RCS – A System for Version Control

The **R**evision **C**ontrol **S**ystem
Edition 1.1 RCS Version 5.7

by Walter F. Tichy, Paul Eggert

1 About the manual

1.1 Description

These 'Texinfo' files are hand converted out of the troff/nroff files, which came with the RCS system.

This has done by hand, cause there does not exist a converter which is able to convert (nroff/troff) into GNU's 'Texinfo' format (I know one, which is able to do the other way,'texinfo' to nroff).

The main reason for doing such a stupid work was, to have the opinion to convert the 'Texinfo' into OS/2-IPF format. This conversion is done with "Texinfo converting Tools Release 1.00" which are able to convert to OS/2-IPF format and HTML(not the Perl-script).

The next possibility is to get a well printed manual, using TEX(I use emTEX myself).

This is the first Edition of this manuals, so don't be angry about the cryptic node names within the 'Texinfo' files. That's the only way to have unique node names for alle command line options. And by the way this is the only way to make references to command line options of every command.

The next thing is to make this manual a little more readable, that means not to double subsection with command line options on every command (rlog, rcs, rcsdiff, ...). The way out of this dilemma is to discribe these options within a chapter 'overall options' or 'global options' which can be used on every command(rlog, rcs, rcsdiff...). But this will be done in the next edition, if you like it.

If you like to make a printed manual yourself, you have to TEX it with the 'texinfo' macro package(V2.185), it also works with version 2.150 of the package. It should also work with edition 2 of the 'Texinfo' macro package.

1.2 Versions

If you found newer versions of troff/nroff manual then please mail to me.

Here are the exact RCS–versions of the troff/roff pages for checking:

- Identification of the "RCS – A System for Version Control" manual part (rcs.ms):
  ```
  $Id: rcs.ms,v 5.4 1995/06/01 16:23:43 eggert Exp $
  ```
- Identification of the ci manual part(ci.1):
  ```
  $Id: ci.1,v 5.17 1995/06/16 06:19:24 eggert Exp $
  ```
- Identification of the co manual part(co.1):
  ```
  $Id: co.1,v 5.13 1995/06/01 16:23:43 eggert Exp $
  ```
- Identification of the rcs manual part(rcs.1):
  ```
  $Id: rcs.1,v 5.13 1995/06/05 08:28:35 eggert Exp $
  ```
- Identification of the ident manual part(ident.1):
  ```
  $Id: ident.1,v 5.4 1993/11/09 17:40:15 eggert Exp $
  ```
- Identification of the rcsclean manual part(rcsclean.1):

```
$Id: rcsclean.1,v 1.12 1993/11/03 17:42:27 eggert Exp $
```

- Identification of the rcsdiff manual part(rcsdiff.1):
  ```
  $Id: rcsdiff.1,v 5.5 1993/11/03 17:42:27 eggert Exp $
  ```
- Identification of the rcsmerge manual part(rcsmerge.1):
  ```
  $Id: rcsmerge.1,v 5.6 1995/06/01 16:23:43 eggert Exp $
  ```
- Identification of the rlog manual part(rlog.1):
  ```
  $Id: rlog.1,v 5.9 1995/06/16 06:19:24 eggert Exp $
  ```
- Identification of the merge manual part(merge.1):
  ```
  $Id: merge.1,v 5.7 1995/06/01 16:23:43 eggert Exp $
  ```

1.3 Bugs?, suggestions?

If you think you have found a bug within this manual, you could write by electronic mail a report.

Also, if you have suggestions for this manual. Changes of chapters, sections and so on.

Do you have good examples for this manual?

Or what else you like.

Please write to me:

```
Karl Heinz Marbaise
KHMarbaise@p69.ks.fido.de
Fido-Net: 2:2452/117.69
```

2 RCS–A System for Version Control

Walter F. Tichy
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

An important problem in program development and maintenance is version control, i.e., the task of keeping a software system consisting of many versions and configurations well organized. The Revision Control System (RCS) is a software tool that assists with that task. RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations. This paper introduces basic version control concepts and discusses the practice of version control using RCS. For conserving space, RCS stores deltas, i.e., differences between successive revisions. Several delta storage methods are discussed. Usage statistics show that RCS's delta storage method is space and time efficient. The paper concludes with a detailed survey of version control tools.

Keywords: configuration management, history management, version control, revisions, deltas.

An earlier version of this paper was published in *Software–Practice & Experience, 7 (July 1985), 637-654.*

2.1 Introduction

Version control is the task of keeping software systems consisting of many versions and configurations well organized. The Revision Control System (RCS) is a set of UNIX commands that assist with that task.

RCS' primary function is to manage **revision groups**. A revision group is a set of text documents, called **revisions**, that evolved from each other. A new revision is created by manually editing an existing one. RCS organizes the revisions into an ancestral tree. The initial revision is the root of the tree, and the tree edges indicate from which revision a given one evolved. Besides managing individual revision groups, RCS provides flexible selection functions for composing configurations. RCS may be combined with MAKE (See Section B.1 [Feldman], page 67), resulting in a powerful package for version control.

RCS also offers facilities for merging updates with customer modifications, for distributed software development, and for automatic identification. Identification is the '`stamping`' of revisions and configurations with unique markers. These markers are akin to serial numbers, telling software maintainers unambiguously which configuration is before them.

RCS is designed for both production and experimental environments. In production environments, access controls detect update conflicts and prevent overlapping changes. In experimental environments, where strong controls are counterproductive, it is possible to loosen the controls.

Although RCS was originally intended for programs, it is useful for any text that is revised frequently and whose previous revisions must be preserved. RCS has been applied successfully to store the source text for drawings, VLSI layouts, documentation, specifications, test data, form letters and articles.

This paper discusses the practice of version control using RCS. It also introduces basic version control concepts, useful for clarifying current practice and designing similar systems. Revision groups of individual components are treated in the next three sections, and the extensions to configurations follow. Because of its size, a survey of version control tools appears at the end of the paper.

2.2 Getting started with RCS

Suppose a text file '`f.c`' is to be placed under control of RCS. Invoking the check-in command

```
ci  f.c
```

creates a new revision group with the contents of '`f.c`' as the initial revision (numbered 1.1) and stores the group into the file '`f.c,v`'. Unless told otherwise, the command deletes '`f.c`'. It also asks for a description of the group. The description should state the common purpose of all revisions in the group, and becomes part of the group's documentation. All later check-in commands will ask for a log entry, which should summarize the changes made. (The first revision is assigned a default log message, which just records the fact that it is the initial revision.)

Files ending in '`,v`' are called **RCS files** (`v` stands for **Rersions**); the others are called working files. To get back the working file '`f.c`' in the previous example, execute the check-out command:

```
co  f.c
```

This command extracts the latest revision from the revision group '`f.c,v`' and writes it into '`f.c`'. The file '`f.c`' can now be edited and, when finished, checked back in with `ci`:

```
ci  f.c
```

`Ci` assigns number 1.2 to the new revision. If `ci` complains with the message

```
ci error: no lock set by <login>
```

then the system administrator has decided to configure RCS for a production environment by enabling the '`strict locking feature`'. If this feature is enabled, all RCS files are initialized such that check-in operations require a lock on the previous revision (the one from which the current one evolved). Locking prevents overlapping modifications if several people work on the same file. If locking is required, the revision should have been locked during the check-out by using the option `-l`:

```
co -l  f.c
```

Of course it is too late now for the check-out with locking, because '`f.c`' has already been changed; checking out the file again would overwrite the modifications. (To prevent accidental overwrites, `co` senses the presence of a working file and asks whether the user really intended to overwrite it. The overwriting check-out is sometimes useful for backing up to the previous revision.) To be able to proceed with the check-in in the present case, first execute

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

command retroactively locks the latest revision, unless someone else locked it in the meantime. In this case, the two programmers involved have to negotiate whose modifications should take precedence.

If an RCS file is private, i.e., if only the owner of the file is expected to deposit revisions into it, the strict locking feature is unnecessary and may be disabled. If strict locking is disabled, the owner of the RCS file need not have a lock for check-in. For safety reasons, all others still do. Turning strict locking off and on is done with the commands:

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

These commands enable or disable the strict locking feature for each RCS file individually. The system administrator only decides whether strict locking is enabled initially.

To reduce the clutter in a working directory, all RCS files can be moved to a subdirectory with the name `RCS`. RCS commands look first into that directory for RCS files. All the commands presented above work with the `RCS` subdirectory[1]. without change.

It may be undesirable that `ci` deletes the working file. For instance, sometimes one would like to save the current revision, but continue editing. Invoking

```
ci  -l  f.c
```

checks in '`f.c`' as usual, but performs an additional check-out with locking afterwards. Thus, the working file does not disappear after the check-in. Similarly, the option `-u` does a check-in followed by a check-out without locking. This option is useful if the file is needed for compilation after the check-in. Both options update the identification markers in the working file (see below).

Besides the operations `ci` and `co`, RCS provides the following commands:

`ident`	extract identification markers
`rcs`	change RCS file attributes
`rcsclean`	remove unchanged working files (optional)
`rcsdiff`	compare revisions
`rcsfreeze`	record a configuration (optional)
`rcsmerge`	merge revisions
`rlog`	read log messages and other information in RCS files

A synopsis of these commands appears in the Appendix.

[1] Pairs of RCS and working files can actually be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. If a pair is given, both files may have arbitrary path prefixes; RCS commands pair them up intelligently

2.2.1 Automatic Identification

RCS can stamp source and object code with special identification strings, similar to product and serial numbers. To obtain such identification, place the marker

```
$Id$
```

into the text of a revision, for instance inside a comment. The check-out operation will replace this marker with a string of the form

```
$Id:  filename  revisionnumber  date  time  author state  locker $
```

This string need never be touched, because `co` keeps it up to date automatically. To propagate the marker into object code, simply put it into a literal character string. In C, this is done as follows:

```
static char rcsid[] = "$Id$";
```

The command `ident` extracts such markers from any file, in particular from object code. `Ident` helps to find out which revisions of which modules were used in a given program. It returns a complete and unambiguous component list, from which a copy of the program can be reconstructed. This facility is invaluable for program maintenance.

There are several additional identification markers, one for each component of Id. The marker

```
$Log$
```

has a similar function. It accumulates the log messages that are requested during check-in. Thus, one can maintain the complete history of a revision directly inside it, by enclosing it in a comment. Figure 1 is an edited version of a log contained in revision 4.1 of the file '`ci.c`'. The log appears at the beginning of the file, and makes it easy to determine what the recent modifications were.

```
/*
 * $Log: ci.c,v $
 * Revision 4.1  1983/05/10 17:03:06  wft
 * Added option -d and -w, and updated assignment of date, etc.
 * to new delta. Added handling of default branches.
 *
 * Revision 3.9  1983/02/15 15:25:44  wft
 * Added call to fastcopy() to copy remainder of RCS file.
 *
 * Revision 3.8  1983/01/14 15:34:05  wft
 * Added ignoring of interrupts while new RCS file is renamed;
 * avoids deletion of RCS files by interrupts.
 *
 * Revision 3.7  1982/12/10 16:09:20  wft
 * Corrected checking of return code from diff.
 * An RCS file now inherits its mode during the first ci from the
 * working file, except that write permission is removed.
 */
```

Figure 1. Log entries produced by the marker Log

Since revisions are stored in the form of differences, each log message is physically stored once, independent of the number of revisions present. Thus, the Log marker incurs negligible space overhead.

2.3 The RCS Revision Tree

RCS arranges revisions in an ancestral tree. The `ci` command builds this tree; the auxiliary command `rcs` prunes it. The tree has a root revision, normally numbered 1.1, and successive revisions are numbered 1.2, 1.3, etc. The first field of a revision number is called the `release number` and the second one the `level number`. Unless given explicitly, the `ci` command assigns a new revision number by incrementing the level number of the previous revision. The release number must be incremented explicitly, using the `-r` option of `ci`. Assuming there are revisions 1.1, 1.2, and 1.3 in the RCS file '`f.c,v`', the command

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

assigns the number 2.1 to the new revision. Later check-ins without the `-r` option will assign the numbers 2.2, 2.3, and so on. The release number should be incremented only at major transition points in the development, for instance when a new release of a software product has been completed.

2.3.1 When are branches needed?

A young revision tree is slender: It consists of only one branch, called the trunk. As the tree ages, side branches may form. Branches are needed in the following 4 situations.

1. Temporary fixes

   Suppose a tree has 5 revisions grouped in 2 releases, as illustrated in Figure 2. Revision 1.3, the last one of release 1, is in operation at customer sites, while release 2 is in active development.

```
+-----+     +-----+     +-----+     +-----+     +-----+
! 1.1 !---->! 1.2 !---->! 1.3 !---->! 2.1 !---->! 2.2 !--->>
+-----+     +-----+     +-----+     +-----+     +-----+
```

Figure 2. A slender revision tree.

