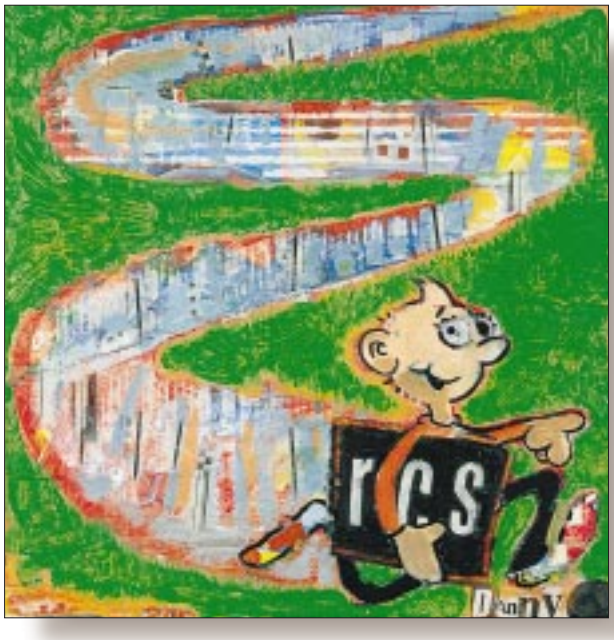
by Jeffreys Copeland and Haemer



Practical RCS

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS' R&D group in Boulder, CO. He's been a software consultant to the Hugo award administrators for several years. He spends his spare time raising children and cats.*

Jeffrey S. Haemer (jsh@canary.com) *now works for QMS, too, and is having a great time. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

```
mkdir RCS
ci -l *
```

These two lines are 90% of what you need. If you're waiting for a bus, or for a compile to finish, or eating lunch, we can help you kill some time. Kill, yes. Waste, no. If you already use Revision Control System (RCS), then by the time you're done reading this column, you'll feel more comfortable about what's going on underneath and you'll know some new tricks. But if you don't use RCS, start now.

Not knowing about the basics of RCS is as silly as not knowing the basics of `sed` or `awk`. It's a fundamental and useful tool, freely available, that runs on just about every kind of machine you can imagine, UNIX and non-UNIX. We use it daily.

'Use the Right Tool for the Job' – *Mr. Natural*

What's the job that RCS is the right tool for? Revision control. If you already know about revision control, skip this section.

Last month, we talked about how to make backup copies of files. We've all inadvertently deleted files, so we don't have to be sold on the idea of making snapshots of our work, but making frequent copies clutters up our directories, uses up a lot of space and becomes hard to organize and administer.

A lovely, and now venerable, alternative is provided by Walter F. Tichy's RCS, which lets you store all your backup snapshots of a file in a single repository.

To illustrate some advantages of RCS, let's look at a real example: `pclmain.c`, a file from one of the products that we work on. We have versions of `pclmain.c` that date back to 1992. The file itself is currently 642 lines long, and we have 152 distinct versions. Some of these versions are intermediate development versions, but others are actually parts of commercial releases, currently in the field, which require ongoing maintenance.

And `pclmain.c` is only one of 445 files that we use to build our interpreter for Hewlett-Packard's PCL printer language, which is just one of many components that go into our products.

Keeping track of all of this variation could be a tremendous amount of work. However, all the versions of `pclmain.c` are stored in a single file, `pclmain.c,v`, which is only about 5,500 lines long, nearly 1,000 of which are administrative notes

outside the source code itself.

The file has grown substantially in the last five years. However, even if it had stayed at its original size (about half its current size), storing all our revisions would have required more than 10 times as much space if we had stored them as complete files.

So RCS saves space and keeps the file system namespace manageable. What else?

• RCS automatically assigns identifiers to every stored revision, to permit future retrieval. We can also group components of a system under a single symbolic name, rather than needing to know the identifying number for each individual source component.

• RCS tags each revision of a file with the name of the person who made the revision and the exact time the revision was made. It also stores the author's comments on the revision external to the code itself, cleanly separating revision history from sources while joining the two physically, so there's no danger that one or the other will be lost.

• RCS controls access to the files, allowing everyone to read any version (RCS calls these "revisions") of a file, but only letting one person at a time lock a version of the file to make changes.

