An Introduction to RCS
The Revision Control System

Last Edit
March 23, 1992

Reg Quinton <reggers@julian.uwo.ca>
Computing and Communications Services
The University of Western Ontario
London, Ontario N6A 5B7
Canada

1. Author

Walter F. Tichy, Purdue University, West Lafayette, Copyright (C) 1982, 1988, 1989 by Walter F. Tichy.

These notes are from *rcsintro*(1) by Tichy.

Within CCS we're currently running Revision Number: 5.6 ; Release Date: 1991/11/03; from cs.purdue.edu.

RCS is in the public domain and distributed by the GNU project (GNU is not Unix). Available by anonymous ftp from prep.ai.mit.edu (and uunet, julian, etc.).

Available on most Unix systems, or install the public domain version.

Beware
    there are incompatiblity problems between versions, eg. we installed the GNU version on our systems (Sun and Mips) to get compatibility.

The principle contender is SCCS (Source Code Control System). This is ATT proprietary code; that's why we don't use it.

2. What is it?

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions.

RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc.

The basic user interface is extremely simple. The novice only needs to learn two commands: *ci*(1L) and *co*(1L).

Ci, short for "check in", deposits the contents of a text file into an archival file called an RCS file. An RCS file contains all revisions of a particular text file.

Co,
short for "check out", retrieves revisions from an RCS file.

The remaining commands are *rlog* (to examine revision history), *rcs* (to manage RCS), *ident* (to check identifiers), and *rcsdiff* (to check differences). However, ci and co are enough to get your started.

3. Functions of RCS

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

Maintenance of a complete history of changes. RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of check-in, and a log message summarizing the change. The logging makes it easy to find out what happened to a module, without having to compare source listings or having to track down colleagues.

Resolution of access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.

Maintenance of a tree of Revisions. RCS can maintain separate lines of development for each module. It stores a tree structure that represents the ancestral relationships among revisions.

Merging of revisions and resolution of conflicts. Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged affect the same sections of code, RCS alerts the user about the overlapping changes.

Release and configuration control. Revisions can be assigned symbolic names and marked as released, stable, experimental, etc. With these facilities, configurations of modules can be described simply and directly.

Automatic identification of each revision with name, revision number, creation time, author, etc. The identification is like a stamp that can be embedded at an appropriate place in the text of a revision. The identification makes it simple to determine which revisions of which modules make up a given configuration.

Minimization of secondary storage. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

Moral
it's an important tool for making, documenting, and coordinating changes to the system.

4. Getting Started

Suppose you have a file f.c that you wish to put under control of RCS.

(1) Make an RCS subdirectory (this isn't required but will make life simpler).

```
% mkdir RCS
```

(2) Add this text, as a comment, to your file (this isn't required but will make life simpler).

```
$Author$
$Date$
$Id$
$Source$
```

(3) Invoke the check-in command

```
% ci f.c
```

(4) Invoke the check-out command

```
% co f.c
```

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

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

```
% co  f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. Note the comments ($Author$, $Date$, etc.) have been expanded by RCS:

```
% page f.c
...
$Author: reggers $
$Date: 1992/03/23 20:00:07 $
$Id: slides,v 1.2 1992/03/23 20:00:07 reggers Exp $
$Source: /usr/src/usr.local/doc/courses/rcs/RCS/slides,v $
...
```


5. Locks

You can co, edit f.c, and check it back in by invoking

```
% co -l f.c
% edit f.c
....testing
% ci f.c
```

Co -l creates a writable version and marks the file as locked:

```
% page f.c
...
$Author: reggers $
$Date: 1992/03/23 20:00:07 $
$Id: slides,v 1.2 1992/03/23 20:00:07 reggers Exp $
...
```

Locking a file prevents others from trying to do the same. It keeps people from tripping over one another.

The file locking is a good communications tool — it indicates that someone is working on the problem.

5.1. Skipping the lock

If *ci* complains with the message

    ci error: no lock set by <your login>

then you've editted a file that hasn't been marked as locked.

This would have required some effort — to get write access you have to *co -l* or have done a *chmod*.

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