   Now imagine a customer requesting a fix of a problem in revision 1.3, although actual development has moved on to release 2. RCS does not permit an extra revision to be spliced in between 1.3 and 2.1, since that would not reflect the actual development history. Instead, create a branch at revision 1.3, and check in the fix on that branch. The first branch starting at 1.3 has number 1.3.1, and the revisions on that branch are numbered 1.3.1.1, 1.3.1.2, etc. The double numbering is needed to allow for another branch at 1.3, say 1.3.2. Revisions on the second branch would be numbered 1.3.2.1, 1.3.2.2, and so on. The following steps create branch 1.3.1 and add revision 1.3.1.1:

```
co -r1.3 f.c      -- check out revision 1.3
edit f.c          -- change it
ci  -r1.3.1  f.c -- check it in on branch 1.3.1
```

This sequence of commands transforms the tree of Figure 2 into the one in Figure 3. Note that it may be necessary to incorporate the differences between 1.3 and 1.3.1.1 into a revision at level 2. The operation `rcsmerge` automates this process (see the Appendix).

```
+-----+     +-----+     +-----+     +-----+     +-----+
! 1.1 !---->! 1.2 !---->! 1.3 !---->! 2.1 !---->! 2.2 !--->>
+-----+     +-----+     +--+--+     +-----+     +-----+
                           !
                           !        +---------+
                           +------->! 1.3.1.1 !---->>
                                    +---------+
```

Figure 3. A revision tree with one side branch

2. Distributed development and customer modifications

   Assume a situation as in Figure 2, where revision 1.3 is in operation at several customer sites, while release 2 is in development. Customer sites should use RCS to store the distributed software. However, customer modifications should not be placed on the same branch as the distributed source; instead, they should be placed on a side branch. When the next software distribution arrives, it should be appended to the trunk of the customer's RCS file, and the customer can then merge the local modifications back into the new release. In the above example, a customer's RCS file would contain the following tree, assuming that the customer has received revision 1.3, added his local modifications as revision 1.3.1.1, then received revision 2.4, and merged 2.4 and 1.3.1.1, resulting in 2.4.1.1.

```
    +-----+                  +-----+
--->! 1.3 !----------------->! 2.4 !---->>
    +--+--+                  +---+-+
       !                         !
       !     +---------+         !       +---------+
       +---->! 1.3.1.1 !         +------>! 2.4.1.1 !
             +---------+                 +---------+
```

Figure 4. A customer's revision tree with local modifications.

   This approach is actually practiced in the CSNET project, where several universities and a company cooperate in developing a national computer network.

3. Parallel development

   Sometimes it is desirable to explore an alternate design or a different implementation technique in parallel with the main line development. Such development should be carried out on a side branch. The experimental changes may later be moved into the main line, or abandoned.

4. Conflicting updates

A common occurrence is that one programmer has checked out a revision, but cannot complete the assignment for some reason. In the meantime, another person must perform another modification immediately. In that case, the second person should check-out the same revision, modify it, and check it in on a side branch, for later merging.

Every node in a revision tree consists of the following attributes: a revision number, a check-in date and time, the author's identification, a log entry, a state and the actual text. All these attributes are determined at the time the revision is checked in. The state attribute indicates the status of a revision. It is set automatically to 'experimental' during check-in. A revision can later be promoted to a higher status, for example 'stable' or 'released'. The set of states is user-defined.

2.3.2 Revisions are represented as deltas

For conserving space, RCS stores revisions in the form of deltas, i.e., as differences between revisions. The user interface completely hides this fact.

A delta is a sequence of edit commands that transforms one string into another. The deltas employed by RCS are line-based, which means that the only edit commands allowed are insertion and deletion of lines. If a single character in a line is changed, the edit scripts consider the entire line changed. The program `diff` produces a small, line-based delta between pairs of text files. A character-based edit script would take much longer to compute, and would not be significantly shorter.

Using deltas is a classical space-time tradeoff: deltas reduce the space consumed, but increase access time. However, a version control tool should impose as little delay as possible on programmers. Excessive delays discourage the use of version controls, or induce programmers to take shortcuts that compromise system integrity. To gain reasonably fast access time for both editing and compiling, RCS arranges deltas in the following way. The most recent revision on the trunk is stored intact. All other revisions on the trunk are stored as reverse deltas. A reverse delta describes how to go backward in the development history: it produces the desired revision if applied to the successor of that revision. This implementation has the advantage that extraction of the latest revision is a simple and fast copy operation. Adding a new revision to the trunk is also fast: `ci` simply adds the new revision intact, replaces the previous revision with a reverse delta, and keeps the rest of the old deltas. Thus, `ci` requires the computation of only one new delta.

Branches need special treatment. The naive solution would be to store complete copies for the tips of all branches. Clearly, this approach would cost too much space. Instead, RCS uses `forward` deltas for branches. Regenerating a revision on a side branch proceeds as follows. First, extract the latest revision on the trunk; secondly, apply reverse deltas until the fork revision for the branch is obtained; thirdly, apply forward deltas until the desired branch revision is reached. Figure 5 illustrates a tree with one side branch. Triangles pointing to the left and right(with five exclamation marks) represent reverse and forward deltas, respectively.

```
      !           !           !           !           !
+-----!     +-----!     +-----!     +-----!     +-----!
! 1.1 !---->! 1.2 !---->! 1.3 !---->! 2.1 !---->! 2.2 !--->>
+-----+     +-----!     +--+--!     +-----!     +-----!
      !           !        !  !           !           !
                           !
                           !            !               !
                           !  +---------!     +---------!
                           +->! 1.3.1.1 !---->! 1.3.1.2 !
                              +---------!     +---------!
                                        !               !
```

Figure 3. A revision tree with one side branch

Although implementing fast check-out for the latest trunk revision, this arrangement has the disadvantage that generation of other revisions takes time proportional to the number of deltas applied. For example, regenerating the branch tip in Figure 5 requires application of five deltas (including the initial one). Since usage statistics show that the latest trunk revision is the one that is retrieved in 95 per cent of all cases (see the section on usage statistics), biasing check-out time in favor of that revision results in significant savings. However, careful implementation of the delta application process is necessary to provide low retrieval overhead for other revisions, in particular for branch tips.

There are several techniques for delta application. The naive one is to pass each delta to a general-purpose text editor. A prototype of RCS invoked the UNIX editor `ed` both for applying deltas and for expanding the identification markers. Although easy to implement, performance was poor, owing to the high start-up costs and excess generality of `ed`. An intermediate version of RCS used a special-purpose, stream-oriented editor. This technique reduced the cost of applying a delta to the cost of checking out the latest trunk revision. The reason for this behavior is that each delta application involves a complete pass over the preceding revision.

However, there is a much better algorithm. Note that the deltas are line oriented and that most of the work of a stream editor involves copying unchanged lines from one revision to the next. A faster algorithm avoids unnecessary copying of character strings by using a `piece table`. A piece table is a one-dimensional array, specifying how a given revision is '`pieced-together`' from lines in the RCS file. Suppose piece table `PTr` represents revision `r`. Then `PTr[i]` contains the starting position of line `i` of revision `r`. Application of the next delta transforms piece table `PTr` into `PTr+1`. For instance, a delete command removes a series of entries from the piece table. An insertion command inserts new entries, moving the entries following the insertion point further down the array. The inserted entries point to the text lines in the delta. Thus, no I/O is involved except for reading the delta itself. When all deltas have been applied to the piece table, a sequential pass through the table looks up each line in the RCS file and copies it to the output file, updating identification markers at the same time. Of course, the RCS file must permit random access, since the copied lines are scattered throughout that file. Figure 6 illustrates an RCS file with two revisions and the corresponding piece tables.

The piece table approach has the property that the time for applying a single delta is roughly determined by the size of the delta, and not by the size of the revision. For example, if a delta is 10 per cent of the size of a revision, then applying it takes only 10 per cent of the time to generate the latest trunk revision. (The stream editor would take 100 per cent.)

There is an important alternative for representing deltas that affects performance. `SCCS`, a precursor of RCS, uses `interleaved` deltas. A file containing interleaved deltas is partitioned into blocks of lines. Each block has a header that specifies to which revision(s) the block belongs. The blocks are sorted out in such a way that a single pass over the file can pick up all the lines belonging to a given revision. Thus, the regeneration time for all revisions is the same: all headers must be inspected, and the associated blocks either copied or skipped. As the number of revisions increases, the cost of retrieving any revision is much higher than the cost of checking out the latest trunk revision with reverse deltas. A detailed comparison of `SCCS's` interleaved deltas and RCS's reverse deltas can be found in Reference 4. This reference considers the version of RCS with the stream editor only. The piece table method improves performance further, so that RCS is always faster than SCCS, except if 10 or more deltas are applied.

Additional speed-up for both delta methods can be obtained by caching the most recently generated revision, as has been implemented in `DSEE` With caching, access time to frequently used revisions can approach normal file access time, at the cost of some additional space.

2.4 Locking: A Controversial Issue

The locking mechanism for RCS was difficult to design. The problem and its solution are first presented in their '`pure`' form, followed by a discussion of the complications caused by '`real-world`' considerations.

RCS must prevent two or more persons from depositing competing changes of the same revision. Suppose two programmers check out revision 2.4 and modify it. Programmer A checks in a revision before programmer **B**. Unfortunately, programmer B has not seen A's changes, so the effect is that A's changes are covered up by B's deposit. A's changes are not lost since all revisions are saved, but they are confined to a single revision[2].

This conflict is prevented in RCS by locking. Whenever someone intends to edit a revision (as opposed to reading or compiling it), the revision should be checked out and locked, using the `-l` option on `co`. On subsequent check-in, `ci` tests the lock and then removes it. At most one programmer at a time may lock a particular revision, and only this programmer may check in the succeeding revision. Thus, while a revision is locked, it is the exclusive responsibility of the locker.

An important maxim for software tools like RCS is that they must not stand in the way of making progress with a project. This consideration leads to several weakenings of the locking mechanism. First of all, even if a revision is locked, it can still be checked out. This is necessary if other people wish to compile or inspect the locked revision while the next

[2] Note that this problem is entirely different from the atomicity problem. Atomicity means that concurrent update operations on the same RCS file cannot be permitted, because that may result in inconsistent data. Atomic updates are essential (and implemented in RCS), but do not solve the conflict discussed here.

one is in preparation. The only operations they cannot do are to lock the revision or to check in the succeeding one. Secondly, check-in operations on other branches in the RCS file are still possible; the locking of one revision does not affect any other revision. Thirdly, revisions are occasionally locked for a long period of time because a programmer is absent or otherwise unable to complete the assignment. If another programmer has to make a pressing change, there are the following three alternatives for making progress:

- find out who is holding the lock and ask that person to release it;
- check out the locked revision, modify it, check it in on a branch, and merge the changes later;
- break the lock. Breaking a lock leaves a highly visible trace, namely an electronic mail message that is sent automatically to the holder of the lock, recording the breaker and a commentary requested from him. Thus, breaking locks is tolerated under certain circumstances, but will not go unnoticed. Experience has shown that the automatic mail message attaches a high enough stigma to lock breaking, such that programmers break locks only in real emergencies, or when a co-worker resigns and leaves locked revisions behind.

If an RCS file is private, i.e., when a programmer owns an RCS file and does not expect anyone else to perform check-in operations, locking is an unnecessary nuisance. In this case, the '`strict locking feature`' discussed earlier may be disabled, provided that file protection is set such that only the owner may write the RCS file. This has the effect that only the owner can check-in revisions, and that no lock is needed for doing so.

As added protection, each RCS file contains an access list that specifies the users who may execute update operations. If an access list is empty, only normal UNIX file protection applies. Thus, the access list is useful for restricting the set of people who would otherwise have update permission. Just as with locking, the access list has no effect on read-only operations such as `co`. This approach is consistent with the UNIX philosophy of openness, which contributes to a productive software development environment.

2.5 Configuration Management

The preceding sections described how `RCS` deals with revisions of individual components; this section discusses how to handle configurations. A configuration is a set of revisions, where each revision comes from a different revision group, and the revisions are selected according to a certain criterion. For example, in order to build a functioning compiler, the 'right' revisions from the scanner, the parser, the optimizer and the code generator must be combined. `RCS`, in conjunction with `MAKE`, provides a number of facilities to effect a smooth selection.

2.5.1 RCS Selection Functions

- Default selection

  During development, the usual selection criterion is to choose the latest revision of all components. The `co` command makes this selection by default. For example, the command

```
co  *,v
```

retrieves the latest revision on the default branch of each RCS file in the current directory. The default branch is usually the trunk, but may be set to be a side branch. Side branches as defaults are needed in distributed software development, as discussed in the section on the RCS revision tree.

– Release based selection

  Specifying a release or branch number selects the latest revision in that release or branch. For instance,

  ```
  co  -r2  *,v
  ```

  retrieves the latest revision with release number 2 from each RCS file. This selection is convenient if a release has been completed and development has moved on to the next release.

– State and author based selection

  If the highest level number within a given release number is not the desired one, the state attribute can help. For example,

  ```
  co  -r2  -sReleased  *,v
  ```

  retrieves the latest revision with release number 2 whose state attribute is 'Released'. Of course, the state attribute has to be set appropriately, using the `ci` or `rcs` commands. Another alternative is to select a revision by its author, using the `-w` option.