Figure 1. File jeff,v Under Revision Control

```
$ nl jeff,v
 1   head 1.1;
 2   access;
 3   symbols;
 4   locks; strict;
 5   comment   @# @;

 6   1.1
 7   date 97.05.02.16.50.41; author jsh;  state Exp;
 8   branches;
 9   next ;

 10  desc
 11  @This is a test file
 12  to illustrate the basics of RCS.
 13  @

 14  1.1
 15  log
 16  @Initial revision
 17  @
 18  text
 19  @Use the right tool for the job.
 20  @
```

(Note: The UNIX `nl` command numbers the nonblank lines of a file.)

One of the things we like about `vi` is that we're not afraid to make global, sweeping changes; if we make a mistake, we can always undo them with a single keystroke. RCS gives us a file-level "undo" key. We keep even trivial files (such as this column) under RCS control, confident that whenever we go down a dead-end street, we can always retreat to an earlier version–all the way back to our first draft, if necessary.

How's It Work?

Let's look at an example:

```
$ echo "Use the right tool for the job." > jeff
$ ci jeff
jeff,v  <--  jeff
enter description, terminated with single '.' or
end of file:
NOTE: This is NOT the log message!
>>
```

Here, we create a test file, then use the `ci` (check-in) command to put it under revision control. We'll use `ci` again later to check in revised versions of our file, but `ci` can tell that our test file isn't already under revision control, so it begins by asking for a description of the file, `NOT the log message!`. We type in an answer:

```
>> This is a test file
>> to illustrate the basics of RCS.
>> .
$ ls
jeff,v
```

When we started, we had a file called `jeff`. Now, we have a file called `jeff,v`. What does it look like? (See Figure 1.)

Fact number one: It's a simple UNIX text file; you can read and make sense of it. (If we had tried to put a binary file under revision control it would have worked, but the extra information added by RCS would still have been ASCII.)

In Figure 1, lines 1 to 5 contain administrative information. We'll talk more about them in a minute, but they contain keywords, such as `head` and `locks`, together with values, such as `1.1` (the last version number).

Fact number two: It's a file full of key-value pairs. Lines 10 to 13 contain the description we typed in. Notice that the key is `desc`, but the value–our description–is surrounded by @ characters. This is how RCS represents strings. If we have an @ sign within our text, RCS handles that by duplicating our @ sign.

The remainder of the file is a series of versions, or "deltas," which let us extract any given version of the file. The easiest way to see how they work is to add a few versions and look at the file again (see Figure 2).

What is all this stuff? Well, first, let's look at the current revision, which we get by checking it out:

```
$ co jeff
jeff,v  -->  jeff
revision 2.1
done
$ ls jeff*
jeff jeff,v
```

We now have both an RCS file, `jeff,v`, and the current version of the file, `jeff` (which is up to revision 2.1). We

know what the RCS file looks like, but what about the other one?

```
$ cat jeff
#$Id: jeff,v 2.1 1997/05/02 17:44:44 jsh Exp $

Snowy, Flowy, Blowy,
Showery, Flowery, Bowery,
Hoppy, Croppy, Droppy,
Breezy, Sneezy, Freezy.

  -- George Ellis, "The Twelve Months,"
```

This, too, is a regular file, even though it's very different from where we began. (How often do you suppose *SunExpert* prints poetry? Though contrast this with the names of the months in the French Revolutionary Calendar used between 1792 and 1806: Vintage, Fog, Sleet, Snow, Rain, Wind, Seed, Blossom, Pasture, Harvest, Heat and Fruit.) The first line may look mysterious, but it's only there because we put it there; it isn't required for RCS.

Now, let's look at the RCS file shown in Figure 2. Lines 1 to 5 are still administrative, global information about the file, and tell us that the most recent version is now 2.1.

Lines 6 to 17 list the individual revisions. Among other things, they tell us the number of each revision, who created it, which revision it was created from, and when it was checked in.

We occasionally find that we want a version of a file from a specific date or time. Accordingly, RCS takes dates and times very seriously. The check-in date is stored in UTC (what we used to call "Greenwich Mean Time"), which means that even if you have the RCS file on a file system that's NFS-mounted across time zones (yes, we sometimes do that!), the date will still make sense.

On the other hand, it's critical that all the machines that manipulate an RCS file have their clocks synchronized. On Suns, you can do this using `rdate`. For other machines, there are publicly available packages that use the Network Time Protocol. If all else fails, you can even create your own time-synchronization utility, which we showed you how to do a few months ago in *RS/Magazine* (see "Let's Synchronize Our Watches," January 1997, Page 30, or check out the software on our Web page at `http://www.alumni.caltech.edu/~copeland/work.html`).

