Version Management
with
CVS

release 0.5, for CVS 1.3-s2/1.3.1.

Per Cederqvist

last updated 15 Mar 1993

About this manual

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

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

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

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


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

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


Credits

Roland Pesch, Cygnus Support <pesch@cygnus.com> wrote the manual pages that was distributed with CVS 1.3. Appendix A and B contains much text that was extracted from them.

The following persons read different early drafts of this manual and told me about many of the mistakes I had made in it. Any errors that exists in the manual now have probably been added later.

- Roland Pesch, Cygnus Support <pesch@cygnus.com>
- Roxanne Brunskill, Sandwell Inc. Datap Systems Division <rbrunski@datap.ca>
- Inge Wallin, Signum Support <ingwa@signum.se>

Roland Pesch deserves an extra thank for his most useful comments.

BUGS

This manual is still very new. Here is a list of known deficiencies in it:

- No index exists yet.
- The 'admin' command is not documented here.
- In the examples, the output from CVS is sometimes displayed, sometimes not displayed.
- The input that you are supposed to type in the examples should have a different font than the output from the computer.
- This manual should be clearer about what file permissions you should set up in the repository, and about setuid/setgid.
- A lot of chapters are only half-written. They are noted by comments in the 'cvs.texinfo' file, and sometimes by 'FIXME' or '///' in the written text. (If you find a too-short chapter without any of those markers, please send me mail!)
- This list is not complete. If you notice any error, omission, or something that is unclear, please send mail to ceder@signum.se.

I hope that you will find this manual useful, despite the above-mentioned shortcomings. Also, you alpha readers, mail all and any comments you have on this document to me, ceder@signum.se, as soon as possible. The earlier I get your mail, the greater the chance that I have time to fix whatever you say before the next release of CVS (and that might come sooner than you think!)

Linkoping, February 1993
Per Cederqvist

1 What is CVS?

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

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

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

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

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

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

2 Basic concepts

CVS stores all files in a centralized *repository*: a directory (such as '/usr/local/cvsroot') which is populated with a hierarchy of files and directories.

Normally, you never access any of the files in the repository directly. Instead, you use CVS commands to get your own copy of the files, and then work on that copy.

The files in the repository are organized in *modules*. Each module is made up of one or more files, and can include files from several directories.

2.1 Revision numbers

Each version of a file is given 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 numbers. By default revision 1.1 is the first revision of a file. Each successive revision is given a new number by increasing the rightmost number by one. The following figure displays a few revisions, with newer revisions to the right.

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

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

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

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

```
                                         +-------------+
                  Branch 1.2.2.3.2 ->        ! 1.2.2.3.2.1 !
```

```
                                                   / +-------------+
                                                  /
                                                 /
                 +---------+    +---------+    +---------+    +---------+
Branch 1.2.2 -> _! 1.2.2.1 !----! 1.2.2.2 !----! 1.2.2.3 !----! 1.2.2.3 !
               / +---------+    +---------+    +---------+    +---------+
              /
             /
+-----+    +-----+    +-----+    +-----+    +-----+
! 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 !
                    +---------+    +---------+    +---------+
```

CVS only assigns even numbers to branch numbers that you create. The branch 1.1.1 has a special meaning. See ⟨undefined⟩ [Tracking sources], page ⟨undefined⟩.

2.2 Versions, revisions and releases

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

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

3 A sample session

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

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

3.1 Getting the source

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

```
$ cvs checkout tc
```

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

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

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

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

3.2 Committing your changes

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

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

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

The environment variable `$EDITOR` determines which editor is started. If `$EDITOR` is not set, the editor is `vi`. If you want to avoid the overhead of starting an editor you can specify the log message on the command line using the '`-m`' flag instead, like this:

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

3.3 Cleaning up

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

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

but a better way is to use the `release` command (see ⟨undefined⟩ [release], page ⟨undefined⟩):

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

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

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

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

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

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

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

3.4 Viewing differences

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

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

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

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

4 The Repository

Figure 3 below shows a typical setup of a repository. Only directories are shown below.

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

The $CVSROOT environment variable should always be an absolute path to the root of the repository, '/usr/local/cvsroot' in this example. With this setup all csh and tcsh users should have this line in their '.cshrc' or '.tcshrc' files:

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

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

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

There is nothing magical about the name '/usr/local/cvsroot'. You can choose to place the repository anywhere you like, but $CVSROOT must always point to it.

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

4.1 User modules

```
$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 figure above shows the contents of the 'tc' module inside the repository. As you can see all file names ends in ',v'. The files are *history files*. They 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. CVS uses the facilities of RCS, a simpler version control system, to maintain these files. For a full description of the file format, see the man page *rcsfile(5)*.

4.1.1 File permissions

All ',v' files are created read-only, and you should not change that. The directories should be writable by the persons that have permission to modify the files. This normally means that you must create a group consisting of the persons that are to edit a project, and set up the repository so that it is that group that owns the directory. CVS tries to set up reasonable file permissions for new directories that are added inside the tree, but you must fix the permissions manually when a new directory should have different permissions than its' parent directory.

Since CVS was not written to be run setuid, it is unsafe to try to run it setuid.

4.2 The administrative files

The directory '$CVSROOT/CVSROOT' contains some *administrative files*. See ⟨undefined⟩ [Administrative files], page ⟨undefined⟩, for a complete description. You can use CVS without any of these files, but some commands works 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         -i mkmodules CVSROOT
modules -i mkmodules 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.

Each module definition can contain options. The '-i mkmodules' is an example of an option. It arranges for CVS to run the mkmodules program whenever any file in the module CVSROOT is committed. That program is responsible for checking out read-only copies from the RCS *history files* of all the administrative files. These read-only copies are used by CVS. You should never edit them directly.

The line that defines the module called 'modules' uses features that are not explained here. See ⟨undefined⟩ [modules], page ⟨undefined⟩, for a full explanation of all the available features.

4.2.1 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. See ⟨undefined⟩ [Bad administrative files], page ⟨undefined⟩ for a hint about how to solve such situations.

5 Starting a project with CVS

Since CVS 1.3 is bad at renaming files and moving them between directories, the first thing you do when you start a new project should be to think through your file organization. It is not impossible—just awkward—to rename or move files in CVS 1.3. See ⟨undefined⟩ [Moving files], page ⟨undefined⟩.

What to do next depends on the situation at hand.

Alpha note: This chapter is not finished yet. I will rewrite it before the final release of this manual. Comments are welcome!

5.1 Setting up the files

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

///The contributed scripts///

5.1.1 Creating a module from a number of files

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

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

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

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

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

Erasing the original source is a good idea, to make sure that you do not accidentally edit them in *dir*, bypassing CVS.

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.

5.1.2 Creating a module from scratch

///

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

After that, you use the `import` command to create a module

5.2 Defining the module

The next step is to define the module in the '`modules`' file. Some CVS commands work without this step, but others (most notably `release`) require that all modules are properly defined in the '`modules`' file.

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

1. Get a working copy of the modules file.

   ```
   $ cvs checkout modules
   $ cd modules
   ```

2. Edit the file and insert a line that defines the module. See ⟨undefined⟩ [Intro administrative files], page ⟨undefined⟩, for an introduction. See ⟨undefined⟩ [modules], page ⟨undefined⟩, 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 modules
```

6 Multiple developers

When more than one person works on a software project things often get complicated. Often, two people try to edit the same file simultaneously. Some other version control systems (including RCS and SCCS) tries to solve that particular problem by introducing *file locking*, so that only one person can edit each file at a time. Unfortunately, file locking can be very counter-productive. If different people want to edit different parts of a file, there may be no reason to prevent either of them from doing that.