– Date based selection

  Revisions may also be selected by date. Suppose a release of an entire system was completed and current on March 4, at 1:00 p.m. local time. Then the command

  ```
  co  -d'March 4, 1:00 pm LT'  *,v
  ```

  checks out all the components of that release, independent of the numbering. The `-d` option specifies a 'cutoff date', i.e., the revision selected has a check-in date that is closest to, but not after the date given.

– Name based selection

  The most powerful selection function is based on assigning symbolic names to revisions and branches. In large systems, a single release number or date is not sufficient to collect the appropriate revisions from all groups. For example, suppose one wishes to combine release 2 of one subsystem and release 15 of another. Most likely, the creation dates of those releases differ also. Thus, a single revision number or date passed to the `co` command will not suffice to select the right revisions. Symbolic revision numbers solve this problem. Each RCS file may contain a set of symbolic names that are mapped to numeric revision numbers. For example, assume the symbol `V3` is bound to release number 2 in file '`s,v`', and to revision number 15.9 in '`t,v`'. Then the single command

  ```
  co  -rV3  s,v  t,v
  ```

  retrieves the latest revision of release 2 from '`s,v`', and revision 15.9 from '`t,v`'. In a large system with many modules, checking out all revisions with one command greatly simplifies configuration management.

  Judicious use of symbolic revision numbers helps with organizing large configurations.

  A special command, `rcsfreeze`, assigns a symbolic revision number to a selected revision in every RCS file. `rcsfreeze` effectively freezes a configuration. The assigned symbolic revision number selects all components of the configuration. If necessary, symbolic numbers may even be intermixed with numeric ones. Thus, `V3.5` in the above example would select revision 2.5 in '`s,v`' and branch 15.9.5 in '`t,v`'.

The options `-r`, `-s`, `-w` and `-d` may be combined. If a branch is given, the latest revision on that branch satisfying all conditions is retrieved; otherwise, the default branch is used.

2.5.2 Combining MAKE and RCS

MAKE (Section B.1 [Feldman], page 67) is a program that processes configurations. It is driven by configuration specifications recorded in a special file, called a 'Makefile'. MAKE avoids redundant processing steps by comparing creation dates of source and processed objects. For example, when instructed to compile all modules of a given system, it only recompiles those source modules that were changed since they were processed last.

MAKE has been extended with an auto-checkout feature[3] for RCS.* When a certain file to be processed is not present, MAKE attempts a check-out operation. If successful, MAKE performs the required processing, and then deletes the checked out file to conserve space. The selection parameters discussed above can be passed to MAKE either as parameters, or directly embedded in the Makefile. MAKE has also been extended to search the subdirectory named `RCS` for needed files, rather than just the current working directory. However, if a working file is present, MAKE totally ignores the corresponding RCS file and uses the working file. (In newer versions of MAKE distributed by AT&T and others, auto-checkout can be achieved with the rule DEFAULT, instead of a special extension of MAKE. However, a file checked out by the rule DEFAULT will not be deleted after processing. `Rcsclean` can be used for that purpose.)

With auto-checkout, RCS/MAKE can effect a selection rule especially tuned for multi-person software development and maintenance. In these situations, programmers should obtain configurations that consist of the revisions they have personally checked out plus the latest checked in revision of all other revision groups. This schema can be set up as follows.

Each programmer chooses a working directory and places into it a symbolic link, named `RCS`, to the directory containing the relevant RCS files. The symbolic link makes sure that `co` and `ci` operations need only specify the working files, and that the Makefile need not be changed. The programmer then checks out the needed files and modifies them. If MAKE is invoked, it composes configurations by selecting those revisions that are checked out, and the rest from the subdirectory `RCS`. The latter selection may be controlled by a symbolic revision number or any of the other selection criteria. If there are several programmers editing in separate working directories, they are insulated from each other's changes until checking in their modifications.

Similarly, a maintainer can recreate an older configuration by starting to work in an empty working directory. During the initial MAKE invocation, all revisions are selected from RCS files. As the maintainer checks out files and modifies them, a new configuration is gradually built up. Every time MAKE is invoked, it substitutes the modified revisions into the configuration being manipulated.

[3] This auto-checkout extension is available only in some versions of MAKE, e.g. GNU MAKE.

A final application of RCS is to use it for storing Makefiles. Revision groups of Makefiles represent multiple versions of configurations. Whenever a configuration is baselined or distributed, the best approach is to unambiguously fix the configuration with a symbolic revision number by calling `rcsfreeze`, to embed that symbol into the Makefile, and to check in the Makefile (using the same symbolic revision number). With this approach, old configurations can be regenerated easily and reliably.

2.6 Usage Statistics

The following usage statistics were collected on two DEC VAX-11/780 computers of the Purdue Computer Science Department. Both machines are mainly used for research purposes. Thus, the data reflect an environment in which the majority of projects involve prototyping and advanced software development, but relatively little long-term maintenance.

For the first experiment, the `ci` and `co` operations were instrumented to log the number of backward and forward deltas applied. The data were collected during a 13 month period from Dec. 1982 to Dec. 1983. Table I summarizes the results.

Oper.	Total	Total deltas	mean deltas	Operations	Branch
	operations	applied	applied	with >1 delta	operations
co	7867	9320	1.18	509 (6%)	203 (3%)
ci	3468	2207	0.64	85 (2%)	75 (2%)
ci & co	11335	11527	1.02	594 (5%)	278 (2%)

Table I. Statistics for co and ci operations

The first two lines show statistics for check-out and check-in; the third line shows the combination. Recall that `ci` performs an implicit check-out to obtain a revision for computing the delta. In all measures presented, the most recent revision (stored intact) counts as one delta. The number of deltas applied represents the number of passes necessary, where the first 'pass' is a copying step.

Note that the check-out operation is executed more than twice as frequently as the check-in operation. The fourth column gives the mean number of deltas applied in all three cases. For `ci`, the mean number of deltas applied is less than one. The reasons are that the initial check-in requires no delta at all, and that the only time `ci` requires more than one delta is for branches. Column 5 shows the actual number of operations that applied more than one delta. The last column indicates that branches were not used often.

The last three columns demonstrate that the most recent trunk revision is by far the most frequently accessed. For RCS, check-out of this revision is a simple copy operation, which is the absolute minimum given the copy-semantics of `co`. Access to older revisions and branches is more common in non-academic environments, yet even if access to older deltas were an order of magnitude more frequent, the combined average number of deltas applied would still be below 1.2. Since RCS is faster than SCCS until up to 10 delta applications, reverse deltas are clearly the method of choice. .PP The second experiment, conducted in March of 1984, involved surveying the existing RCS files on our two machines. The goal was to determine the mean number of revisions per RCS file, as well as the space consumed by them. Table II shows the results. (Tables I and II were produced at different times and are unrelated.)

	Total RCS files	Total revisions	Mean revisions	Means size RCS files	Mean size revisions	Overhead
All Files	8033	11133	1.39	6156	5585	1.10
Files with >= 2 delta	1477	4578	3.10	8074	6041	1.34

Table II. Statistics for RCS files

The mean number of revisions per RCS file is 1.39. Columns 5 and 6 show the mean sizes (in bytes) of an RCS file and of the latest revision of each RCS file, respectively. The ‘overhead’ column contains the ratio of the mean sizes. Assuming that all revisions in an RCS file are approximately the same size, this ratio gives a measure of the space consumed by the extra revisions.

In our sample, over 80 per cent of the RCS files contained only a single revision. The reason is that our systems programmers routinely check in all source files on the distribution tapes, even though they may never touch them again. To get a better indication of how much space savings are possible with deltas, all measures with those files that contained 2 or more revisions were recomputed. Only for those files is RCS necessary. As shown in the second line, the average number of revisions for those files is 3.10, with an overhead of 1.34. This means that the extra 2.10 deltas require 34 per cent extra space, or 16 per cent per extra revision. Rochkind(Section B.3 [Rochkind], page 67) measured the space consumed by SCCS, and reported an average of 5 revisions per group and an overhead of 1.37 (or about 9 per cent per extra revision). In a later paper, Glasser (Section B.6 [Glasser], page 68) observed an average of 7 revisions per group in a single, large project, but provided no overhead figure. In his paper on DSEE , Leblang (Section B.5 [Leblang], page 67) reported that delta storage combined with blank compression results in an overhead of a mere 1-2 per cent per revision. Since leading blanks accounted for about 20 per cent of the surveyed Pascal programs, a revision group with 5-10 members was smaller than a single cleartext copy.

The above observations demonstrate clearly that the space needed for extra revisions is small. With delta storage, the luxury of keeping multiple revisions online is certainly affordable. In fact, introducing a system with delta storage may reduce storage requirements, because programmers often save back-up copies anyway. Since back-up copies are stored much more efficiently with deltas, introducing a system such as RCS may actually free a considerable amount of space.

2.7 Survey of Version Control Tools

The need to keep back-up copies of software arose when programs and data were no longer stored on paper media, but were entered from terminals and stored on disk. Back-up copies are desirable for reliability, and many modern editors automatically save a back-up copy for every file touched. This strategy is valuable for short-term back-ups, but not suitable for long-term version control, since an existing back-up copy is overwritten whenever the corresponding file is edited.

Tape archives are suitable for long-term, offline storage. If all changed files are dumped on a back-up tape once per day, old revisions remain accessible. However, tape archives are unsatisfactory for version control in several ways. First, backing up the file system every 24 hours does not capture intermediate revisions. Secondly, the old revisions are not online, and accessing them is tedious and time-consuming. In particular, it is impractical to compare several old revisions of a group, because that may require mounting and searching several tapes. Tape archives are important fail-safe tools in the event of catastrophic disk failures or accidental deletions, but they are ill-suited for version control. Conversely, version control tools do not obviate the need for tape archives.

A natural technique for keeping several old revisions online is to never delete a file. Editing a file simply creates a new file with the same name, but with a different sequence number. This technique, available as an option in DEC's VMS operating system, turns out to be inadequate for version control. First, it is prohibitively expensive in terms of storage costs, especially since no data compression techniques are employed. Secondly, indiscriminately storing every change produces too many revisions, and programmers have difficulties distinguishing them. The proliferation of revisions forces programmers to spend much time on finding and deleting useless files. Thirdly, most of the support functions like locking, logging, revision selection, and identification described in this paper are not available.

An alternative approach is to separate editing from revision control. The user may repeatedly edit a given revision, until freezing it with an explicit command. Once a revision is frozen, it is stored permanently and can no longer be modified. (In RCS, freezing a revisions is done with `ci`.) Editing a frozen revision implicitly creates a new one, which can again be changed repeatedly until it is frozen itself. This approach saves exactly those revisions that the user considers important, and keeps the number of revisions manageable. IBM's CLEAR/CASTER (Section B.7 [Brown], page 68), AT&T's SCCS (Section B.3 [Rochkind], page 67), CMU's SDC (Section B.8 [Habermann], page 68), and DEC's CMS (Section B.9 [DEC], page 68), are examples of version control systems using this approach. CLEAR/CASTER maintains a data base of programs, specifications, documentation and messages, using deltas. Its goal is to provide control over the development process from a management viewpoint. SCCS stores multiple revisions of source text in an ancestral tree, records a log entry for each revision, provides access control, and has facilities for uniquely identifying each revision. An efficient delta technique reduces the space consumed by each revision group. SDC is much simpler than SCCS because it stores not more than two revisions. However, it maintains a complete log for all old revisions, some of which may be on back-up tape. CMS, like SCCS, manages tree-structured revision groups, but offers no identification mechanism.

Tools for dealing with configurations are still in a state of flux. SCCS, SDC and CMS can be combined with MAKE or MAKE-like programs. Since flexible selection rules are missing from all these tools, it is sometimes difficult to specify precisely which revision of each group should be passed to MAKE for building a desired configuration. The Xerox Cedar system (Section B.10 [Lampson], page 68) provides a 'System Modeller' that can rebuild a configuration from an arbitrary set of module revisions. The revisions of a module are only distinguished by creation time, and there is no tool for managing groups. Since the selection rules are primitive, the System Modeller appears to be somewhat tedious to use. Apollo's DSEE (Section B.5 [Leblang], page 67) is a sophisticated software engineering environment. It manages revision groups in a way similar to SCCS and CMS. Configurations are built using 'configuration threads'. A configuration thread states which revision of each group named in a configuration should be chosen. A configuration thread may contain dynamic specifiers (e.g., 'choose the revisions I am currently working on, and the most recent revisions otherwise'), which are bound automatically at build time. It also provides a notification mechanism for alerting maintainers about the need to rebuild a system after a change.

RCS is based on a general model for describing multi-version/multi-configuration systems (Section B.11 [Tichy1], page 68). The model describes systems using AND/OR graphs, where AND nodes represent configurations, and OR nodes represent version groups. The model gives rise to a suit of selection rules for composing configurations, almost all of which are implemented in RCS. The revisions selected by RCS are passed to MAKE for configuration building. Revision group management is modelled after SCCS. RCS retains SCCS's best features, but offers a significantly simpler user interface, flexible selection rules, adequate integration with MAKE and improved identification. A detailed comparison of RCS and SCCS appears in Reference 4.

An important component of all revision control systems is a program for computing deltas. SCCS and RCS use the program `diff` (Section B.3 [Rochkind], page 67), which first computes the longest common substring of two revisions, and then produces the delta from that substring. The delta is simply an edit script consisting of deletion and insertion commands that generate one revision from the other.