The next part, the description, is unchanged. This is provided by the person who created the file and describes the file as a whole, not any particular revision, so there's no reason that creating new revisions should change anything. Comments on individual revisions will come later.

But what do you do if you want to change the description? This brings us to the catch-all command `rcs`. A quick look at the `rcs` man page reveals that this command lets you do all sorts of out-of-the-ordinary operations, including things like `rcs -t- 'Here's`

Figure 2. File jeff,v with Some Revisions

```
$ nl jeff,v
 1   head 2.1;
 2   access;
 3   symbols;
 4   locks; strict;
 5   comment   @# @;

 6   2.1
 7   date 97.05.02.17.44.44;  author jsh;  state Exp;
 8   branches;
 9   next 1.2;

 10  1.2
 11  date 97.05.02.17.40.38;  author jsh;  state Exp;
 12  branches;
 13  next 1.1;

 14  1.1
 15  date 97.05.02.16.50.41;  author jsh;  state Exp;
 16  branches;
 17  next ;

 18  desc
 19  @This is a test file
 20  to illustrate the basics of RCS.
 21  @

 22  2.1
 23  log
 24  @A third version, which changes a line.

 25  The file is now so different
 26  that we've given it a new major revision number.
 27  @
 28  text
 29  @# $ Id:$

 30  Snowy, Flowy, Blowy,
 31  Showery, Flowery, Bowery,
 32  Hoppy, Croppy, Droppy,
 33  Breezy, Sneezy, Freezy.

 34       -- George Ellis, "The Twelve Months,"
 35  @

 36  1.2
 37  log
 38  @Create a second revision by adding a few lines.
 39  @
 40  text
 41  @d1 1
 42  a1 1
 43  Use the right tool for the job.
 44  d7 2
 45  @

 46  1.1
 47  log
 48  @Initial revision
 49  @
 50  text
 51  @d2 5
 52  @
```

```
a new description.'
```

Fact number three: When all else fails, the `rcs(1)` man page often provides a solution. (This is, we note, a corollary of the more general Fact number zero: RTFM.)

The remainder of the file is a series of "deltatext" segments, each consisting of a revision number, a log comment and a string enclosed, as always, within @ symbols. The first of these, the "head" or "top-of-the-tree" revision, is the current version of the file. The command `co jeff` retrieves a copy of that text.

Before we go any further, we'll digress to point out line 29, the RCS `Id` keyword. RCS provides several such keywords, described in the *`co`*`(1)` man page, which let RCS insert various kinds of descriptive text into files as they are checked out. In Figure 2, the checked-out version of our file is labeled with the name of the repository, the version, the date and time that the version was created, and the developer who created it.

The deltatext segments that follow are instructions on how to reconstruct each version from the version that follows it. For example, lines 36 to 45 say that Version 1.2 can be constructed from Version 2.1 by deleting line 1, (line 41), adding one line after line 1 (line 42), the text of which is "`Use the right tool for the job`" and then the information on line 43 of the RCS archive tells us to delete several lines starting at line 2 of the file.

Of course, these instructions look just like instructions to `ed`. Instead of reinventing the wheel, RCS just uses existing UNIX tools to generate these deltatexts. A quick glance at the man page for `diff` or `diff3` will show you that both these utilities have options that generate `ed` scripts of this sort.

Broken down like this, the RCS file becomes quite comprehensible, and you can imagine that it would be simple both to generate and to interpret such files. You may be thinking you could write your own RCS substitute without much trouble. You probably could. For example, Brian W. Kernighan and Rob Pike do just that in *The UNIX Programming Environment* published by Prentice-Hall, 1984, ISBN 0-13-937699-2. Their `ci` equivalent, `put`, is a 30-line shell script, counting blank lines and comments. Their `co` equivalent, `get`, is three lines longer. "`get` is more complicated than `put`," they comment dryly, "mostly because it has options."

So what makes RCS any better than a roll-your-own version, and why has it swept the field, nearly completely displacing commercial competitors and its intellectual parent, SCCS (an older, historically important, UNIX version control system developed at AT&T by Marc Rochkind)? We can't speak for anyone else, of course, but we like it for these reasons:

1. It doesn't break.

Once you put a source under RCS (or anything like it), the RCS repository stores the accumulated history of that file: who worked on it, when it changed, why it changed and so on. In practice, RCS files are kept, used and maintained for years, and nothing that handles them can ever afford to break. RCS files become like the family jewels, and you can't afford to have them stepped on when you run into an obscure error condition that someone's forgotten to consider.