CVS does not use file locking. Instead, it allows many people to edit their own *working copy* of a file simultaneously. The first person that commits his changes has no automatic way of knowing that another has started to edit it. Others will get an error message when they try to commit the file. They must then use CVS commands to bring their working copy up to date with the repository revision. This process is almost automatic, and explained in this chapter.

There are many ways to organize a team of developers. CVS does not try to enforce a certain organization. It is a tool that can be used in several ways. It is often useful to inform the group of commits you have done. CVS has several ways of automating that process. See ⟨undefined⟩ [Informing others], page ⟨undefined⟩. See ⟨undefined⟩ [Revision management], page ⟨undefined⟩, for more tips on how to use CVS.

6.1 File status

After you have checked out a file out from CVS, it is in one of these four states:

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

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

Needing update
: Someone else has committed a newer revision to the repository.

Needing merge
: Someone else have committed a newer revision to the repository, and you have also made modifications to the file.

You can use the `status` command to find out the status of a given file. See ⟨undefined⟩ [status], page ⟨undefined⟩.

6.2 Bringing a file up to date

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

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

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

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

6.3 Conflicts example

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

```
#include <stdio.h>

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

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

```
#include <stdio.h>

int main(int argc,
```

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

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

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

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

You run 'cvs update':

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

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

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

6.4 Informing others about commits

It is often useful to inform others when you commit a new revision of a file. The '-i' option of the 'modules' file, or the 'loginfo' file, can be used to automate this process. See ⟨undefined⟩ [modules], page ⟨undefined⟩. See ⟨undefined⟩ [loginfo], page ⟨undefined⟩. You can instruct CVS to mail a message to all developers, or post a message to a local newsgroup.

7 Branches

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

7.1 Tags–Symbolic revisions

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

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

You can use the `tag` command to give a symbolic name to a certain revision of a file. You can use the '`-v`' flag to the `status` command to see all tags that a file has, and which revision numbers they stand for. (The output of `status` unfortunately uses the word "version" instead of "revision".)

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

    Version:            1.4     Tue Dec  1 14:39:01 1992
    RCS Version:        1.4     /usr/local/cvsroot/yoyodyne/tc/backend.c,v
    Sticky Tag:         (none)
    Sticky Date:        (none)
```

```
Sticky Options:     (none)

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

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

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

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

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

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

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

You can also check out a module as it was at any given date. See ⟨undefined⟩ [checkout options], page ⟨undefined⟩.

7.2 What branches are good for

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

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

7.3 Creating a branch

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

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

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

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

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

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

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

===================================================================
```

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

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

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

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

7.4 Sticky tags

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

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

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

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

```
        release-1-0                 (revision: 1.7)
```

The sticky tags will remain on your working files until you delete them with 'cvs update -A'. See ⟨undefined⟩ [update], page ⟨undefined⟩.

Sticky tags are not just for branches. If you check out a certain revision (such as 1.4) it will also become sticky. Subsequent 'cvs update' will not retrieve the latest revision until you reset the tag with 'cvs update -A'.

See the descriptions in Appendix A for more information about sticky tags. Dates and some other options can also be sticky. Again, see Appendix A for details.

8 Merging

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

8.1 Merging an entire branch

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

The '`-j`' stands for "join". In previous versions of CVS there was a special command, '`cvs join`', that was used to merge changes between branches.

Consider this revision tree:

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

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, 1.5

$ cvs update -j R1fix m.c        # Merge all changes made on the branch,
                                 # i.e. the changes between revision 1.2
                                 # and 1.2.2.3, into the main trunk.

$ cvs commit -m "Included R1fix" # Create revision 1.6.
```

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

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

8.2 Merging differences between any two revisions

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

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

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

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

9 Recursive behaviour

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

```
$HOME
  |
  +--‘tc’
  |   |
      +--‘CVS’
      |      (internal CVS files)
      +--‘Makefile’
      +--‘backend.c’
      +--‘driver.c’
      +--‘frontend.c’
      +--‘parser.c’
      +--‘man’
      |    |
      |    +--‘CVS’
      |    |  (internal CVS files)
      |    +--‘tc.1’
      |
      +--‘testing’
           |
           +--‘CVS’
           |  (internal CVS files)
           +--‘testpgm.t’
           +--‘test2.t’
```

If ‘tc’ is the current working directory, the following is true:

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

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

The recursive behaviour of the CVS subcommands can be turned off with the ‘-l’ option.

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

10 Adding files to a module

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

- You must have a working copy of the module. See ⟨undefined⟩ [Getting the source], page ⟨undefined⟩.
- Create the new file inside your working copy of the module.
- Use '`cvs add` *filename*' to tell CVS that you want it to handle the file.
- 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 inside a module.

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
```

See ⟨undefined⟩ [add], page ⟨undefined⟩, for a more complete description of the `add` command.

11 Removing files from a module

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

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

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

What happens when you commit the removal of the file is that inside the source repository, it is moved into a subdirectory called '`Attic`'. CVS normally doesn't look in that directory when you run e.g. `checkout`. However, if you are retrieving a certain revision via e.g. '`cvs checkout -r` *some-tag*', it will look at the files inside the '`Attic`' and include any files that contain the specified tag.

This method is simple and works quite well, but it has some known deficiencies:

- If you remove the file '`foo.c`', you cannot later create a new file called '`foo.c`' unless you manually remove the file '`Attic/foo.c,v`' inside the repository.
- If the file '`bar.c`' is present in release 1.0 of a product, and was accidentally removed in release 1.1, you cannot easily resurrect it to release 1.2. You have to move the file out of the '`Attic`' manually inside the repository. (Do a '`mv Attic/`*file file*').

There is a design for a *rename database* that will solve these problems and many others, but it is not yet implemented.

12 Tracking third-party sources

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

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

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

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

12.1 Importing a module for the first time

Use the `import` command to check in the sources for the first time. See ⟨undefined⟩ [From files], page ⟨undefined⟩, for an example. 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 1.1.1, unless you use the '`-b` *branch*' flag—See ⟨undefined⟩ [import options], page ⟨undefined⟩). The *release tags* are symbolic names for a particular release, such as '`SunOS_4_1`'.