A delta based on a longest common substring is not necessarily minimal, because it does not take advantage of crossing block moves. Crossing block moves arise if two or more blocks of lines (e.g., procedures) appear in a different order in two revisions. An edit script derived from a longest common substring first deletes the shorter of the two blocks, and then reinserts it. Heckel (Section B.12 [Heckel], page 69) proposed an algorithm for detecting block moves, but since the algorithm is based on heuristics, there are conditions under which the generated delta is far from minimal. DSEE uses this algorithm combined with blank compression, apparently with satisfactory overall results. A new algorithm that is guaranteed to produce a minimal delta based on block moves appears in Reference 13. A future release of RCS will use this algorithm.

```
Acknowledgements:
Many people have helped make RCS a success by contributed
criticisms, suggestions, corrections, and even whole new commands
(including manual pages). The list of people is too long to be
reproduced here, but my sincere thanks for their help and
goodwill goes to all of them.
```

3 Introduction to RCS commands

3.1 Description

The Revision Control System (RCS) manages multiple revisions of 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, and form letters.

The basic user interface is extremely simple. The novice only needs to learn two commands: `ci` and `co`. `ci`, short for "check in", deposits the contents of a file into an archival file called an RCS file. An RCS file contains all revisions of a particular file. `co`, short for "check out", retrieves revisions from an RCS file.

3.2 Functions of RCS

- Store and retrieve 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.
- Maintain 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.
- Resolve 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.
- Maintain 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.
- Merge revisions and resolve 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.
- Control releases and configurations. 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.
- Automatically identify 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.
- Minimize secondary storage. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

3.3 Getting Started with RCS

Suppose you have a file ‘f.c’ that you wish to put under control of RCS. If you have not already done so, make an RCS directory with the command

```
mkdir  RCS
```

Then invoke the check-in command

```
ci  f.c
```

This command creates an RCS file in the RCS directory, 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 in the RCS directory are called RCS files; 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 the RCS file and writes it into ‘f.c’. If you want to edit ‘f.c’, you must lock it as you check it out with the command

```
co  -l  f.c
```

You can now edit ‘f.c.’

Suppose after some editing you want to know what changes that you have made. The command

```
rcsdiff  f.c
```

tells you the difference between the most recently checked-in version and the working file. You can check the file back in by invoking

```
ci  f.c
```

This increments the revision number properly.

If ci complains with the message

```
ci error: no lock set by your name
```

then you have tried to check in a file even though you did not lock it when you checked it out. Of course, it is too late now to do the check-out with locking, because another 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.

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     and     rcs  -L  f.c
```

If you don't want to clutter your working directory with RCS files, create a subdirectory called RCS in your work- ing directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any modification. (Actually, pairs of RCS and working files can be specified in three ways: (a) both are given, (b) only the working file is given, (c) only the RCS file is given. Both RCS and working files may have arbitrary path prefixes; RCS commands pair them up intelligently.)

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

```
    ci  -l  f.c     or     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 you want to continue editing, the second one if you just want to read the file. Both update the identification markers in your working file (see below).

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     or     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     and     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 two fields. Numbers with more than two 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 Section 7.2.5 [rcsOptb], page 42.

3.4 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:  filename revision date time author state $
```

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[] = "$Id$";
```

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.

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. There are several additional identification markers; see Section 6.3 [coKeyword], page 39 for details.

4 Synopsis of RCS Operations

- ci – check in revisions

  `Ci` stores the contents of a working file into the corresponding RCS file as a new revision. If the RCS file doesn't exist, `ci` creates it. `Ci` removes the working file, unless one of the options `-u` or `-l` is present. For each check-in, `ci` asks for a commentary describing the changes relative to the previous revision.

  `Ci` assigns the revision number given by the `-r` option; if that option is missing, it derives the number from the lock held by the user; if there is no lock and locking is not strict, `ci` increments the number of the latest revision on the trunk. A side branch can only be started by explicitly specifying its number with the `-r` option during check-in.

  `Ci` also determines whether the revision to be checked in is different from the previous one, and asks whether to proceed if not. This facility simplifies check-in operations for large systems, because one need not remember which files were changed.

  The option `-k` searches the checked in file for identification markers containing the attributes revision number, check-in date, author and state, and assigns these to the new revision rather than computing them. This option is useful for software distribution: Recipients of distributed software using RCS should check in updates with the `-k` option. This convention guarantees that revision numbers, check-in dates, etc., are the same at all sites.

- co – check out revisions

  `Co` retrieves revisions according to revision number, date, author and state attributes. It either places the revision into the working file, or prints it on the standard output. `Co` always expands the identification markers.

- ident – extract identification markers

  `Ident` extracts the identification markers expanded by `co` from any file and prints them.

- rcs – change RCS file attributes

  `Rcs`is an administrative operation that changes access lists, locks, unlocks, breaks locks, toggles the strict-locking feature, sets state attributes and symbolic revision numbers, changes the description, and deletes revisions. A revision can only be deleted if it is not the fork of a side branch.

- rcsclean – clean working directory

  `Rcsclean` removes working files that were checked out but never changed.[1]

- rcsdiff – compare revisions

  `Rcsdiff` compares two revisions and prints their difference, using the UNIX tool `diff`. One of the revisions compared may be checked out. This command is useful for finding out about changes.

- rcsfreeze – freeze a configuration

  `Rcsfreeze` assigns the same symbolic revision number to a given revision in all RCS files. This command is useful for accurately recording a configuration.

[1] The `rcsclean` and `rcsfreeze` commands are optional and are not always installed.

- rcsmerge – merge revisions

  `Rcsmerge` merges two revisions, `rev1` and `rev2`, with respect to a common ancestor. A 3-way file comparison determines the segments of lines that are (a) the same in all three revisions, or (b) the same in 2 revisions, or (c) different in all three. For all segments of type (b) where `rev1` is the differing revision, the segment in `rev1` replaces the corresponding segment of `rev2`. Type (c) indicates an overlapping change, is flagged as an error, and requires user intervention to select the correct alternative.
- rlog – read log messages

  `Rlog` prints the log messages and other information in an RCS file.

5 ci – check in RCS revisions

5.1 ci description

`ci` stores new revisions into RCS files. Each pathname matching an RCS suffix is taken to be an RCS file. All others are assumed to be working files containing new revisions. `ci` deposits the contents of each working file into the corresponding RCS file. If only a working file is given, `ci` tries to find the corresponding RCS file in an RCS subdirectory and then in the working file's directory. For more details, See Section 5.3 [ciFileNaming], page 30 below.

For `ci` to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file if non-strict locking is used (see Chapter 7 [rcs], page 42). A lock held by someone else can be broken with the `rcs` command.

Unless the `-f` option is given, `ci` checks whether the revision to be deposited differs from the preceding one. If not, instead of creating a new revision `ci` reverts to the preceding one. To revert, ordinary `ci` removes the working file and any lock; `ci -l` keeps and `ci -u` removes any lock, and then they both generate a new working file much as if `co -l` or `co -u` had been applied to the preceding revision. When reverting, any `-n` and `-s` options apply to the preceding revision.

For each revision deposited, `ci` prompts for a log message. The log message should summarize the change and must be terminated by end-of-file or by a line containing `.` by itself. If several files are checked in `ci` asks whether to reuse the previous log message. If the standard input is not a terminal, `ci` suppresses the prompt and uses the same log message for all files(Option `-m` of `ci`, Section 5.2.12 [ciOptm], page 28).

If the RCS file does not exist, `ci` creates it and deposits the contents of the working file as the initial revision (default number: `1.1`). The access list is initialized to empty. Instead of the log message, `ci` requests descriptive text (Option `-t` of `ci`, Section 5.2.16 [ciOptt], page 29) below).

The number `rev` of the deposited revision can be given by any of the options `-f`, `-i`, `-I`, `-j`, `-k`, `-l`, `-M`, `-q`, `-r` or `-u` `rev` can be symbolic, numeric, or mixed. Symbolic names in `rev` must already be defined; see the Option `-n` (Section 5.2.13 [ciOptn], page 28) and `-N` (Section 5.2.14 [ciOptNu], page 28) options for assigning names during checkin.

If `rev` is `$`, `ci` determines the revision number from keyword values in the working file.

If `rev` begins with a period, then the default branch (normally the trunk) is prepended to it. If `rev` is a branch number followed by a period, then the latest revision on that branch is used.

If `rev` is a revision number, it must be higher than the latest one on the branch to which `rev` belongs, or must start a new branch.

If `rev` is a branch rather than a revision number, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If `rev` indicates a non-existing branch, that branch is created with the initial revision numbered `rev .1.`

If `rev` is omitted, `ci` tries to derive the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are `1.`.

If `rev` is omitted and the caller has no lock, but owns the file and locking is not set to `strict`, then the revision is appended to the default branch (normally the trunk; see the option `-b` of `rcs` Section 7.2.5 [rcsOptb], page 42).

Exception: On the trunk, revisions can be appended to the end, but not inserted.

5.2 Command line options of ci

5.2.1 Check in revision

`-r'rev'`

Check in revision **rev**.

`-r`

The bare `-r` option (without any revision) has an unusual meaning in `ci`. With other RCS commands, a bare `-r` option specifies the most recent revision on the default branch, but with `ci`, a bare `-r` option reestablishes the default behavior of releasing a lock and removing the working file, and is used to override any default `-l` or `-u` options established by shell aliases or scripts.

5.2.2 lock a revision

`-l['rev']`

works like `-r`, except it performs an additional `co-l` for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

5.2.3 unlock a revision

`-u['rev']`

works like `-l`, except that the deposited revision is not locked. This lets one read the working file immediately after checkin.

The `-l`, bare `-r`, and `-u` options are mutually exclusive and silently override each other. For example, `ci -u -r` is equivalent to `ci -r` because bare `-r` overrides `-u`.

5.2.4 force a deposit

`-f['rev']`
: forces a deposit; the new revision is deposited even it is not different from the preceding one.

5.2.5 keyword values

`-k['rev']`
: searches the working file for keyword values to determine its revision number, creation date, state, and author (see Chapter 6 [CheckOut], page 34), and assigns these values to the deposited revision, rather than computing them locally. It also generates a default login message noting the login of the caller and the actual checkin date. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the `-k` option at these sites to preserve the original number, date, author, and state. The extracted keyword values and the default log message can be overridden with the options `-d`, `-m`, `-s`, `-w`, and any option that carries a revision number.

5.2.6 quiet

`-q['rev']`
: quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless `-f` is given.

5.2.7 initial check in

`-i['rev']`
: initial checkin; report an error if the RCS file already exists. This avoids race conditions in certain applications.

5.2.8 just check in

`-j['rev']`
: just checkin and do not initialize; report an error if the RCS file does not already exist.

5.2.9 interactive mode

`-I['rev']`
: interactive mode; the user is prompted and questioned even if the standard input is not a terminal.

5.2.10 date

`-d[‘date’]`

uses `date` for the checkin date and time. The `date` is specified in free format as explained in `co` (See Chapter 6 [CheckOut], page 34). This is useful for lying about the checkin date, and for `-k` if no date is available. If `date` is empty, the working file’s time of last modification is used.

5.2.11 modification time

`-M[‘rev’]`

Set the modification time on any new working file to be the date of the retrieved revision. For example, `ci -d -M -u f` does not alter ‘`f`’ ’s modification time, even if ‘`f`’’s contents change due to keyword substitution. Use this option with care; it can confuse `MAKE`.

5.2.12 log message

`-m[‘msg’]`

uses the string `msg` as the log message for all revisions checked in. By convention, log messages that start with `#` are comments and are ignored by programs like GNU Emacs’s `vc` package. Also, log messages that start with `{ clumpname }` (followed by white space) are meant to be clumped together if possible, even if they are associated with different files; the `{ clumpname }` label is used only for clumping, and is not considered to be part of the log message itself.

5.2.13 symbolic name

`-n[‘name’]`

assigns the symbolic name `name` to the number of the checked-in revision. `ci` prints an error message if `name` is already assigned to another number.

5.2.14 replace symbolic name

`-N[‘name’]`

same as `-n`, except that it overrides a previous assignment of `name`.

5.2.15 states

`-s[‘state’]`

sets the state of the checked-in revision to the identifier `state`. The default state is `Exp`.

5.2.16 description

`-t‘file’`
: writes descriptive text from the contents of the named ‘`file`’ into the RCS file, deleting the existing text. The ‘`file`’ cannot begin with `-`.

`-t-‘string’`
: Write descriptive text from the ‘`string`’ into the RCS file, deleting the existing text.

  The `-t` option, in both its forms, has effect only during an initial checkin; it is silently ignored otherwise.

  During the initial checkin, if `-t` is not given, `ci` obtains the text from standard input, terminated by end-of-file or by a line containing `.` by itself. The user is prompted for the text if interaction is possible; see optiom `-I` of `ci`(See Section 5.2.9 [ciOptIu], page 27).

  For backward compatibility with older versions of RCS, a bare `-t` option is ignored.