2. It's available in source code form, for free.

Not only does the price fit everyone's budget, but the availability of source means that as your organization switches from Suns running SunOS to DEC Alphas running OSF/1 to Pentium Pros running Linux, your old RCS files will migrate smoothly from each system to the next. In fact, there are versions of RCS for DOS, Windows and OS/2 for the unconvertible Microsofties out there.

3. It's easier to use than not to use.

First, RCS comes with scads of options and several auxiliary commands, but almost all of the work is done with two basic commands, `ci` and `co`, which you can learn in less than a minute.

Second, we all back up old versions in various ways, but with RCS, you end up backing them up into a single file and not cluttering up the directory listing. Better still, if you create a subdirectory called RCS, RCS will automatically store and look for its repository files in that directory. (See the first couple of lines of this column for an example.)

Third, several other ubiquitous UNIX software development tools, including `emacs` and GNU's version of `make`, know about RCS and RCS files. Fourth, RCS is built around simple command-line tools and normal files. UNIX has long made it relatively easy to build elaborately tailored interfaces on top of simpler ones, and graphics toolkits such as `Tk` now make it possible to layer on an attractive GUI. Going the other way is another story: It's usually extraordinarily hard to take an elaborate, integrated, one-size-fits-all, source-code-control database and tailor it for environments not contemplated by the vendor.

In addition to the three commands we've already mentioned–`co`, `ci` and `rcs`–RCS provides a handful of other useful, related utilities–including `ident`, `rcsclean`, `rcsdiff`, `rcsmerge` and `rlog`–that have their own man pages, which we encourage you to look over (review Fact number zero).

4. It's a de facto standard.

RCS is certain to be installed at nearly every company you will ever work for, and on almost every machine that you will ever work on. Time invested in learning RCS will continue to pay back for years, and once you've learned it, the knowledge will stay useful across projects, machines and employers. What's more, it's easy to continue to build your knowledge over the years by rereading the man pages every few months to learn something new.

Some Drawbacks

Over the years, we've learned that whenever someone tries to sell us on a new tool, product or idea, the question that cuts through the hyperbole faster than any other is this: "What's it *not* good for?" If we get an answer like, "It's good for *everything*. You should always use it," we say, "Gosh, thanks. Well, will you look at the time? We really must be going." Haemer was the first of us to discover this trick, which he applied at the first Usenix Conference on "very high-level languages" in October 1994. He asked everyone what C++ *wasn't* good for. The sometimes interesting answers are discussed in his conference report

in the February issue of the Usenix newsletter *;login:*.

What are RCS' shortcomings?

1. It's space intensive.

Whenever you have a lot of users looking at each file simultaneously, it's space intensive. RCS lets you extract a copy of any version you want, but if five people each get their own copy, then they take up five times as much disk space as a single copy.

2. File locking can be abused.

We once took charge of a source-code-control database with thousands of source files. A little investigation uncovered dozens of files that had been locked for years, most of them by employees who no longer worked at the company.

3. There is no way to force useful log comments.

This one is a little like complaining that there's no way to force programmers to write readable, maintainable C or Perl, but it's still a problem. We have seen too many log comments like "Fixed bug." (To be honest, we've even written too many like that ourselves.) Worse still, there's not even a way to force nonempty log comments.

4. RCS is built to handle files, not projects.

No project we've ever worked on has been confined to a single file. When we work on particular releases, we don't want to say, "Give me Version 1.3 of `foo.c` and Version 2.91 of `main.c` and …" we want to say, "Give me Release 2.02 of the product." Similarly, when we make changes, we're typically checking in new versions of a suite of files rather than a single file. Moreover, RCS has no provision for saying: "When we check out revisions older than January 1, 1995, the checked-out file should be called `jeffrey`, but for all newer revisions, the file should be called `jeff`."

On a related note, if we delete a file from a project, we must not delete the RCS file–otherwise, we could never reconstruct older releases–but RCS has no convenient way of marking the RCS files that correspond to deleted files. Files added partway through a project pose analogous problems.

One way to handle these sorts of problems is to abandon RCS for a product that has the needed functionality. This throws the baby out with the bath water. Another is to write a layer on top of RCS that provides the needed functionality. Most of us who have been in the UNIX industry for several years have done this more than once.

The good news is, that's no longer necessary. A few years back, Brian Berliner, at Prisma, wrote such a layer, called Concurrent Versions System (CVS), that has many of the advantages of RCS: It's mature, it's stable, it's well-designed, it's available in source form and it's free. Next month, we'll talk about CVS.

Until then, happy trails.