rsh, 15
Keyword expansion, 59
Keyword List, 59
Keyword substitution, 59
Keyword substitution, and merging, 39
Keyword substitution, changing modes, 60
keywords, 118
Kflag, 60

L
Layout of repository, 6
Left-hand options, 69
Library interface, 112
Linear development, 28
Link, symbolic, importing, 87
List, mailing list, 2
Locally Added, 50
Locally Modified, 50
Locally Removed, 50
LockDir, in CVSROOT/config, 126
Locker keyword, 59
Locking files, 50
Locks, cvs, and backups, 14
Locks, cvs, introduction, 53
Locks, cvs, technical details, 9
Lockserver, 24
LockServer, in CVSROOT/config, 126
Lockserver, setting up, 24
log, 89
Log information, saving, 123
Log keyword, 59
Log message entry, 3
Log message template, 117
Log message, correcting, 73
Log message, verifying, 113
Log messages, 114
LogHistory, in CVSROOT/config, 126
Login, 18
login, 91
loginfo, 114
Logout, 18
logout, 91
ls, 91
lsacl, 92

M
Mail, automatic mail on commit, 53
Mailing list, 2
Mailing log messages, 114
Main trunk and branches, 34
make, 66
Many repositories, 13
Markers, conflict, 53
Merge, an example, 51
Merge, branch example, 37
Merging, 34
Merging a branch, 37
Merging a file, 51
Merging two revisions, 38
Merging, and keyword substitution, 39
mkmodules, 131
Modifications, copying between branches, 34
Module status, 107
Module, defining, 26
Modules (admin file), 105
Modules file, 12
Modules file program options, 108
Modules file, changing, 26
modules.db, 10
modules.dir, 10

modules.pag, 10
modules2 (admin file), 108
modules2, compared to modules, 108
modules2, syntax, 108
Motivation for branches, 34
Moving a repository, 14
Moving directories, 45
Moving files, 43
Moving tags, 31
Multiple developers, 50
Multiple repositories, 13

N
Name keyword, 59
Name, symbolic (tag), 29
Needs Checkout, 50
Needs Merge, 50
Needs Patch, 50
Newsgroups, 2
notify, 118
notify (admin file), 55
Notify file, in CVS directory, 12
Notify.tmp file, in CVS directory, 12
Number, branch, 28, 36
Number, revision-, 28

O
Option defaults, 69
Options, global, 69
Options, in modules file, 107
Outdating revisions, 73
Overlap, 51
Overriding CVSREAD, 70
Overriding CVSROOT, 69
Overriding EDITOR, 69
Overriding rcsBIN, 69
Overriding TMPDIR, 69
Overview, 1
Ownership, saving in CVS, 67

P
Parallel repositories, 13
passwd, 93
passwd (admin file), 17
PATH, environment variable, 127
Per-directory sticky tags/dates, 12
Permissions, general, 8
Permissions, saving in CVS, 67
Policy, 58
port, specifying for remote repositories, 14, 17
postcommand, 116
Postcommand actions, 116
postcommit, 117
postmodule, 117
Postmodule actions, 117
precommand, 116
Precommand actions, 116
Precommit checking, 113
Prefix, Repository, 20
premodule, 116
Premodule actions, 116
pserver (client/server connection method), port specification, 14
PVCS, importing files from, 26

R
rannotate, 93
rchacl, 94
rchown, 94
rcs history files, 8
RCS keywords, redefining, 118
rcs revision numbers, 29
rcs, importing files from, 26
rcs-style locking, 50
rcsBIN, overriding, 69
rcsfile keyword, 60
RCSHeader keyword, 59
rcsinfo, 117
rdiff, 94
Read-only files, and -r, 70
Read-only files, and CVSREAD, 127
Read-only files, and watches, 54
Read-only files, in repository, 8
Read-only mode, 70
Read-only repository access, 23
readers (admin file), 23
REAL_CVSROOT, internal variable, 125
Recursive (directory descending), 40
Reference manual (files), 105
Reference manual for variables, 127
Regular expression syntax, 111
Regular modules, 106
release, 95
Releases, revisions and versions, 28
Releasing your working copy, 4
Remote repositories, 14
Remote repositories, port specification, 14, 17
Remove, 42
remove, 96
Removing a change, 38
Removing directories, 43
Removing files, 42
Removing tags, 31
Removing your working copy, 4
rename, 97
Rename file, in CVS directory, 12
Renaming directories, 45
Renaming files, 43
Renaming tags, 31
Replacing a log message, 73
Reporting bugs, 135
Repositories, multiple, 13
Repositories, remote, 14
Repositories, remote, port specification, 14, 17