12.2 Updating a module with the import command

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

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 -jtag:yesterday -jtag module
```

The above command will check out the latest revision of *module*, merging the changes made on the vendor branch *tag* since yesterday into the working copy. If any conflicts arise during the merge they should be resolved in the normal way (see ⟨undefined⟩ [Conflicts example], page ⟨undefined⟩). Then, the modified files may be committed.

12.3 Tracking sources—a success story

This is a story from Brian Berliner, the main author of CVS, who describes how an early version of CVS helped him. Since this text was written, the `join` command has been replaced by the '`-j`' flag in `checkout` and `update`, and there are no hard-coded limits when CVS is used together with GNU `diff`.

> The true test of the CVS vendor branch support came with the arrival of the SunOS 4.0.3 source upgrade tape. As described above, the checkin program was used to install the new sources and the resulting output file listed the files that had been locally modified, needing to be merged manually. For the kernel, there were 94 files in conflict. The CVS `join` command was used on each of the 94 conflicting files, and the remaining conflicts were resolved.
>
> The `join` command performs an rcsmerge operation. This in turn uses `diff3` to produce a three-way diff file. As it happens, the `diff3` program has a hard-coded limit of 200 source-file changes maximum. This proved to be too small for a few of the kernel files that needed merging by hand, due to the large number of local changes that Prisma had made. The `diff3` problem was solved by increasing the hard-coded limit by an order of magnitude.
>
> The SunOS 4.0.3 kernel source upgrade distribution contained 346 files, 233 of which were modifications to previously released files, and 113 of which were newly added files. checkin added the 113 new files to the source repository without intervention. Of the 233 modified files, 139 dropped in cleanly by checkin, since Prisma had not made any local changes to them, and 94 required manual merging due to local modifications. The 233 modified files consisted of 20,766 lines of differences. It took one developer two days to manually merge the 94 files using the `join` command and resolving conflicts manually. An additional day was required for kernel debugging. The entire process of merging over 20,000 lines of differences was completed in less than a week. This one time-savings alone was justification enough for the CVS development effort; we expect to gain even more when tracking future SunOS releases.

13 Moving and renaming files

One of the biggest design flaw with the current release of CVS is that it is very difficult to move a file to a different directory or rename it. There are workarounds, and they all have their strong and weak points. (Moving or renaming a directory is even harder. See ⟨undefined⟩ [Moving directories], page ⟨undefined⟩).

The examples below assume that the file *old* is renamed to *new*. Both files reside in the same module, *module*, but not necessarily in the same directory. The relative path to the module inside the repository is assumed to be *module*.

13.1 Moving outside the repository

One way to move the file is to copy *old* to *new*, and then issue the normal CVS commands to remove *old* from the repository, and add *new* to it. (Both *old* and *new* could contain relative paths inside the module).

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

Advantages:

- Checking out old revisions works correctly.

Disadvantages:

- You cannot easily see the history of the file across the rename.
- Unless you use the '`-r rev`' flag when *new* is committed its revision numbers will start at 1.0 again.

13.2 Move the history file

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

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

Advantages:

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

Disadvantages:

- Old releases of the module cannot easily be fetched from the repository. (The file will show up as *new* even in revisions from the time before it was renamed).
- There is no log information of when the file was renamed.
- Nasty things might happen if someone accesses the history file while you are moving it. Make sure no one else runs any of the CVS commands while you manually perform these steps.

14 Moving and renaming directories

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

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

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

   ```
   $ cd $CVSROOT/module
   $ mv old-dir new-dir
   ```
3. Fix the CVS administrative files, if necessary (for instance if you renamed an entire module).
4. Tell everyone that they can check out the module and continue working.

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

15 History browsing

Alpha note: This chapter is not yet finished.

CVS tries to make it easy for a group of people to work together. This is done in two ways:

- Isolation—You have your own working copy of the source. You are not affected by modifications made by others until you decide to incorporate those changes (via the `update` command—see ⟨undefined⟩ [update], page ⟨undefined⟩).
- Traceability—When something has changed, you can always see *exactly* what changed.

There are several features of CVS that together lead to traceability.

- Each revision of a file has an accompanying log message.
- All commits are optionally logged to a central history database.
- Logging information can be sent to a user-defined program (see ⟨undefined⟩ [loginfo], page ⟨undefined⟩).

///FIXME

This chapter should talk about the history file, the `log` command, the usefulness of ChangeLogs even when you run CVS, and things like that.

15.1 Log messages

Whenever you commit a file you specify a log message. ///

15.2 The history database

///

15.3 User-defined logging

///

16 Keyword expansion

Alpha note: This chapter is going to talk about keyword expansion of things such as `$Id:$`, but I haven't had time to write it down yet.

17 Revision management

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

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

17.1 When to commit?

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

If you commit files too quickly you might commit files that doesn't even compile. If your partner updates his working sources to include your buggy file, he will be unable to compile the code.

The following list is intended to give you something to think about.

Compileable
: Only files that are compileable are committed.

Test suit
: Only files that have passed a test suit are committed.

Tests like the ones above can be enforced using the commitinfo file (see ⟨undefined⟩ [commitinfo], page ⟨undefined⟩), but you should think twice before you enforce such a convention. By making the development environment too controlled it might be boring and thus counter-productive.

Appendix A Reference manual for CVS commands

This appendix describes every subcommand of CVS in detail.

A.1 Overall structure of CVS commands

The first release of CVS consisted of a number of shell-scripts. Today CVS is implemented as a single program that is a front-end to RCS and `diff`.

The overall format of all CVS commands is

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

`cvs` The program that is a front-end to RCS.

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

`cvs_command`
: One of 16 different sub-commands. Some of the commands have aliases that can be used instead; those aliases are noted in the reference manual for that command. There are only two situations where you may omit '`cvs_command`': '`cvs -H`' elicits a list of available commands, and '`cvs -v`' displays version information on CVS itself.

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

`command_args`
: Arguments to the commands.

The available '`cvs_options`' are:

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

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

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

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

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

-l
: Do not log the cvs_command in the command history (but execute it anyway). See ⟨undefined⟩ [history], page ⟨undefined⟩, for information on command history.

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

-t
: Trace program execution; display messages showing the steps of CVS activity. Particularly useful with '`-n`' to explore the potential impact of an unfamiliar command.

-r
: Makes new working files files read-only. Same effect as if the `$CVSREAD` environment variable is set (see ⟨undefined⟩ [Environment variables], page ⟨undefined⟩). The default is to make working files writable.

-v
: Displays version and copyright information for CVS.

-w
: Makes new working files read-write. Overrides the setting of the `$CVSREAD` environment variable. This option is on by default.

A.2 Common command options

This section describes the '`command_options`' that are available across several CVS commands. Not all commands support all of these options; each option is only supported for commands where it makes sense. However, when a command has one of these options you can almost always count on the same meaning for the option as in other commands. (Other command options, which are listed with the individual commands, may have different meanings from one CVS command to another).

Warning: the '`history`' command is an exception; it supports many options that conflict even with these standard options.

-D *date_spec*
: Use the most recent revision no later than *date_spec*. *date_spec* is a single argument, a date description specifying a date in the past.

The specification is *sticky* when you use it to make a private copy of a source file; that is, when you get a working file using '-D', CVS records the date you specified, so that further updates in the same directory will use the same date (unless you explicitly override it; see ⟨undefined⟩ [update], page ⟨undefined⟩).

A wide variety of date formats are supported by the underlying RCS facilities, similar to those described in co(1), but not exactly the same. The *date_spec* is interpreted as being in the local timezone, unless a specific timezone is specified. Examples of valid date specifications include:

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

'-D' is available with the `checkout`, `diff`, `history`, `rdiff`, `rtag`, and `update` commands. Remember to quote argument to the '-D' flag so that your shell doesn't interpret the spaces as argument separators. A command using the '-D' flag can look like this:

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

`-f` When you specify a particular date or tag to CVS commands, they normally ignore files that do not contain the tag (or did not exist on the date) that you specified. Use the '-f' option if you want files retrieved even when there is no match for the tag or date. (The most recent version is used in this situation).

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

Warning: The `commit` command also has a '-f' option, but it has a different meaning in that command. See ⟨undefined⟩ [commit options], page ⟨undefined⟩.

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

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

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

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

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

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

`-n` Do not run any checkout/commit/tag program. (A program can be specified to run on each of these activities, in the modules database (see ⟨undefined⟩ [modules], page ⟨undefined⟩); 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.

`-m` *message*

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

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

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

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

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

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

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

The tag specification is sticky when you use this option with `checkout` or `update` to make your own copy of a file: CVS remembers the tag and continues to use it on future update commands, until you specify otherwise. The tag can be either a symbolic or numeric tag. See ⟨undefined⟩ [Tags], page ⟨undefined⟩.

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

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

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

A.3 add—Add a new file/directory to the repository

- Synopsis: add [-k kflag] [-m 'message'] files...
- Requires: repository, working directory.
- Changes: working directory.
- Synonym: new

Use the add command to create a new file or directory in the source repository. The files or directories specified with add must already exist in the current directory (which must have been created with the checkout command). To add a whole new directory hierarchy to the source repository (for example, files received from a third-party vendor), use the import command instead. See ⟨undefined⟩ [import], page ⟨undefined⟩.

If the argument to add refers to an immediate sub-directory, the directory is created at the correct place in the source repository, and the necessary CVS administration files are created in your working directory. If the directory already exists in the source repository, add still creates the administration files in your version of the directory. This allows you to use add to add a particular directory to your private sources even if someone else created that directory after your checkout of the sources. You can do the following:

```
$ mkdir new_directory
$ cvs add new_directory
$ cvs update new_directory
```

An alternate approach using update might be:

```
$ cvs update -d new_directory
```

(To add any available new directories to your working directory, it's probably simpler to use checkout (see ⟨undefined⟩ [checkout], page ⟨undefined⟩) or 'update -d' (see ⟨undefined⟩ [update], page ⟨undefined⟩)).

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 resurrect the file, unless a `commit` command intervened. See ⟨undefined⟩ [remove examples], page ⟨undefined⟩ for an example.

Unlike most other commands `add` never recurses down directories. It cannot yet handle relative paths. Instead of

```
$ cvs add foo/bar.c
```

you have to do

```
$ cd foo
$ cvs add bar.c
```

A.3.1 add options

`-k` *kflag* This option specifies the default way that this file will be checked out. See `rcs(1)` and co(1). The *kflag* argument is stored in the RCS file and can be changed with `admin`. Specifying '`-ko`' is useful for checking in binaries that shouldn't have the RCS id strings expanded.

Warning: this option is reported to be broken in version 1.3 and 1.3-s2 of CVS. Use '`admin -k`' after the commit instead. See ⟨undefined⟩ [admin], page ⟨undefined⟩.

`-m` *description*
Using this option, you can give a description for the file. This description appears in the history log (if it is enabled, see ⟨undefined⟩ [history file], page ⟨undefined⟩). It will also be saved in the RCS history file inside the repository when the file is committed. The `log` command displays this description.

The description can be changed using '`admin -t`'. See ⟨undefined⟩ [admin], page ⟨undefined⟩.

If you omit the '`-m` *description*' flag, an empty string will be used. You will not be prompted for a description.

A.3.2 add examples

To add the file '`backend.c`' to the repository, with a description, the following can be used.

```
$ cvs add -m "Optimizer and code generation passes." backend.c
$ cvs commit -m "Early version. Not yet compilable." backend.c
```

A.4 admin—Administration front end for rcs

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

This is the CVS interface to assorted administrative RCS facilities, documented in rcs(1). `admin` simply passes all its options and arguments to the `rcs` command; it does no filtering or other processing. This command *does* work recursively, however, so extreme care should be used.

Since this command is seldom used, it is not documented here. Complete documentation will appear here in the future.

A.4.1 admin options

See rcs(1).

A.4.2 admin examples

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

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

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

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

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

A.5 checkout—Checkout sources for editing

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

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

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

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`' option).

Running `checkout` on a directory that was already built by a prior `checkout` is also permitted, and has the same effect as specifying the '`-d`' option to the `update` command. See ⟨undefined⟩ [update], page ⟨undefined⟩.

A.5.1 checkout options

These standard options are supported by `checkout` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

`-D` *date* Use the most recent revision no later than *date*. This option is sticky, and implies '`-P`'.

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

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

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

`-n` Do not run any checkout program (as specified with the '`-o`' option in the modules file; see ⟨undefined⟩ [modules], page ⟨undefined⟩).

`-P` Prune empty directories.

`-p` Pipe files to the standard output.

`-Q` Really quiet.

`-q` Somewhat quiet.

`-r` *tag* Use revision *tag*. This option is sticky, and implies '`-P`'.

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

`-A` Reset any sticky tags, dates, or '`-k`' options. (If you get a working file using one of the '`-r`', '`-D`', or '`-k`' options, CVS remembers the corresponding tag, date, or *kflag* and continues using it for future updates; use the '`-A`' option to make CVS forget these specifications, and retrieve the 'head' revision of the file).

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

`-d` *dir* Create a directory called *dir* for the working files, instead of using the module name. Unless you also use '`-N`', the paths created under *dir* will be as short as possible.

`-j` *tag* Merge the changes made between the resulting revision and the revision that it is based on (e.g., if *tag* refers to a branch, CVS will merge all changes made on that branch into your working file).

With two '`-j` *tag*' options, CVS will merge in the changes between the two respective revisions. This can be used to undo changes made between two revisions (see ⟨undefined⟩ [Merging two revisions], page ⟨undefined⟩) in your working copy, or to move changes between different branches.

In addition, each -j option can contain on optional date specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag. An example might be what `import` tells you to do when you have just imported sources that have conflicts with local changes:

```
$ cvs checkout -jTAG:yesterday -jTAG module
```

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

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

A.5.2 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.6 commit—Checks files into the repository

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

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

Use commit when you want to incorporate changes from your working source files into the source repository.

If you don't specify particular files to commit, all of the files in your working current directory are examined. commit is careful to change in the repository only those files that you have really changed. By default (or if you explicitly specify the '-R' option), files in subdirectories are also examined and committed if they have changed; you can use the '-l' option to limit `commit` to the current directory only.

`commit` verifies that the selected files are up to date with the current revisions in the source repository; it will notify you, and exit without committing, if any of the specified files must be made current first with `update` (see ⟨undefined⟩ [update], page ⟨undefined⟩). `commit` does not call the `update` command for you, but rather leaves that for you to do when the time is right.

When all is well, an editor is invoked to allow you to enter a log message that will be written to one or more logging programs (see ⟨undefined⟩ [modules], page ⟨undefined⟩, and see ⟨undefined⟩ [loginfo], page ⟨undefined⟩) and placed in the RCS history file inside the repository. This log message can be retrieved with the `log` command; See ⟨undefined⟩ [log], page ⟨undefined⟩. 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.6.1 commit options

These standard options are supported by `commit` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

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

`-n` Do not run any module program.

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

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

`commit` also supports these options:

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

`-f` This option is present in CVS 1.3-s3 and later releases of CVS. Note that this is not the standard meaning of the '-f' option as defined in See ⟨undefined⟩ [Common options], page ⟨undefined⟩.

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

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

`-f` *file*
: This option is present in CVS releases 1.3, 1.3-s1 and 1.3-s2. Note that this is not the standard meaning of the '`-f`' option as defined in See ⟨undefined⟩ [Common options], page ⟨undefined⟩.

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

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

A.6.2 commit examples

A.6.2.1 New major release number

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

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

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

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

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

A.6.2.2 Committing to a branch

You can commit to a branch revision (one that has an even number of dots) with the '`-r`' option. To create a branch revision, use the '`-b`' option of the `rtag` or `tag` commands (see ⟨undefined⟩ [tag], page ⟨undefined⟩ or see ⟨undefined⟩ [rtag], page ⟨undefined⟩). 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.6.2.3 Creating the branch after editing

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

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

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

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

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

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

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

A.7 diff—Runs diffs between revisions

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

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

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

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

A.7.1 diff options

These standard options are supported by `diff` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

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

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

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

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

`-Q`
: Really quiet.

`-q`
: Somewhat quiet.

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

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

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

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

A.7.2 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 expanded, so differences that only depends on keyword expansion 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.8 export—Export sources from CVS, similar to checkout

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

This command is a variant of `checkout`; use it when you want a copy of the source for module without the CVS administrative directories. For example, you might use `export` to prepare source for shipment off-site. This command requires that you specify a date or tag (with '`-D`' or '`-r`'), so that you can count on reproducing the source you ship to others.

The keyword expansion option '`-kv`' is always set when export is used. This causes any RCS keywords to be expanded such that an import done at some other site will not lose the keyword revision information. There is no way to override this. Note that this breaks the `ident` command (which is part of the RCS suite—see ident(1)) which looks for RCS keyword strings. If you want to be able to use `ident` you must use `checkout` instead.

A.8.1 export options

These standard options are supported by `export` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, 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.

`-Q` Really quiet.

`-q` Somewhat quiet.

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

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

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

`-d` *dir* Create a directory called *dir* for the working files, instead of using the module name. Unless you also use '`-N`', the paths created under *dir* will be as short as possible.

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

A.8.2 export examples

Contributed examples are gratefully accepted.

A.9 history—Shows status of files and users

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

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

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

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

A.9.1 history options

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

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

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

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

`-o` Report on checked-out modules.

`-T` Report on all tags.

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

Certain commands have a single record type:

`F` release

`O` checkout

`T`
: rtag

One of four record types may result from an update:

`C`
: A merge was necessary but collisions were detected (requiring manual merging).

`G`
: A merge was necessary and it succeeded.

`U`
: A working file was copied from the repository.

`W`
: The working copy of a file is 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*.

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

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

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

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

A.9.2 history examples

Contributed examples will gratefully be accepted.

A.10 import—Import sources into CVS, using vendor branches

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

Use `import` to incorporate an entire source distribution from an outside source (e.g., a source vendor) into your source repository directory. You can use this command both for initial creation of a repository, and for wholesale updates to the module from the outside source. See ⟨undefined⟩ [Tracking sources], page ⟨undefined⟩, for a discussion on this subject.

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

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

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

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

If the file '`$CVSROOT/CVSROOT/cvsignore`' exists, any files whose names match the specifications in that file will also be ignored.

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

At least three arguments are required. *repository* is needed to identify the collection of source. *vendortag* is a tag for the entire branch (e.g., for 1.1.1). You must also specify at least one *releasetag* to identify the files at the leaves created each time you execute `import`.

A.10.1 import options

These standard options are supported by `import` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

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

`-Q`
: Really quiet.

`-q`
: Somewhat quiet.

There are two additional special options.

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

`-I` *name*
: Specify file names that should be ignored during import. You can use this option repeatedly. To avoid ignoring any files at all (even those ignored by default), specify '-I !'.

  name can be a file name pattern of the same type that you can specify in the '`.cvsignore`' file. See ⟨undefined⟩ [cvsignore], page ⟨undefined⟩.

A.10.2 import examples

See ⟨undefined⟩ [Tracking sources], page ⟨undefined⟩.

A.11 log—Prints out 'rlog' information for files

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

Display log information for files. `log` calls the RCS utility `rlog`, which prints all available information about the RCS history file. This includes the location of the RCS file, the *head* revision (the latest revision on the trunk), all symbolic names (tags) and some other things. For each revision, the revision number, the author, the number of lines added/deleted and the log message are printed. All times are displayed in Coordinated Universal Time (UTC). (Other parts of CVS print times in the local timezone).

A.11.1 log options

Only one option is interpreted by CVS and not passed on to `rlog`:

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

By default, `rlog` prints all information that is available. All other options (including those that normally have other meanings) are passed through to `rlog` and restrict the output. See rlog(1) for a complete description of options. This incomplete list (which is a slightly edited extract from rlog(1)) lists all options that are useful in conjunction with CVS.

Please note: There can be no space between the option and its argument, since `rlog` parses its options in a different way than CVS.

`-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 ranges given by the semicolon-separated list of dates. The following table explains the available range formats:

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

<*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 date/time strings *d*, *d1*, and *d2* are in the free format explained in co(1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon (;).

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

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

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

rev1:*rev2* Revisions *rev1* to *rev2* (which must be on the same branch).

:*rev* Revisions from the beginning of the branch up to and including *rev*

rev: Revisions starting with *rev* to the end of the branch containing *rev*.

branch An argument that is a branch means all revisions on that branch. You can unfortunately not specify a symbolic branch here. You must specify the numeric branch number. See ⟨undefined⟩ [Branches and log], page ⟨undefined⟩.

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.

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

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

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

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

A.11.2 log examples

A.12 rdiff—'patch' format diffs between releases

- rdiff [-flags] [-V vn] [-r t|-D d [-r t2|-D d2]] modules...
- Requires: repository.
- Changes: nothing.
- Synonym: patch

Builds a Larry Wall format patch(1) file between two releases, that can be fed directly into the patch program to bring an old release up-to-date with the new release. (This is one of the few CVS commands that operates directly from the repository, and doesn't require a prior checkout.) The diff output is sent to the standard output device.

You can specify (using the standard '`-r`' and '`-D`' options) any combination of one or two revisions or dates. If only one revision or date is specified, the patch file reflects differences between that revision or date and the current head revisions in the RCS file.

Note that if the software release affected is contained in more than one directory, then it may be necessary to specify the '`-p`' option to the patch command when patching the old sources, so that patch is able to find the files that are located in other directories.

A.12.1 rdiff options

These standard options are supported by `rdiff` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, 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.

`-Q` Really quiet.

`-q` Somewhat quiet.

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

In addition to the above, these options are available:

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

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

`-t` A diff of the top two revisions is sent to the standard output device. This is most useful for seeing what the last change to a file was.

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

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

A.12.2 rdiff examples

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

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

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

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

A.13 release—Indicate that a Module is no longer in use

- release [-dQq] modules...
- Requires: Working directory.

- Changes: Working directory, history log.

This command is meant to safely cancel the effect of 'cvs checkout'. Since CVS doesn't lock files, it isn't strictly necessary to use this command. You can always simply delete your working directory, if you like; but you risk losing changes you may have forgotten, and you leave no trace in the CVS history file (see ⟨undefined⟩ [history file], page ⟨undefined⟩) that you've abandoned your checkout.

Use 'cvs release' to avoid these problems. This command checks that no un-committed changes are present; that you are executing it from immediately above a CVS working directory; and that the repository recorded for your files is the same as the repository defined in the module database.

If all these conditions are true, 'cvs release' leaves a record of its execution (attesting to your intentionally abandoning your checkout) in the CVS history log.

A.13.1 release options

Only these standard options are supported by `release`.

`-Q` Really quiet.

`-q` Somewhat quiet.

In addition to the above, it supports one additional flag.

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

Warning: The `release` command uses '`rm -r 'module'`' to delete your file. This has the very serious side-effect that any directory that you have created inside your checked-out sources, and not added to the repository (using the `add` command; see ⟨undefined⟩ [add], page ⟨undefined⟩) will be silently deleted—even if it is non-empty!

A.13.2 release output

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

Warning: Any new directories that you have created, but not added to the CVS directory hierarchy with the add command (see ⟨undefined⟩ [add], page ⟨undefined⟩) will be silently ignored, even if they contain files.

`U` *file*
: There exists a newer revision of this file in the repository, and you have not modified your local copy of the file.

`A` *file*
: The file has been added to your private copy of the sources, but has not yet been committed to the repository. If you delete your copy of the sources this file will be lost.

`R` *file*
: The file has been removed from your private copy of the sources, but has not yet been removed from the repository, since you have not yet committed the removal. See ⟨undefined⟩ [commit], page ⟨undefined⟩.

`M` *file*
: The file is modified in your working directory. There might also be a newer revision inside the repository.

`?` *file*
: *file* is in your working directory, but does not correspond to anything in the source repository, and is not in the list of files for CVS to ignore (see the description of the '`-I`' option, and see ⟨undefined⟩ [cvsignore], page ⟨undefined⟩). If you remove your working sources, this file will be lost.

  Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

A.13.3 release examples

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

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

A.14 remove—Removes an entry from the repository

- remove [-lR] [files...]
- Requires: Working directory.
- Changes: Working directory.

- Synonyms: rm, delete

Use this command to declare that you wish to remove files from the source repository. Like most CVS commands, 'cvs remove' works on files in your working directory, not directly on the repository. As a safeguard, it also requires that you first erase the specified files from your working directory.

The files are not actually removed until you apply your changes to the repository with `commit`; at that point, the corresponding RCS files in the source repository are moved into the 'Attic' directory (also within the source repository).

This command is recursive by default, scheduling all physically removed files that it finds for removal by the next commit. Use the '-l' option to avoid this recursion, or just specify the actual files that you wish removed.

A.14.1 remove options

Two of the standard options are the only options supported by `remove`.

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

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

A.14.2 remove examples

Remove a couple of files.

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

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

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

If you realise 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
```