5.2.17 modification time

`-T`
: Set the RCS file’s modification time to the new revision’s time if the former precedes the latter and there is a new revision; preserve the RCS file’s modification time otherwise. If you have locked a revision, `ci` usually updates the RCs file’s modification time to the current time, because the lock is stored in the RCS file and removing the lock requires changing the RCS file. This can create an RCS file newer than the working file in one of two ways: first, `ci -M` can create a working file with a date before the current time; second, when reverting to the previous revision the RCS file can change while the working file remains unchanged. These two cases can cause excessive recompilation caused by a `MAKE` dependency of the working file on the RCS file. The `-T` option inhibits this recompilation by lying about the RCS file’s date. Use this option with care; it can suppress recompilation even when a checkin of one working file should affect another working file associated with the same RCS file. For example, suppose the RCS file’s time is 01:00, the (changed) working file’s time is 02:00, some other copy of the working file has a time of 03:00, and the current time is 04:00. Then `ci -d -T` sets the RCS file’s time to 02:00 instead of the usual 04:00; this causes `MAKE` to think (incorrectly) that the other copy is newer than the RCs file.

5.2.18 login

`-w‘login’`
: uses `login` for the author field of the deposited revision. Useful for lying about the author, and for `-k` if no author is available.

5.2.19 version and emulation

`-V`

Print RCS's version number.

`-V‘n’`

Emulate RCS version `n`. See Section 6.2.15 [coOptV], page 38

5.2.20 suffixes

`-x‘suffixes’`

specifies the suffixes for RCS files. A nonempty suffix matches any pathname ending in the suffix. An empty suffix matches any pathname of the form `RCS/`**path** or `path1/RCS/path2`. The `-x` option can specify a list of suffixes separated by . For example, `-x,v/` specifies two suffixes: `,v` and the empty suffix. If two or more suffixes are specified, they are tried in order when looking for an RCS file; the first one that works is used for that file. If no RCS file is found but an RCS file can be created, the suffixes are tried in order to determine the new RCS file's name. The default for `suffixes` is installation-dependent; normally it is `,v/` for hosts like Unix that permit commas in filenames, and is empty (i.e. just the empty suffix) for other hosts.

5.2.21 time zoone

`-z‘zoone’`

specifies the date output format in keyword substitution, and specifies the default time zone for `date` in the `-d‘date’` option. The ‘`zoone`’ should be empty, a numeric UTC offset, or the special string ‘`LT`’ for local time. The default is an empty ‘`zoone`’, which uses the traditional RCS format of UTC without any time zone indication and with slashes separating the parts of the date; otherwise, times are output in ISO 8601 format with time zone indication. For example, if local time is January 11, 1990, 8pm Pacific Standard Time, eight hours west of UTC, then the time is output as follows:

```
Option   time output
-z       1990/01/11 04:00:00       (default)
-zLT     1990-01-11 20:00:00-0800
-z+0530  1990-01-11 09:30:00+0530
```

This option does not affect dates stored in RCS files, which are always UTC.

5.3 FILE NAMING

Pairs of RCS files and working files can be specified in three ways (see also the example section).

1. Both the file and the working file are given. The RCS pathname is of the form `path1/workfileX` and the working pathname is of the form `path2/workfile` where `path1/` and `path2/` are (possibly different or empty) paths, `workfile` is a filename,

and `X` is an RCs suffix. If `X` is empty, `path1/` must start with `RCS/` or must contain `/RCS/`.

2. Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing `path1/` and the suffix `X`.
3. Only the working file is given. Then `ci` considers each RCS suffix `X` in turn, looking for an RCS file of the form `path2/RCS/workfileX` or (if the former is not found and `X` is nonempty) `path2/workfileX`.

If the RCS file is specified without a path in 1) and 2), `ci` looks for the RCS file first in the directory `./RCS` and then in the current directory.

`ci` reports an error if an attempt to open an RCS file fails for an unusual reason, even if the RCS file's pathname is just one of several possibilities. For example, to suppress use of RCS commands in a directory `d` , create a regular file named `d/RCS` so that casual attempts to use RCS commands in `d` fail because `d/RCS` is not a directory.

5.4 Examples

Suppose `,v` is an RCS suffix and the current directory contains a subdirectory `RCS` with an RCS file '`io.c,v`'. Then each of the following commands check in a copy of '`io.c`' into '`RCS/io.c,v`' as the latest revision, removing '`io.c`'.

```
ci  io.c
ci  RCS/io.c,v
ci  io.c,v
ci  io.c RCS/io.c,v
ci  io.c io.c,v
ci  RCS/io.c,v  io.c
ci  io.c,v  io.c
```

Suppose instead that the empty suffix is an RCS suffix and the current directory contains a subdirectory `RCS` with an RCS file `io.c`. The each of the following commands checks in a new revision.

```
ci  io.c
ci  RCS/io.c
ci  io.c  RCS/io.c
ci  RCS/io.c  io.c
```

5.5 File Modes

An RCS file created by `ci` inherits the read and execute permissions from the working file. If the RCS file exists already, `ci` preserves its read and execute permissions. `ci` always turns off all write permissions of RCS files.

5.6 Files

Temporary files are created in the directory containing the working file, and also in the temporary directory (See Section 5.8 [ciEnv], page 33 under Section 6.6 [coEnv], page 41). A semaphore file or files are created in the directory containing the RCS file. With a nonempty suffix, the semaphore names begin with the first character of the suffix; therefore, do not specify an suffix whose first character could be that of a working filename. With an empty suffix, the semaphore names end with `_` so working filenames should not end in `_`

`ci` never changes an RCS or working file. Normally, `ci` unlinks the file and creates a new one; but instead of breaking a chain of one or more symbolic links to an RCS file, it unlinks the destination file instead. Therefore, `ci` breaks any hard or symbolic links to any working file it changes; and hard links to RCS files are ineffective, but symbolic links to RCS files are preserved.

The effective user must be able to search and write the directory containing the RCS file. Normally, the real user must be able to read the RCS and working files and to search and write the directory containing the working file; however, some older hosts cannot easily switch between real and effective users, so on these hosts the effective user is used for all accesses. The effective user is the same as the real user unless your copies of `ci` and `co` have setuid privileges. As described in the next section, these privileges yield extra security if the effective user owns all RCS files and directories, and if only the effective user can write RCS directories.

Users can control access to RCS files by setting the permissions of the directory containing the files; only users with write access to the directory can use RCS commands to change its RCS files. For example, in hosts that allow a user to belong to several groups, one can make a group's RCS directories writable to that group only. This approach suffices for informal projects, but it means that any group member can arbitrarily change the group's RCS files, and can even remove them entirely. Hence more formal projects sometimes distinguish between an RCS administrator, who can change the RCS files at will, and other project members, who can check in new revisions but cannot otherwise change the RCS files.

5.7 Setuid use

To prevent anybody but their RCS administrator from deleting revisions, a set of users can employ setuid privileges as follows.

- Check that the host supports RCS setuid use. Consult a trustworthy expert if there are any doubts. It is best if the `seteuid` system call works as described in Posix 1003.1a Draft 5, because RCS can switch back and forth easily between real and effective users, even if the real user is `root`. If not, the second best is if the `setuid` system call supports saved setuid (the `_POSIX_SAVED_IDS` behavior of Posix 1003.1-1990); this fails only if the real or effective user is `root`. If RCS detects any failure in setuid, it quits immediately.
- Choose a user `A` to serve as RCS administrator for the set of users. Only `A` can invoke the `rcs` command on the users' RCS files. `A` should not be `root` or any other user with special powers. Mutually suspicious sets of users should use different administrators.
- Choose a pathname `B` to be a directory of files to be executed by the users.
- Have `A` set up `B` to contain copies of `ci` and `co` that are setuid to `A` by copying the commands from their standard installation directory `D` as follows:

```
mkdir 'B'
cp 'D'/c[io] 'B'
chmod go-w,u+s 'B'/c[io]
```

- Have each user prepend `B` to their path as follows:

  ```
  PATH='B':$PATH; export PATH # ordinary shell
  set path=('B' $path)        # C - shell
  ```

- Have `A` create each RCS directory `R` with write access only to `A` as follows:

  ```
  mkdir 'R'
  chmod go-w 'R'
  ```

- If you want to let only certain users read the RCS files, put the users into a group `G`, and have `A` further protect the RCs directory as follows:

  ```
  chgrp 'G'  'R'
  chmod g-w, o-rwx  'R'
  ```

- Have `A` copy old RCS files (if any) into `R`, to ensure that `A` owns them.
- An RCS file's access list limits who can check in and lock revisions. The default access list is empty, which grants checkin access to anyone who can read the RCS file. If you want limit checkin access, have `A` invoke `rcs -a` on the file; see Chapter 7 [rcs], page 42 In particular, `rcs -e -a'A'` limits access to just `A`.
- Have `A` initialize any new RCS files with `rcs -i` before initial checkin, adding the `-a` option if you want to limit checkin access.
- Give setuid privileges only to `ci`, `co`, and `rcsclean`; do not give them to `rcs` or to any other command.
- Do not use other setuid commands to invoke RCS commands; setuid is trickier than you think!

5.8 Environment

`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. Useful `RCSINIT` options include `-q`, `-V`, `-x` and `-z`.

`TMPDIR` Name of the temporary directory. If not set, the environment variables `TMP` and `TEMP` are inspected instead and the first value found is taken; if none of them are set, a host-dependent default is used, typically `/tmp`.

5.9 Diagnostics

For each revision, `ci` prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status is zero if and only if all operations were successful.

6 co – check out RCS revisions

6.1 co description

`co` retrieves a revision from each RCS file and stores it into the corresponding working file.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in Section 5.1 [ciIntro], page 25.

Revisions of an RCS file can be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Checkout with locking fails if the revision to be checked out is currently locked by another user. (A lock can be broken with `rcs` Chapter 7 [rcs], page 42). Checkout with locking also requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. Checkout without locking is not subject to accesslist restrictions, and is not affected by the presence of locks.

A revision is selected by options for revision or branch number, checkin date/time, author, or state. When the selection options are applied in combination, `co` retrieves the latest revision that satisfies all of them. If none of the selection options is specified, `co` retrieves the latest revision on the default branch (normally the trunk, see the `-b` option of Chapter 7 [rcs], page 42). A revision or branch number can be attached to any of the options `-f`, `-I`, `-l`, `-M`, `-p`, `-q`, `-r` or `-u`. The options `-d` (date), `-s`(state), and `-w` (author) retrieve from a single branch, the `selected` branch, which is either specified by one of `-f`, . . ., `-u`, or the default branch.

A `co` command applied to an RCS file with no revisions creates a zero-length working file. `co` always performs keyword substitution (see below).

6.2 Command line options of co

6.2.1 Check out revision

`-r‘rev’`

retrieves the latest revision whose number is less than or equal to ‘`rev`’. If ‘`rev`’ indicates a branch rather than a revision, the latest revision on that branch is retrieved. If ‘`rev`’ is omitted, the latest revision on the default branch (see the `-b` option of Chapter 7 [rcs], page 42) is retrieved. If ‘`rev`’ is `$`, `co` determines the revision number from keyword values in the working file. Otherwise, a revision is composed of one or more numeric or symbolic fields separated by periods. If ‘`rev`’ begins with a period, then the default branch (normally the trunk) is prepended to it. If ‘`rev`’ is a branch number followed by a period, then the latest revision on that branch is used. The numeric equivalent of a symbolic field is specified with the `-n` option of the commands See Section 5.2.13 [ciOptn], page 28 and See Section 7.2.14 [rcsOptn], page 44.

6.2.2 Lock revision

`-l‘rev’`

same as `-r`, except that it also locks the retrieved revision for the caller.

6.2.3 Unlock revision

`-l‘rev’`

same as `-r`, except that it unlocks the retrieved revision if it was locked by the caller. If ‘`rev`’ is omitted, `-u` retrieves the revision locked by the caller, if there is one; otherwise, it retrieves the latest revision on the default branch.

6.2.4 Force overwriting working file

`-f‘rev’`

forces the overwriting of the working file; useful in connection with `-q`. See Section 6.4 [coFileModes], page 40

6.2.5 Keyword strings

`-kkv`

Generate keyword strings using the default form, e.g. `$Revision : 5.12 $` for the `Revision` keyword. A locker’s name is inserted in the value of the `Header`, `Id`, and `Locker` keyword strings only as a file is being locked, i.e. by `ci -l` and `co -l`. This is the default.

`-kkvl`

Like `-kkv`, except that a locker’s name is always inserted if the given revision is currently locked.

`-kk`

Generate only keyword names in keyword strings; omit their values. See Section 6.3 [coKeyword], page 39 below. For example, for the `Revision` keyword, generate the string `$Revsion$` instead of `$Revision: 5.12 $`. This option is useful to ignore differences due to keyword substitution when comparing different revisions of a file. Log messages are inserted after `$Log$` keywords even if `-kk` is specified, since this tends to be more useful when merging changes.

`-ko`

Generate the old keyword string, present in the working file just before it was checked in. For example, for the `Revision` keyword, generate the string `$Revision: 1.1 $` instead of `$Revision: 5.12 $` if that is how the string appeared when the file was checked in. This can be useful for file formats that cannot tolerate any changes to substrings that happen to take the form of keyword strings.