```
% rcs -l  f.c
```

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

Moral
if you're working on a file lock it first!

5.2. Lock Summary

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

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

```
% rcs -U  f.c
% rcs -L  f.c
```

To get rid of a lock (say you've done *rcs -l* or *co -l* but decided against doing any changes):

```
% rcs -u  f.c
```

6. Ci [-l -u]

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

```
% ci -l  f.c
% ci -u  f.c
```

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

It's a bad idea to ci a revision and leave your working copy around — its RCS headers will say it's locked and this will confuse people.

Moral
  alias ci 'ci -u'

7. Revision Numbers

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

```
% ci -r2  f.c
% ci -r2.1  f.c
```

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

```
% co -r2  f.c
% co -r2.1  f.c
```

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields.

7.1. Branches

Usually revision numbers are a simple sequence of changes to a body of text numbered 1.1, 1.2, ..., 2.1, 2.2, ..., etc.

Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

```
% ci -r1.3.1  f.c
```

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

Branching usually isn't necessary (I've never used it).

8. Revision History

RCS maintains a revision history. Every time you check in a version you're asked for explanatory text. Why did you make these changes?

```
[11:33am zebra] ci Makefile
RCS/Makefile,v  <--  Makefile
new revision: 1.6; previous revision: 1.5
enter log message:
(terminate with ^D or single '.')
>> clean target should ignore error condition
>> .
done
```

Moral
: document what you've done and why you did it. Leave some foot prints for those who follow.

You examine the revision history (and RCS options you've set) by saying *rlog Makefile*:

9. Automatic Identification

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

```
$Id$
```

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

```
$Id: slides,v 1.2 1992/03/23 20:00:07 reggers Exp $
```

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

```
static char *rcsid=\
 "$Id: slides,v 1.2 1992/03/23 20:00:07 reggers Exp $";
```

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

```
% ident /usr/lib/sendmail
/usr/lib/sendmail:
    $Header: /usr/src/usr.local/rsendmail
    .../RCS/rsendmail.c,v 1.4 90/11/13
    ....13:36:21 reggers Exp $
```

You may also find it useful to put the marker Log into your text, inside a comment. This marker accumulates the log messages that are requested during check-in. Thus, you can maintain the complete history of your file directly inside it.

9.1. Keyword list

The complete set of keywords and their corresponding values is:

$Author$
: The login name of the user who checked in the revision. *Recommended* — who did this?

$Date$
: The date and time the revision was checked in. *Recommended* — when did they do this?

$Header$
: A standard header containing the full pathname of the RCS file, the revision number, the date, the author, the state, and the locker (if locked). *Recommended* — everything you need to know!

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

$Locker$
: The login name of the user who locked the revision (empty if not locked). *Recommended* — who is working on this file?

Log

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

$RCSfile$

The name of the RCS file without path.

$Revision$

The revision number assigned to the revision.

$Source$

The full pathname of the RCS file. *Recommended* — where are the sources?

$State$

The state assigned to the revision with the option of or Any identifier is acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, sets the state of a revision to *Exp*. *Recommended* — what's the status?

10. Group Projects

To create an access control list:

```
% rcs -areggers,peter,purple,mcote f.c
```

This gives the listed users the right to lock the file (ie. they can ci and co). Note that you have to list yourself.

Beware

This requires that the listed users have read/write priviledges for the directories (eg. can't co f.c if you can't write the file f.c). This is usually done with a common group:

```
% chmod 775 .
% chmod 775 RCS
```

Beware

This requires that the listed users create files that one another can read (eg. you can't co f.c if you can't read f.c,v). This is usually done with a common group and a default file mask:

```
% umask 027
```

11. See Also

ci(1L), co(1L), ident(1L), merge(1L), rcs(1L), rcsdiff(1L), rcsmerge(1L), rlog(1L), rcsfile(5L),

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

By anonymous ftp to julian.uwo.ca, the article *rcs.howto* in the directory ˜ftp/doc (also as /usr/doc).

By anonymous ftp to julian.uwo.ca, these slides in the directory ˜ftp/doc/courses/rcs.ps (also as /usr/doc).