A.15 rtag—Add a tag to the RCS file

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

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

A.15.1 rtag options

These standard options are supported by rtag (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, 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).

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

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

`-p`
: Pipe files to the standard output.

`-Q`
: Really quiet.

`-q`
: Somewhat quiet.

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

`-r` *tag*
: Only tag those files that contain *tag*. This can be used to rename a tag: tag only the files identified by the old tag, then delete the old tag, leaving the new tag on exactly the same files as the old tag.

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

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

`-b`
: Make the tag a branch tag. See ⟨undefined⟩ [Branches], page ⟨undefined⟩.

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

: In general, tags (often the symbolic names of software distributions) should not be removed, but the '`-d`' option is available as a means to remove completely obsolete symbolic names if necessary (as might be the case for an Alpha release, or if you mis-tagged a module).

A.15.2 rtag examples

A.16 status—Status info on the revisions

- status [-lR] [-v] [-Q] [files...]
- Requires: working directory, repository.

- Changes: nothing.

Display a brief report on the current status of files with respect to the source repository, including any sticky tags, dates, or '-k' options.

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

A.16.1 status options

These standard options are supported by status (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

-l Local; run only in current working directory.

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

-Q Really quiet. Do not print empty sticky parts.

There is one additional option:

-v Verbose. In addition to the information normally displayed, print all symbolic tags, together with the numerical value of the revision or branch they refer to.

A.16.2 status examples

A.17 tag—Add a symbolic tag to checked out version of RCS file

- tag [-lQqR] [-b] [-d] symbolic_tag [files...]
- Requires: working directory, repository.
- Changes: repository.
- Synonym: freeze

Use this command to assign symbolic tags to the nearest repository versions to your working sources. The tags are applied immediately to the repository, as with rtag, but the versions are supplied implicitly by the CVS records of your working files' history rather than applied explicitly.

One use for tags is to record a snapshot of the current sources when the software freeze date of a project arrives. As bugs are fixed after the freeze date, only those changed sources that are to be part of the release need be re-tagged.

The symbolic tags are meant to permanently record which revisions of which files were used in creating a software distribution. The `checkout` and `update` commands allow you to extract an exact copy of a tagged release at any time in the future, regardless of whether files have been changed, added, or removed since the release was tagged.

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

A.17.1 tag options

These standard options are supported by `tag` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

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

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

`-Q` Really quiet.

`-q` Somewhat quiet.

Two special options are available:

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

`-d` Delete a tag.

If you use '`cvs tag -d symbolic_tag`', the symbolic tag you specify is deleted instead of being added. Warning: Be very certain of your ground before you delete a tag; doing this effectively discards some historical information, which may later turn out to have been valuable.

A.17.2 tag examples

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

- update [-AdflPpQqR] [-d] [-r tag|-D date] files...
- Requires: repository, working directory.
- Changes: working directory.

After you've run checkout to create your private copy of source from the common repository, other developers will continue changing the central source. From time to time, when it is convenient in your development process, you can use the `update` command from within your working directory to reconcile your work with any revisions applied to the source repository since your last checkout or update.

A.18.1 update options

These standard options are available with `update` (see ⟨undefined⟩ [Common options], page ⟨undefined⟩, for a complete description of them):

`-D date` Use the most recent revision no later than *date*. This option is sticky, and implies '`-P`'.

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

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

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

`-P` Prune empty directories.

`-p` Pipe files to the standard output.