`-kb`

Generate a binary image of the old keyword string. This acts like `-ko`, except it performs all working file input and output in binary mode. This makes little difference on Posix and Unix hosts, but on DOS-like hosts one should use `rcs -i -kb` to initialize an RCS file intended to be used for binary files. Also, on all hosts, Chapter 11 [rcsmerge], page 55 normally refuses to merge files when `-kb` is in effect.

`-kv`

Generate only keyword values for keyword strings. For example, for the `Revision` keyword, generate the string `5.12` instead of `$Revision: 5.12 $` This can help generate files in programming languages where it is hard to strip keyword delimiters like `$Revision: $` from a string. However, further keyword substitution cannot be performed once the keyword names are removed, so this option should be used with care. Because of this danger of losing keywords, this option cannot be combined with `-l`, and the owner write permission of the working file is turned off; to edit the file later, check it out again without `-kv`.

6.2.6 Print on standard output

`-p‘rev’`

prints the retrieved revision on the standard output rather than storing it in the working file. This option is useful when `co` is part of a pipe.

6.2.7 Quiet mode

`-q‘rev’`

quiet mode; diagnostics are not printed.

6.2.8 Interactive Mode

`-I‘rev’`

interactive mode; the user is prompted and questioned even if the standard input is not a terminal.

6.2.9 Checkout by date/time

`-d‘date’`

retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to ‘`date`’. The date and time can be given in free format. The time zone `LT` stands for local time; other common time zone names are understood. For example, the following ‘`date`’s are equivalent if local time is January 11, 1990, 8pm Pacific Standard Time, eight hours west of Coordinated Universal Time (UTC):

```
8:00 pm lt
4:00 AM, Jan. 12, 1990            default is UTC
1990-01-12 04:00:00+00            ISO 8601 (UTC)
1990-01-11 20:00:00-08            ISO 8601 (local time)
1990/01/12 04:00:00               traditional RCS format
Thu Jan 11 20:00:00 1990 LT       output of ctime(3) + LT
Thu Jan 11 20:00:00 PST 1990      output of date(1)
Fri Jan 12 04:00:00 GMT 1990
Thu, 11 Jan 1990 20:00:00 -0800   Internet RFC 822
12-January-1990, 04:00 WET
```

Most fields in the date and time can be defaulted. The default time zone is normally UTC, but this can be overridden by the `-z` option. The other defaults are determined in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the time zone's current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, without `-z`, the date `20, 10:30` defaults to 10:30:00 UTC of the 20th of the UTC time zone's current month and year. The date/time must be quoted if it contains spaces.

6.2.10 Modification time

`-M‘rev’`

Set the modification time on the new working file to be the date of the retrieved revision. Use this option with care; it can confuse `make`.

6.2.11 State

`-s‘state’`

retrieves the latest revision on the selected branch whose state is set to ‘`state`’.

6.2.12 Preserve Modification time

`-T`

Preserve the modification time on the RCS file even if the RCS file changes because a lock is added or removed. This option can suppress extensive recompilation caused by a `make` dependency of some other copy of the working file on the RCS file. Use this option with care; it can suppress recompilation even when it is needed, i.e. when the change of lock would mean a change to keyword strings in the other working file.

6.2.13 Checked in by user

`-w‘login’`
: retrieves the latest revision on the selected branch which was checked in by the user with login name ‘`login`’. If the argument ‘`login`’ is omitted, the caller’s login is assumed.

6.2.14 Joining revisions

`-j‘joinlist’`
: generates a new revision which is the join of the revisions on ‘`joinlist`’. This option is largely obsoleted by `rcsmerge` (Chapter 11 [rcsmerge], page 55) but is retained for backwards compatibility.

  The ‘`joinlist`’ is a comma-separated list of pairs of the form `rev2 : rev3`, where `rev2` and `rev3` are (symbolic or numeric) revision numbers. For the initial such pair, `rev1` denotes the revision selected by the above options `-f`, . . ., `-w`. For all other pairs, `rev1` denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

  For each pair, `co` joins revisions `rev1` and `rev3` with respect to `rev2`. This means that all changes that transform `rev2` into `rev1` are applied to a copy of `rev3`. This is particularly useful if `rev1` and `rev3` are the ends of two branches that have `rev2` as a common ancestor. If `rev1 < rev2 < rev3` on the same branch, joining generates a new revision which is like `rev3`, but with all changes that lead from `rev1` to `rev2` undone. If changes from `rev2` to `rev1` overlap with changes from `rev2` to `rev3`, `co` reports overlaps as described in `merge`.

  For the initial pair, `rev2` can be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. The options `-l` and `-u` lock or unlock `rev1`.

6.2.15 Version, Emulate RCS Version

`-V`
: Print RCS’s version number.

`-V‘n’`
: Emulate RCS version ‘`n`’, where ‘`n`’ can be `3`, `4`, or `5`. This can be useful when interchanging RCS files with others who are running older versions of RCS. To see which version of RCS your correspondents are running, have them invoke `rcs -V`; this works with newer versions of RCS. If it doesn’t work, have them invoke `rlog` (Chapter 12 [rlog], page 58) on an RCS file; if none of the first few lines of output contain the string `branch:` it is version 3; if the dates’ years have just two digits, it is version 4; otherwise, it is version 5. An RCS file generated while emulating version 3 loses its default branch. An RCS revision generated while emulating version 4 or earlier has a time stamp that is off by up to 13 hours. A revision extracted while emulating version 4 or earlier contains abbreviated dates of the form `yy/mm/dd` and can also contain different white space and line prefixes in the substitution for Log.

6.2.16 Suffixes

`-x‘suffixes’`
: Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30 for details.

6.2.17 Time Zoone

`-z‘zoone’`
: See Section 5.2.21 [ciOptz], page 30 for details.

6.3 Keyword Substitution

Strings of the form `$keyword$` and `$keyword : ...$` embedded in the text are replaced with strings of the form `$keyword:value$` where `keyword` and `value` are pairs listed below. Keywords can be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form `$keyword$`. On checkout, `co` replaces these strings with strings of the form `$keyword:value$`. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout. This automatic substitution can be modified by the `-k` options.

Keywords and their corresponding values:

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

`$Date$`
: The date and time the revision was checked in. With `-z‘zone’` a numeric time zone offset is appended; otherwise, the date is UTC.

`$Header$`
: A standard header containing the full pathname of the RCS file, the revision number, the date and time, the author, the state, and the locker (if locked). With `-z‘zone’` a numeric time zone offset is appended to the date; otherwise, the date is RCS.

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

`$Locker$`
: The login name of the user who locked the revision (empty if not locked).

`$Log$`
: The log message supplied during checkin, preceded by a header containing the RCS filename, the revision number, the author, and the date and time. With `-z‘zone’` a numeric time zone offset is appended; otherwise, the date is UTC. 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.

Each inserted line is prefixed by the string that prefixes the `$Log$` line. For example, if the `$Log$` line is `"// $Log: tan.cc $"`, RCS prefixes each line of the log with `"// "`. This is useful for languages with comments that go to the end of the line. The convention for other languages is to use a `" * "` prefix inside a multiline comment. For example, the initial log comment of a C program conventionally is of the following form:

```
/*
 * $Log$
 */
```

For backwards compatibility with older versions of RCS, if the log prefix is `/*` or `(*` surrounded by optional white space, inserted log lines contain a space instead of `/` or `(`; however, this usage is obsolescent and should not be relied on.

`$Name$`

The symbolic name used to check out the revision, if any. For example, `"co -rJoe"` generates `"$Name: Joe $"`. Plain `co` generates just `"$Name: $"`

`$RCSFile$`

The name of the RCS file without a path.

`$Revsion$`

The revision number assigned to the revision.

`$Source$`

The full pathname of the RCS file.

`$State$`

The state assigned to the revision with the `-s` option of `rcs` or `ci`.

The following characters in keyword values are represented by escape sequences to keep keyword strings well-formed.

char	escape sequence
tab	\t
newline	\n
space	\040
$	\044
\	\\

6.4 File Modes

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permis- sion is turned on, unless `-kv` is set or the file is checked out unlocked and locking is set to strict (See Chapter 7 [rcs], page 42).

If a file with the name of the working file exists already and has write permission, `co` aborts the checkout, asking beforehand if possible. If the existing working file is not writable or `-f` is given, the working file is deleted without asking.

6.5 Files

`co` accesses files much as `ci` does, except that it does not need to read the working file unless a revision number of `$` is specified.

6.6 Environment

`RCSINIT`	options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

6.7 Diagnostics

The RCS pathname, the working pathname, and the revision number retrieved are written to the diagnostic output. The exit status is zero if and only if all operations were successful.

6.8 Limits

Links to the RCS and working files are not preserved.

7 rcs – change RCS file attributes

7.1 Description

RCS creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For rcs to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the `-i` option is present.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in See Section 5.1 [ciIntro], page 25. Revision numbers use the syntax described in See Section 5.1 [ciIntro], page 25.

7.2 Command line options of rcs

7.2.1 Initialize RCS File

`-i`

Create and initialize a new RCS file, but do not deposit any revision. If the RCS file has no path prefix, try to place it first into the subdirectory `./RCS`, and then into the current directory. If the RCS file already exists, print an error message.

7.2.2 Append Login names

`-a‘logins’`

Append the login names appearing in the comma-separated list ‘`logins`’ to the access list of the RCS file.

7.2.3 Append Access list

`-A‘oldfile’`

Append the access list of ‘`oldfile`’ to the access list of the RCS file.

7.2.4 Erase Login names

`-e‘logins’`

Erase the login names appearing in the comma- separated list ‘`logins`’ from the access list of the RCS file. If ‘`logins`’ is omitted, erase the entire access list.

7.2.5 Default branch

`-b‘rev’`

Set the default branch to ‘`rev`’. If ‘`rev`’ is omitted, the default branch is reset to the (dynamically) highest branch on the trunk.

7.2.6 Coment leader

`-c‘string’`
: Set the comment leader to ‘`string`’. An initial `ci`, or an `rcs -i` without `-c`, guesses the comment leader from the suffix of the working filename.

  This option is obsolescent, since RCS normally uses the preceding `$Log$` line’s prefix when inserting log lines during checkout (See Section 6.1 [coIntro], page 34). However, older versions of RCS use the comment leader instead of the `$Log$` line’s prefix, so if you plan to access a file with both old and new versions of RCS, make sure its comment leader matches its `$Log$` line prefix.

7.2.7 Keyword substitution

`-k‘subst’`
: Set the default keyword substitution to ‘`subst`’. The effect of keyword substitution is described in Section 6.3 [coKeyword], page 39. Giving an explicit `-k` option to co, `rcsdiff`, and `rcsmerge` overrides this default. Beware `rcs -kv`, because `-kv` is incompatible with `co -l`. Use `rcs -kkv` to restore the normal default keyword substitution.

7.2.8 Lock revision

`-l‘rev’`
: Lock the revision with number ‘`rev`’. If a branch is given, lock the latest revision on that branch. If ‘`rev`’ is omitted, lock the latest revision on the default branch. Locking prevents overlapping changes. If someone else already holds the lock, the lock is broken as with `rcs -u` (see below).

7.2.9 Unlock revisions

`-u‘rev’`
: Unlock the revision with number ‘`rev`’. If a branch is given, unlock the latest revision on that branch. If ‘`rev`’ is omitted, remove the latest lock held by the caller. Normally, only the locker of a revision can unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated by end-of-file or by a line containing `.` by itself.

7.2.10 Strict Locking

`-L`
: Set locking to `strict`. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

7.2.11 Locking non-strict

`-U`

Set locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should `not` be used for files that are shared. Whether default locking is strict is determined by your system administrator, but it is normally strict.

7.2.12 Replace log message

`-m‘rev’:‘msg’`
Replace revision ‘`rev`’’s log message with ‘`msg`’.

7.2.13 Do not send Mail

`-M`

Do not send mail when breaking somebody else’s lock. This option is not meant for casual use; it is meant for programs that warn users by other means, and invoke `rcs -u` only as a low-level lock-breaking operation.

7.2.14 Symbolic name

`-n‘name’[:[‘rev’]]`
Associate the symbolic name ‘`name`’ with the branch or revision ‘`rev`’. Delete the symbolic name if both: and ‘`rev`’ are omitted; otherwise, print an error message if ‘`name`’ is already associated with another number. If ‘`rev`’ is symbolic, it is expanded before association. A ‘`rev`’ consisting of a branch number followed by a . stands for the current latest revision in the branch. A : with an empty ‘`rev`’ stands for the current latest revision on the default branch, normally the trunk. For example,

```
rcs -nname: RCS/*
```

associates `name` with the current latest revision of all the named RCS files; this contrasts with

```
rcs -nname:$ RCS/*
```

which associates `name` with the revision numbers extracted from keyword strings in the corresponding working files.

7.2.15 Override Symbolic name

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

7.2.16 Delete revisions

`-o‘range’`

deletes ("outdates") the revisions given by ‘`range`’. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form `‘rev1’:‘rev2’` means revisions `rev1` to `rev2` on the same branch, `:‘rev’` means from the beginning of the branch containing ‘`rev`’ up to and including ‘`rev`’, and `rev:` means from revision `rev` to the end of the branch containing `rev` . None of the outdated revisions can have branches or locks.