Repository (intro), 6
Repository administrators, 23
Repository Alias, 20
Repository file, in CVS directory, 10
Repository Prefix, 20
Repository, backing up, 14
Repository, example, 6
Repository, how data is stored, 7
Repository, moving, 14
Repository, setting up, 13
RereadLogAfterVerify, in CVSROOT/config, 126
Reserved checkouts, 50
Resetting sticky tags, 33
Resolving a conflict, 53
Restoring old version of removed file, 38
Resurrecting old version of dead file, 42
Retrieve a branch, 35
Retrieving an old revision using tags, 30
Reverting to repository version, 56
Revision keyword, 60
Revision management, 58
Revision numbers, 28
Revision numbers (branches), 36
Revision tree, 28
Revision tree, making branches, 34
Revisions, merging differences between, 38
Revisions, versions and releases, 28
Right-hand options, 71
rlog, 97
rls, 91
rlsacl, 93
Root file, in CVS directory, 6
rtag, 31, 97
rtag, creating a branch using, 34

S

Saving space, 73
SCCS, importing files from, 26
script.js
    javascript, 123
script.pl
    perlscript, 123
script.rb
    rubyscript, 123
script.vbs
    vbscript, 123
Security (intro), 21
Security, example, 21
Security, file permissions in repository, 8
Security, GSSAPI, 19
Security, of pserver, 19
Security, of sserver, 19
Security, of sspi, 19
Security, running as a nonprivileged user, 23
Security, running within a chroot jail, 23
Security, setuid (Unix only), 8
Secutiry, of ntserver, 19
Server configuration, 126
Server, CVS, 14
Server, temporary directories, 24
sessionid, 32
SESSIONID, internal variable, 125
Setgid (Unix only), 8
Setting permissions for files and directories, 21
Setting up a repository, 13
Setting up the Lockserver, 24
Setuid (Unix only), 8
Shadow file, 123
Source keyword, 60
Source, getting cvsnt source, 1
Source, getting from CVS, 3
Special files, 67
Specifying dates, 71
Spreading information, 53
sserver (client/server connection method), port specification, 14
ssh, 15
ssh replacements (Kerberized RSH, &c), 15
sspi (client/server connection method), port specification, 14
Starting a project with CVS, 25
State keyword, 60
status, 98
Status of a file, 50
Status of a module, 107
Sticky date, 33
Sticky tags, 32
Sticky tags, resetting, 33
Sticky tags/dates, per-directory, 12
Storing log messages, 114
Stream authentication, 69
Structure, 68
Subdirectories, 40
Symbolic link, importing, 87
Symbolic links, 67
Symbolic name (tag), 29
Syntax of info files, 111
SystemAuth, in CVSROOT/config, 125

T

tag, 30, 98
Tag file, in CVS directory, 12
Tag program, 107
tag, command, introduction, 29
tag, creating a branch using, 34
Tag, example, 29
Tag, retrieving old revisions, 30
Tag, symbolic name, 29
taginfo, 46
Tags, 29
Tags, alias, 32
Tags, deleting, 31
Tags, moving, 31
Tags, renaming, 31

Tags, sticky, 32
tc, Trivial Compiler (example), 3
Team of developers, 50
TEMP, environment variable, 128
Template file, in CVS directory, 12
Template for log message, 117
Temporary directories, and server, 24
Temporary files, location of, 128
Third-party sources, 63
Time, 71
Timezone, in input, 71
Timezone, in output, 89
TMP, environment variable, 128
TMPDIR, environment variable, 128
TMPDIR, overriding, 69
TopLevelAdmin, in CVSROOT/config, 125
Trace, 70
Traceability, 46
Tracking sources, 63
triggers, 112
Trivial Compiler (example), 3
Typical repository, 6

U
Umask, for repository files (Unix only), 8
Undoing a change, 38
unedit, 56, 99
Unknown, 50
Unreserved checkouts, 50
Up-to-date, 50
update, 100
Update, introduction, 51
update, to display file status, 51
Updating a file, 51
User aliases, 17
User variables, 125
USER, internal variable, 125
users (admin file), 55
Users, adding and removing, 21
Using CVSNT protocols with 3rd party clients, 16

V
Variables, 124
Vendor, 63
Vendor branch, 63
verifymsg, 113
version, 102
Versions, of CVS, 129
Versions, revisions and releases, 28
Viewing differences, 4
VIRTUAL_CVSROOT, internal variable, 125
VISUAL, internal variable, 125

W
watch, 103
watch add, 55
watch off, 54
watch on, 54
watch remove, 55
watch rw, 54
Watcher, in CVSROOT/config, 126
watchers, 56, 103
Watches, 54
wdiff (import example), 63
Web pages, maintaining with CVS, 123
What (shell command), 60
What branches are good for, 34
What is CVS not?, 2
What is CVS?, 1
When to commit, 58
Work-session, example of, 3
Working copy, 50
Working copy, removing, 4
Wrappers, 109
writers (admin file), 23

X
xdiff, 103

Z
Zone, time, in input, 71
Zone, time, in output, 89