`-Q` Really quiet.

`-q` Somewhat quiet.

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

`-r tag` Retrieve revision *tag*. This option is sticky, and implies '`-P`'.

These special options are also available with `update`.

`-A` Reset any sticky tags, dates, or '`-k`' options. (If you get a working copy of a file by using one of the '`-r`', '`-D`', or '`-k`' options, CVS remembers the corresponding tag, date, or *kflag* and continues using it on future updates; use the '-A' option to make CVS forget these specifications, and retrieve the head revision of the file).

`-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. By default, `update` ignores files whose names match any of the following:

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

Use '-I !' to avoid ignoring any files at all. See ⟨undefined⟩ [cvsignore], page ⟨undefined⟩, for other ways to make CVS ignore some files.

`-j`*branch* Merge the changes made between the resulting revision and the revision that it is based on (e.g., if the tag refers to a branch, CVS will merge all changes made in that branch into your working file).

With two '-j' options, CVS will merge in the changes between the two respective revisions. This can be used to remove a certain delta from your working file; if the file 'foo.c' is based on revision 1.6 and you want to remove the changes made between 1.3 and 1.5, you might do:

```
$ cvs update -j1.5 -j1.3 foo.c   # note the order...
```

In addition, each -j option can contain on optional date specification which, when used with branches, can limit the chosen revision to one within a specific date. An optional date is specified by adding a colon (:) to the tag: '-j*Symbolic_Tag*:*Date_Specifier*'.

A.18.2 update output

`update` keeps you informed of its progress by printing a line for each file, prefaced with one character that indicate the status of the file:

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

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

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

`M` *file*
: The file is modified in your working directory.

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

  CVS will print some messages if it merges your work, and a backup copy of your working file (as it looked before you ran `update`) will be made. The exact name of that file is printed while `update` runs.

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

`?` *file*
: *file* is in your working directory, but does not correspond to anything in the source repository, and is not in the list of files for CVS to ignore (see the description of the '`-I`' option, and see ⟨undefined⟩ [cvsignore], page ⟨undefined⟩).

  Note that no warning message like this is printed for spurious directories that CVS encounters. The directory, and all its contents, are silently ignored.

A.18.3 update examples

The following line will display all files are not up-to-date without actually change anything in your working directory. It can be used to check what has been going on with the project.

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

Appendix B Reference manual for the Administrative files

Inside the repository, in the directory '$CVSROOT/CVSROOT', there are a number of supportive files for CVS. You can use CVS in a limited fashion without any of them, but if they are set up properly they can help make life easier.

The most important of these files is the 'modules' file, that defines the modules inside the repository.

B.1 The modules file

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

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

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

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

```
$ cvs checkout modules
```

to get an editable copy of the file. You should define similar module entries for the other configuration files described in this appendix, except 'history').

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

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

mname `-a` *aliases*...

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

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

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

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

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

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

A module definition can refer to other modules by including '`&`*module*' in its definition. `checkout` creates a subdirectory for each such module, in your working directory.

`-d` *name* Name the working directory something other than the module name.

`-i` *prog* Specify a program *prog* to run whenever files in a module are committed. *prog* runs with a single argument, the full pathname of the affected directory in a source repository. The '`commitinfo`', '`loginfo`', and '`editinfo`' files provide other ways to call a program on commit.

`-o` *prog* Specify a program *prog* to run whenever files in a module are checked out. *prog* runs with a single argument, the module name.

`-s` *status* Assign a status to the module. When the module file is printed with '`cvs checkout -s`' the modules are sorted according to primarily module status, and secondarily according to the module name. This option has no other meaning. You can use this option for several things besides status: for instance, list the person that is responsible for this module.

`-t` *prog* Specify a program *prog* to run whenever files in a module are tagged with `rtag`. *prog* runs with two arguments: the module name and the symbolic tag specified to `rtag`. There is no way to specify a program to run when `tag` is executed.

`-u` *prog* Specify a program *prog* to run whenever '`cvs update`' is executed from the top-level directory of the checked-out module. *prog* runs with a single argument, the full path to the source repository for this module.

B.2 The commit support files

The '-i' flag in the 'modules' file can be used to run a certain program whenever files are committed (see ⟨undefined⟩ [modules], page ⟨undefined⟩). The files described in this section provides other, more flexible, ways to run programs whenever something is committed.

There are three kind of programs that can be run on commit. They are specified in files in the repository, as described below. The following table summarizes the file names and the purpose of the corresponding programs.

'commitinfo'
: The program is responsible for checking that the commit is allowed. If it exits with a non-zero exit status the commit will be aborted.

'editinfo'
: The specified program is used to edit the log message, and possibly verify that it contains all required fields. This is most useful in combination with the 'rcsinfo' file, which can hold a log message template (see ⟨undefined⟩ [rcsinfo], page ⟨undefined⟩).

'loginfo'
: The specified program is called when the commit is complete. It receives the log message and some additional information and can store the log message in a file, or mail it to appropriate persons, or maybe post it to a local newsgroup, or... Your imagination is the limit!

B.2.1 The common syntax

The four files 'commitinfo', 'loginfo', 'rcsinfo' and 'editinfo' all have a common format. The purpose of the files are described later on. The common syntax is described here.

Each line contains the following:

- A regular expression
- A whitespace separator—one or more spaces and/or tabs.

- A file name or command-line template.

Blank lines are ignored. Lines that start with the character '#' are treated as comments. Long lines unfortunately can *not* be broken in two parts in any way.

Whenever one of the regular expression matches a directory name in the repository, the rest of the line is used.

B.3 Commitinfo

The '`commitinfo`' file defines programs to execute whenever '`cvs commit`' is about to execute. These programs are used for pre-commit checking to verify that the modified, added and removed files are really ready to be committed. This could be used, for instance, to verify that the changed files conform to to your site's standards for coding practice.

As mentioned earlier, each line in the '`commitinfo`' file consists of a regular expression and a command-line template. The template can include a program name and any number of arguments you wish to supply to it. The full path to the current source repository is appended to the template, followed by the file names of any files involved in the commit (added, removed, and modified files).

All lines with a regular expression matching the relative path to the module will be used. If any of the commands return 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.

If the name '`ALL`' appears as a regular expression it is always used in addition to any matching regular expression or '`DEFAULT`'.

B.4 Editinfo

If you want to make sure that all log messages look the same way, you can use the '`editinfo`' file to specify a program that is used to edit the log message. This program could be a custom-made editor that always enforces a certain style of the log message, or maybe a simple shell script that calls an editor, and checks that the entered message contains the required fields.

If no matching line is found in the 'editinfo' file, the editor specified in the environment variable $EDITOR is used instead. If that variable is not set a precompiled default, normally vi, will be used.

The 'editinfo' file is often most useful together with the 'rcsinfo' file, which can be used to specify a log message template.

Each line in the 'editinfo' file consists of a regular expression and a command-line template. The template must include a program name, and can include any number of arguments. The full path to the current log message template file is appended to the template.

One thing that should be noted is that the ALL keyword is not supported. If more than one matching line is found, the last one is used. This can be useful for specifying a default edit script in a module, and then overriding it in a subdirectory.