7.2.17 Quiet mode

`-q`

Run quietly; do not print diagnostics.

7.2.18 Interactive

`-I`

Run interactively, even if the standard input is not a terminal.

7.2.19 States

`-s‘state’:[‘rev’]`

Set the state attribute of the revision ‘`rev`’ to ‘`state`’. If ‘`rev`’ is a branch number, assume the latest revision on that branch. If ‘`rev`’ is omitted, assume the latest revision on the default branch. Any identifier is acceptable for ‘`state`’ . A useful set of states is Exp (for experimental), Stab (for stable), and Rel (for released). By default, Section 5.2.15 [ciOpts], page 28 sets the state of a revision to Exp.

7.2.20 Descriptive Text

`-t‘file’`

Write descriptive text from the contents of the named ‘`file`’ into the RCS file, deleting the existing text. The ‘`file`’ pathname cannot begin with `-`. If ‘`file`’ is omitted, obtain the text from standard input, terminated by end-of-file or by a line containing. by itself. Prompt for the text if interaction is possible; see `-I` Section 7.2.18 [rcsOptIu], page 45. With `-i`, descriptive text is obtained even if `-t` is not given.

`-t-‘string’`

Write descriptive text from the ‘`string`’ into the RCS file, deleting the existing text.

7.2.21 Modification Time

`-T`

Preserve the modification time on the RCS file unless a revision is removed. This option can sup- press extensive recompilation caused by a `make` dependency of some copy of the working file on the RCS file. Use this option with care; it can suppress recompilation even when it is needed, i.e. when a change to the RCS file would mean a change to keyword strings in the working file.

7.2.22 Version

`-V`

Print RCS's version number.

`-V‘n’`

Emulate RCS version ‘`n`’. See Section 6.2.15 [coOptV], page 38 for details.

7.2.23 Suffixes

`-x‘suffixes’`

Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30.

7.2.24 Time Zone

`-z‘zone’`

Use ‘`zone`’ as the default time zone. This option has no effect; it is present for compatibility with other RCS commands.

7.2.25 Compatibility

The `-b‘rev’` option generates an RCS file that cannot be parsed by RCS version 3 or earlier.

The `-k‘subst’` options (except `-kkv`) generate an RCS file that cannot be parsed by RCS version 4 or earlier.

Use rcs `-V‘n’` to make an RCS file acceptable to RCS version ‘`n`’ by discarding information that would confuse version ‘`n`’.

RCS version 5.5 and earlier does not support the `-x` option, and requires a `,v` suffix on an RCS pathname.

7.2.26 Files

rcs accesses files much as `ci` does, except that it uses the effective user for all accesses, it does not write the working file or its directory, and it does not even read

7.3 Environment

`RCSINIT`	options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

7.4 Diagnostics

The RCS pathname and the revisions outdated are written to the diagnostic output. The exit status is zero if and only if all operations were successful.

7.5 Bugs

A catastrophe (e.g. a system crash) can cause RCS to leave behind a semaphore file that causes later invocations of RCS to claim that the RCS file is in use. To fix this, remove the semaphore file. A semaphore file's name typically begins with '`,`' or ends with '`_`'.

The separator for revision ranges in the `-o` option used to be '`-`' instead of '`:`', but this leads to confusion when symbolic names contain '`-`'. For backwards compatibility `rcs -o` still supports the old - separator, but it warns about this obsolete use.

Symbolic names need not refer to existing revisions or branches. For example, the `-o` option does not remove symbolic names for the outdated revisions; you must use `-n` to remove the names.

8 ident – identify RCS keywords

8.1 Description

Ident searches for all instances of the pattern `$keyword: text $` in the named files or, if no files are named, the standard input.

These patterns are normally inserted automatically by the RCS command `co` (Section 6.3 [coKeyword], page 39), but can also be inserted manually. The option `-q`(See Section 8.2.1 [identOptq], page 48) suppresses the warning given if there are no patterns in a file. The option `-V`(See Section 8.2.2 [identOptV], page 48) prints ident's version number.

ident works on text files as well as object files and dumps. For example, if the C program in f.c contains:

```
#include <stdio.h>
static char const rcsid[] =
    "$Id: f.c,v 5.4 1993/11/09  17:40:15  eggert  Exp $";
int main() { return printf("%s\n", rcsid) == EOF; }
```

and f.c is compiled into f.o, then the command

```
ident  f.c  f.o
```

will output

```
f.c:
    $Id: f.c,v 5.4 1993/11/09 17:40:15 eggert Exp $
f.o:
    $Id: f.c,v 5.4 1993/11/09 17:40:15 eggert Exp $
```

If a C program defines a string like rcsid above but does not use it, `lint` may complain, and some C compilers will optimize away the string. The most reliable solution is to have the program use the rcsid string, as shown in the example above.

ident finds all instances of the `$keyword: text $` pattern, even if `keyword` is not actually an RCS-supported keyword. This gives you information about nonstandard keywords like `$XConsortium$`.

8.2 Command line options of ident

8.2.1 Quiet mode

`-q`

Suppress warning given if there are no patterns in a file.

8.2.2 Version

`-V`

Print ident's version number.

8.3 Keywords

The maintained list of keywords of `co` you'll find in Section 6.3 [coKeyword], page 39.

All times are given in Coordinated Universal Time (UTC, sometimes called GMT) by default, but if the files were checked out with `co`'s `-z‘zone’` (See Section 6.2.17 [coOptz], page 39) option, times are given with a numeric time zone indication appended.

9 rcsclean – clean up working files

9.1 Description

`rcsclean` removes files that are not being worked on. `rcsclean -u` (Section 9.2.6 [rcscleanOptu], page 51) also unlocks and removes files that are being worked on but have not changed.

For each '`file`' given, rcsclean compares the working file and a revision in the corresponding RCS file. If it finds a difference, it does nothing. Otherwise, it first unlocks the revision if the `-u` (See Section 9.2.6 [rcscleanOptu], page 51 option is given, and then removes the working file unless the working file is writable and the revision is locked. It logs its actions by outputting the corresponding `rcs -u` (See Section 7.2.9 [rcsOptu], page 43) and `rm -f` commands on the standard output.

Files are paired as explained in Section 5.6 [ciFiles], page 31. If no '`file`' is given, all working files in the current directory are cleaned. Pathnames matching an RCS suffix denote RCS files; all others denote working files.

The number of the revision to which the working file is compared may be attached to any of the options `-n`, `-q`, `-r`, or `-u`. If no revision number is specified, then if the `-u` option is given and the caller has one revision locked, rcsclean uses that revision; otherwise rcsclean uses the latest revision on the default branch, normally the root.

`rcsclean` is useful for clean targets in makefiles. See Chapter 10 [rcsdiff], page 53, which prints out the differences, and `ci` (Section 5.1 [ciIntro], page 25), which normally reverts to the previous revision if a file was not changed.

9.2 Command line options of rcsclean

9.2.1 Keyword substitution

`-k'subst'`
: Use '`subst`' style keyword substitution when retrieving the revision for comparison. See Section 6.3 [coKeyword], page 39 for details.

9.2.2 No removing

`-n'rev'`
: Do not actually remove any files or unlock any revisions. Using this option will tell you what `rcsclean` would do without actually doing it.

9.2.3 Quiet mode

`-q'rev'`
: Do not log the actions taken on standard output.

9.2.4 Revision for comparison

`-r‘rev’`
: This option has no effect other than specifying the revision for comparison.

9.2.5 Modification time

`-T`
: Preserve the modification time on the RCS file even if the RCS file changes because a lock is removed. This option can suppress extensive recompilation caused by a `make` dependency of some other copy of the working file on the RCS file. Use this option with care; it can suppress recompilation even when it is needed, i.e. when the lock removal would mean a change to keyword strings in the other working file.

9.2.6 Unlock revision

`-u‘rev’`
: Unlock the revision if it is locked and no difference is found.

9.2.7 Version and Emulation

`-V`
: Print RCS's version number.

`-V‘n’`
: Emulate RCS version ‘`n`’. See Section 6.2.15 [coOptV], page 38 for details.

9.2.8 Suffixes

`-x‘suffixes’`
: Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30 for details.

9.2.9 Time zone

`-z‘zone’`
: `-z‘zone’` Use ‘`zone`’ as the time zone for keyword substitution; see Section 6.2.17 [coOptz], page 39 for details.

9.3 Examples

```
rcsclean  *.c  *.h
```

removes all working files ending in .c or .h that were not changed since their checkout.

```
rcsclean
```

removes all working files in the current directory that were not changed since their checkout.

9.4 Files

rcsclean accesses files much as `ci` (Section 5.6 [ciFiles], page 31)does.

9.5 Environment

`RCSINIT` options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

9.6 Diagnostics

The exit status is zero if and only if all operations were successful. Missing working files and RCS files are silently ignored.

9.7 Bugs

At least one '`file`' must be given in older Unix versions that do not provide the needed directory scanning operations.

10 rcsdiff – compare RCS revisions

10.1 Description

rcsdiff runs `diff` to compare two revisions of each RCS file given.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in Section 5.6 [ciFiles], page 31.

10.2 Command line options of rcsdiff

10.2.1 Keyword substitution

`-k‘subst’`

`-k‘subst’` affects keyword substitution when extracting revisions, as described in Section 6.2.5 [coOptk], page 35; for example, `-kk -r1.1 -r1.2` ignores differences in keyword values when comparing revisions `1.1` and `1.2`. To avoid excess output from locker name substitution, `-kkvl` is assumed if

1. at most one revision option is given,
2. no `-k` option is given,
3. `-kkv` is the default keyword substitution, and
4. the working file's mode would be produced by `co -l`.

10.2.2 Quiet mode

`-q‘rev’`

The option -q suppresses diagnostic output.

10.2.3 Revision for comparison

`-r‘rev1’ -r‘rev2’`

Zero, one, or two revisions may be specified with `-r`. If both ‘`rev1`’ and ‘`rev2`’ are omitted, rcsdiff compares the latest revision on the default branch (by default the trunk) with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If ‘`rev1`’ is given, but ‘`rev2`’ is omitted, rcsdiff compares revision ‘`rev1`’ of the RCS file with the contents of the corresponding working file.

If both ‘`rev1`’ and ‘`rev2`’ are given, rcsdiff compares revisions ‘`rev1`’ and ‘`rev2`’ of the RCS file.

Both ‘`rev1`’ and ‘`rev2`’ may be given numerically or symbolically.

10.2.4 Modification time

`-T`

For details See Section 6.2.12 [coOptT], page 37.

10.2.5 Version and Emulation

`-V`
: Print RCS's version number.

`-V‘n’`
: Emulate RCS version ‘`n`’. See Section 6.2.15 [coOptV], page 38 for details.

10.2.6 Suffixes

`-x‘suffixes’`
: Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30 for details.

10.3 Examples

```
rcsdiff  *.c  *.h
```

removes all working files ending in .c or .h that were not changed since their checkout.

```
rcsdiff
```

removes all working files in the current directory that were not changed since their checkout.

10.4 Environment

`RCSINIT` options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

10.5 Diagnostics

Exit status is 0 for no differences during any comparison, 1 for some differences, 2 for trouble.

11 rcsmerge – merge RCS versions

11.1 Description

rcsmerge incorporates the changes between two revisions of an RCS file into the corresponding working file.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in Section 5.6 [ciFiles], page 31.

At least one revision must be specified with one of the options described below, usually `-r`. At most two revisions may be specified. If only one revision is specified, the latest revision on the default branch (normally the highest branch on the trunk) is assumed for the second revision. Revisions may be specified numerically or symbolically.

rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in Chapter 13 [merge], page 62. The command is useful for incorporating changes into a checked-out revision.

11.2 Command line options of rcsmerge

11.2.1 Output conflicts

`-A`
: Output conflicts using the `-A` style of `diff3`, if supported by `diff3`. This merges all changes leading from '`file2`' to '`file3`' into '`file1`', and generates the most verbose output.

11.2.2 Less information

`-E -e`
: These options specify conflict styles that generate less information than `-A`. See `diff3` for details. The default is `-E`. With `-e`, `rcsmerge` does not warn about conflicts.

11.2.3 Keyword substitution

`-k'subst'`
: Use '`subst`' style keyword substitution. See Section 6.3 [coKeyword], page 39 for details. For example, `-kk -r1.1 -r1.2` ignores differences in keyword values when merging the changes from `1.1` to `1.2`. It normally does not make sense to merge binary files as if they were text, so `rcsmerge` refuses to merge files if `-kb` expansion is used.

11.2.4 Send to standard out

`-p['rev']`
: Send the result to standard output instead of over- writing the working file.

11.2.5 Modification Time

`-T`
: This option has no effect; it is present for compatibility with other RCS commands.

11.2.6 Version

`-V`
: Print RCS's version number.

`-V‘n’`
: Emulate RCS version ‘`n`’. See Section 6.2.15 [coOptV], page 38 for details.

11.2.7 Suffixes

`-x‘suffixes’`
: Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30.

11.2.8 Time Zone

`-z‘zone’`
: Use ‘`zone`’ as the time zone for keyword substitution. See Section 6.2.17 [coOptz], page 39 for details.