If the edit script exits with a non-zero exit status, the commit is aborted.

B.4.1 Editinfo example

The following is a little silly example of a 'editinfo' file, together with the corresponding 'rcsinfo' file, the log message template and an editor script. We begin with the log message template. We want to always record a bug-id number on the first line of the log message. The rest of log message is free text. The following template is found in the file '/usr/cvssupport/tc.template'.

```
BugId:
```

The script '/usr/cvssupport/bugid.edit' is used to edit the log message.

```
#!/bin/sh
#
#       bugid.edit filename
#
#  Call $EDITOR on FILENAME, and verify that the
#  resulting file contains a valid bugid on the first
#  line.
$EDITOR $1
until head -1|grep '^BugId:[ ]*[0-9][0-9]*$' < $1
do  echo -n  "No BugId found.  Edit again? ([y]/n)"
    read ans
    case ${ans} in
        n*) exit 1;;
```

```
        esac
        $EDITOR $1
    done
```

The 'editinfo' file contains this line:

```
^tc     /usr/cvssupport/bugid.edit
```

The 'rcsinfo' file contains this line:

```
^tc     /usr/cvssupport/tc.template
```

B.5 Loginfo

The 'loginfo' file is used to control where 'cvs commit' log information is sent. The first entry on a line is a regular expression which is tested against the directory that the change is being made to, relative to the $CVSROOT. If a match is found, then the remainder of the line is a filter program that should expect log information on its standard input.

The filter program may use one and only one % modifier (a la printf). If '%s' is specified in the filter program, a brief title is included (enclosed in single quotes) showing the modified file names.

If the repository name does not match any of the regular expressions in this file, the 'DEFAULT' line is used, if it is specified.

If the name 'ALL' appears as a regular expression it is always used in addition to any matching regular expression or 'DEFAULT'.

All matching regular expressions are used.

See ⟨undefined⟩ [commit files], page ⟨undefined⟩, for a description of the syntax of the 'loginfo' file.

B.5.1 Loginfo example

The following 'loginfo' file, together with the tiny shell-script below, appends all log messages to the file '$CVSROOT/CVSROOT/commitlog', and any commits to the administrative files (inside the 'CVSROOT' directory) are a also logged in '/usr/adm/cvsroot-log' and mailed to ceder.

```
ALL /usr/local/bin/cvs-log $CVSROOT/CVSROOT/commitlog
^CVSROOT Mail -s %s ceder
^CVSROOT /usr/local/bin/cvs-log /usr/adm/cvsroot-log
```

The shell-script '/usr/local/bin/cvs-log' looks like this:

```
#!/bin/sh
(echo "------------------------------------------------------------------";
 echo -n $USER"  ";
 date;
 echo;
 sed '1s+'${CVSROOT}'++') >> $1
```

B.6 Rcsinfo

The 'rcsinfo' file can be used to specify a form to edit when filling out the commit log. The 'rcsinfo' file has a syntax similar to the 'editinfo', 'commitinfo' and 'loginfo' files. See ⟨undefined⟩ [syntax], page ⟨undefined⟩. 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.

If the name 'ALL' appears as a regular expression it is always used in addition to the first matching regular expression or 'DEFAULT'.

The log message template will be used as a default log message. If you specify a log message with 'cvs commit -m *message*' or 'cvs commit -f *file*' that log message will override the template.

See ⟨undefined⟩ [editinfo example], page ⟨undefined⟩, for an example 'rcsinfo' file.

B.7 Ignoring files via cvsignore

There are certain file names that frequently occur inside your working copy, but that you don't want to put under CVS control. Examples are all the object files that you get while you compile your sources. Normally, when you run 'cvs update', it prints a line for each file it encounters that it doesn't know about (see ⟨undefined⟩ [update output], page ⟨undefined⟩).

CVS has a list of files (or sh(1) file name patterns) that it should ignore while running update. This list is constructed in the following way.

- The list is initialized to the following file name patterns:

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

- The per-repository list in '$CVSROOT/CVSROOT/cvsignore' is appended to the list, if that file exists.
- The per-user list in ''$HOME'/.cvsignore' is appended to the list, if it exists.
- As CVS traverses through your directories, the contents of any '.cvsignore' will be appended to the list. The patterns found in '.cvsignore' are only valid for the directory that contains them, not for any sub-directories.

B.8 The history file

The file '$CVSROOT/CVSROOT/history' is used to log information for the history command (see ⟨undefined⟩ [history], page ⟨undefined⟩). This file must be created to turn on logging. This is done automatically if the cvsinit script is used to set up the repository.

The file format of the 'history' file is unfortunately not yet documented anywhere, but it is fairly easy to understand most of it.

B.9 Setting up the repository

When you install CVS for the first time, you should follow the instructions in the 'INSTALL' file to set up the repository.

If you want to set up another repository, the easiest way to get a reasonable set of working administrative files is to get the source to CVS, and run the `cvsinit` shell script. It will set up an empty repository in the directory defined by the environment variable `$CVSROOT`. (`cvsinit` is careful to never overwrite any existing files in the repository, so no harm is done if you run `cvsinit` on an already set-up repository.)

Appendix C All environment variables that affects CVS

This is a complete list of all environment variables that affect CVS.

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

`$CVSROOT`
: Should contain the full pathname to the root of the CVS source repository (where the RCS history files are kept). This information must be available to CVS for most commands to execute; if `$CVSROOT` is not set, or if you wish to override it for one invocation, you can supply it on the command line: '`cvs -d cvsroot cvs_command`...' You may not need to set `$CVSROOT` if your CVS binary has the right path compiled in.

  If your site has several repositories, you must be careful to set `$CVSROOT` to the appropriate one when you use CVS, even if you just run '`cvs update`' inside an already checked-out module. Future releases of CVS will probably store information about which repository the module came from inside the '`CVS`' directory, but version 1.3 relies totally on `$CVSROOT`.

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

`$PATH`
: If `$RCSBIN` is not set, and no path is compiled into CVS, it will use `$PATH` to try to find all programs it uses.

`$RCSBIN`
: Specifies the full pathname of the location of RCS programs, such as co(1) and ci(1). If not set, a compiled-in value is used, or your `$PATH` is searched.

CVS is a front-end to RCS. The following environment variables affects RCS:

`$LOGNAME`
`$USER`
: If set, they affect who RCS thinks you are. If you have trouble checking in files it might be because your login name differs from the setting of e.g. `$LOGNAME`.

`$RCSINIT`
: Options prepended to the argument list, separated by spaces. A backslash escapes spaces within an option. The `$RCSINIT` options are prepended to the argument lists of most RCS commands.

`$TMPDIR`
`$TMP`
`$TEMP`
: Name of the temporary directory. The environment variables are inspected in the order they appear above and the first value found is taken; if none of them are set, a host-dependent default is used, typically '`/tmp`'.

Appendix D Troubleshooting

D.1 Bad administrative files

D.2 Branches and log

Appendix E GNU GENERAL PUBLIC LICENSE

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a

   1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a

   1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Table of Contents