11.2.9 Examples

Suppose you have released revision `2.8` of ‘`f.c`’. Assume furthermore that after you complete an unreleased revision `3.4`, you receive updates to release `2.8` from someone else. To combine the updates to `2.8` and your changes between `2.8` and `3.4`, put the updates to `2.8` into file ‘`f.c`’ and execute

```
rcsmerge -p -r2.8 -r3.4 f.c >f.merged.c
```

Then examine ‘`f.merged.c`’. Alternatively, if you want to save the updates to `2.8` in the RCS file, check them in as revision `2.8.1.1` and execute `co -j`:

```
ci -r2.8.1.1 f.c
co -r3.4 -j2.8:2.8.1.1 f.c
```

As another example, the following command undoes the changes between revision `2.4` and `2.8` in your currently checked out revision in ‘`f.c`’.

```
rcsmerge -r2.8 -r2.4 f.c
```

Note the order of the arguments, and that ‘`f.c`’ will be overwritten.

11.3 Environment

`RCSINIT`
: options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

11.4 Diagnostics

Exit status is 0 for no overlaps, 1 for some overlaps, 2 for trouble.

12 rlog – print log messages

12.1 Description

`rlog` prints information about RCS files.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in Section 5.6 [ciFiles], page 31.

`rlog` prints the following information for each RCS file: RCS pathname, working pathname, head (i.e., the number of the latest revision on the trunk), default branch, access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for the selected revisions in reverse chronological order for each branch. For each revision, rlog prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. All times are displayed in Coordinated Universal Time (UTC) by default; this can be overridden with `-z`. Without options, `rlog` prints complete information. The options below restrict this output.

12.2 Command line options of rlog

12.2.1 Ignore RCS files without locks

`-L`

Ignore RCS files that have no locks set. This is convenient in combination with `-h`, `-l`, and `-R`.

12.2.2 Only names of RCS files

`-R`

Print only the name of the RCS file. This is convenient for translating a working pathname into an RCS pathname.

12.2.3 Only pathname

`-h`

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

12.2.4 Descriptive text

`-t`

Print the same as `-h`, plus the descriptive text.

12.2.5 No symbolic name

`-N`

Do not print the symbolic names.

12.2.6 Default branch

`-b`

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

12.2.7 Checkin date/time

`-d‘dates’`

Print information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of ‘`dates`’. A range of the form ‘`d1`’`<`‘`d2`’ or ‘`d2`’`>`‘`d1`’ selects the revisions that were deposited between ‘`d1`’ and ‘`d2`’ exclusive. A range of the form `<`‘`d`’ or ‘`d`’`>` selects all revisions earlier than ‘`d`’. A range of the form ‘`d`’`<` or `>`‘`d`’ selects all revisions dated later than ‘`d`’. If `<` or `>` is followed by `=` then the ranges are inclusive, not exclusive. A range of the form ‘`d`’ selects the single, latest revision dated ‘`d`’ or earlier. The date/time strings ‘`d`’, ‘`d1`’, and ‘`d2`’ are in the free format explained in Section 6.2.9 [coOptd], page 36. Quoting is normally necessary, especially for `<` and `>`. Note that the separator is a semicolon.

12.2.8 Locked revisions

`-l‘lockers’`

Print information about locked revisions only. In addition, if the comma-separated list ‘`lockers`’ of login names is given, ignore all locks other than those held by the ‘`lockers`’. For example,

```
rlog -L -R -lwft RCS/*
```

prints the name of RCS files locked by the user `wft`.

12.2.9 Informations about revisions

`-r‘revisions’`

prints information about revisions given in the commaseparated list `revisions` of revisions and ranges. A range `‘rev1’:‘rev2’` means revisions ‘`rev1`’ to ‘`rev2`’ on the same branch, `:‘rev’` means revisions from the beginning of the branch up to and including ‘`rev`’ , and `‘rev’:` means revisions starting with ‘`rev`’ to the end of the branch containing ‘`rev`’. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range. A branch followed by a `.` means the latest revision in that branch. A bare `-r` with no ‘`revisions`’ means the latest revision on the default branch, normally the trunk.

12.2.10 Information about Rev. with given state

`-s‘states’`
: prints information about revisions whose state attributes match one of the states given in the commaseparated list ‘`states`’.

12.2.11 Revisons checked in by user

`-w‘logins’`
: prints 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.

12.2.12 Modification Time

`-T`
: This option has no effect; it is present for compatibility with other RCS commands.

12.2.13 Version and Emulation

`-V`
: Print RCS’s version number.

`-V‘n’`
: Emulate RCS version ‘`n`’ when generating logs. See Section 6.2.15 [coOptV], page 38 for more.

12.2.14 Suffixes

`-x‘suffixes’`
: Use ‘`suffixes`’ to characterize RCS files. See Section 5.2.20 [ciOptx], page 30 for details.

12.2.15 Time zone

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`.

`-z‘zone’`
: specifies the date output format, and specifies the default time zone for ‘`date`’ in the `-d‘dates’` option. The ‘`zone`’ should be empty, a numeric UTC offset, or the special string `LT` for local time. The default is an empty ‘`zone`’ , which uses the traditional RCS format of UTC without any time zone indication and with slashes separating the parts of the date; oth- erwise, times are output in ISO 8601 format with time zone indication. For example, if local time is January 11, 1990, 8pm Pacific Standard Time, eight hours west of UTC, then the time is output as follows:

```
option    time output
-z        1990/01/12 04:00:00       (default)
-zLT      1990-01-11 20:00:00-08
-z+05:30  1990-01-12 09:30:00+05:30
```

12.2.16 Examples

```
rlog  -L  -R  RCS/*
rlog  -L  -h  RCS/*
rlog  -L  -l  RCS/*
rlog  RCS/*
```

The first command prints the names of all RCS files in the subdirectory RCS that have locks. The second command prints the headers of those files, and the third prints the headers plus the log messages of the locked revisions. The last command prints complete information.

12.2.17 Environment

`RCSINIT`

options prepended to the argument list, separated by spaces. See Section 5.8 [ciEnv], page 33 for details.

12.3 Diagnostics

The exit status is zero if and only if all operations were successful.

12.4 Bugs

The separator for revision ranges in the `-r` option used to be `-` instead of `:`, but this leads to confusion when symbolic names contain `-`. For backwards compatibility `rlog -r` still supports the old `-` separator, but it warns about this obsolete use.

13 merge – three-way file merge

13.1 Description

`merge` incorporates all changes that lead from '`file2`' to '`file3`' into '`file1`'. The result ordinarily goes into '`file1`'. merge is useful for combining separate changes to an original. Suppose '`file2`' is the original, and both '`file1`' and '`file3`' are modifications of '`file2`' . Then merge combines both changes.

A conflict occurs if both '`file1`' and '`file3`' have changes in a common segment of lines. If a conflict is found, merge normally outputs a warning and brackets the conflict with `<<<<<<<` and `>>>>>>>` lines. A typical conflict will look like this:

```
<<<<<<< file A
lines in file A
=======
lines in file B
>>>>>>> file B
```

If there are conflicts, the user should edit the result and delete one of the alternatives.

13.2 Command line options of merge

13.2.1 Output conflicts

`-A`

Output conflicts using the `-A` style of `diff3`, if supported by diff3. This merges all changes leading from '`file2`' to '`file3`' into '`file1`', and generates the most verbose output.

13.2.2 Specify conflict styles

`-E -e`

These options specify conflict styles that generate less information than `-A`. See `diff3` for details. The default is `-E`. With `-e`, merge does not warn about conflicts.

13.2.3 Label

`-L'label'`

This option may be given up to three times, and specifies labels to be used in place of the corresponding file names in conflict reports. That is, `merge -L x -L y -L z a b c` generates output that looks like it came from files '`x`', '`y`' and '`z`'instead of from files '`a`', '`b`' and '`c`'.

13.2.4 Standard output

`-p`

Send results to standard output instead of overwriting ‘`file1`’.

13.2.5 Quiet Mode

`-q`

Quiet; do not warn about conflicts.

13.2.6 Version

`-V`

Print’s version number.

13.3 Diagnostics

Exit status is 0 for no conflicts, 1 for some conflicts, 2 for trouble.

13.4 Bugs

It normally does not make sense to merge binary files as if they were text, but merge tries to do it anyway.

Appendix A Format of RCS file

A.1 Description

An RCS file's contents are described by the grammar below. The text is free format: space, backspace, tab, newline, vertical tab, form feed, and carriage return (collectively, `white space`) have no significance except in strings. However, white space cannot appear within an id, num, or sym, and an RCS file must end with a newline.

Strings are enclosed by `@`. If a string contains a `@`, it must be doubled; otherwise, strings can contain arbitrary binary data.

The meta syntax uses the following conventions: '`|`' (bar) separates alternatives; '`{`' and '`}`' enclose optional phrases; '`{`' and '`}`*' enclose phrases that can be repeated zero or more times; '`{`' and '`}`+' enclose phrases that must appear at least once and can be repeated; Terminal symbols are in boldface; nonterminal symbols are in `italic`.

```
rcstext    ::=  admin {delta}* desc {deltatext}*
admin      ::=  head {num};
                { branch   {num}; }
                access {id}*;
                symbols {sym : num}*;
                locks {id : num}*;  {strict  ;}
                { comment  {string}; }
                { expand   {string}; }
                { newphrase }*
delta      ::=  num
                date num;
                author id;
                state {id};
                branches {num}*;
                next {num};
                { newphrase }*
desc       ::=  desc string
deltatext  ::=  num
                log string
                { newphrase }*
                text string
num        ::=  {digit | .}+
digit      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
id         ::=  {num} idchar {idchar | num }*
sym        ::=  {digit}* idchar {idchar | digit }*
idchar     ::=  any visible graphic character except special
special    ::=  $ | , | . | : | ; | @
string     ::=  @{any character, with @ doubled}*@
newphrase  ::=  id word* ;
word       ::=  id | num | string | :
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers can overlap. In most environments RCS uses the ISO 8859/1 encoding: visible graphic characters are codes 041-176 and 240-377, and white space characters are codes 010-015 and 040.

Dates, which appear after the date keyword, are of the form `Y.mm.dd.hh.mm.ss`, where `Y` is the year, `mm` the month (01-12), `dd` the day (01-31), `hh` the hour (00-23), `mm` the minute (00-59), and `ss` the second (00-60). `Y` contains just the last two digits of the year for years from 1900 through 1999, and all the digits of years thereafter. Dates use the Gregorian calendar; times use UTC.

The `newphrase` productions in the grammar are reserved for future extensions to the format of RCS files. No `newphrase` will begin with any keyword already in use.

The `delta` nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the trunk, and are linked through the next field in order of decreasing numbers. The head field in the `admin` node points to the head of that sequence (i.e., contains the highest pair). The branch node in the admin node indicates the default branch (or revision) for most RCS operations. If empty, the default branch is the highest branch on the trunk.

All `delta` nodes whose numbers consist of 2`n` fields (`n`>=2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first 2`n`-1 number fields are identical are linked through the next field in order of increasing numbers. For each such sequence, the `delta` node whose number is identical to the first 2`n`-2 number fields of the deltas on that sequence is called the branchpoint. The branches field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

A.2 Example Revision tree

The following diagram shows an example of an RCS file's organization.

```
                           Head
                             |
                             v                        / \
                         ---------                   /   \
   / \          / \      |       |      / \         /     \
  /   \        /   \     |  2.1  |     /   \       /       \
 /     \      /     \    |       |    /     \     /         \
/1.2.1.3\    /1.3.1.1\   |       |   /1.2.2.2\   /1.2.2.1.1.1\
---------    ---------   ---------   ---------   -------------
    ^            ^           |           ^             ^
    |            |           v           |             |
   / \           |       ---------      / \            |
  /   \          |       \  1.3  /     /   \           |
 /     \         ---------\     /     /     \-----------
/1.2.1.1\                  \   /     /1.2.2.1\
---------                   \ /      ---------
    ^                        |           ^
    |                        v           |
    |                    ---------       |
    |                    \  1.2  /       |
    ----------------------\     /---------
                           \   /
                            \ /
                             |
                             v
                         ---------
                         \  1.1  /
                          \     /
                           \   /
                            \ /
```

Appendix B Books and related manuals

B.1 Feldman,Stuart I.

```
Feldman, Stuart I.,

"Make -- A Program for Maintaining Computer Programs",
Software -- Practice & Experience,
vol. 9, no. 3, pp. 255-265, March 1979.
```

B.2 Hunt, James W. and McIlroy, M. D.

```
Hunt, James W. and McIlroy, M. D.,

"An Algorithm for Differential File Comparison",
41, Computing Science Technical Report,
Bell Laboratories, June 1976.
```

B.3 Rochkind, Marc J.

```
Rochkind, Marc J.

"The Source Code Control System",
IEEE Transactions on Software Engineering, vol. SE-1,
no. 4, pp. 364-370, Dec. 1975.
```

B.4 Tichy, Walter F.

```
Tichy, Walter F.

"Design, Implementation, and Evaluation of a Revision Control System"
Proceedings of the 6th International Conference on Software Engineering,
pp. 58-67, ACM, IEEE, IPS, NBS, September 1982.
```

B.5 Leblang, David B. and Chase, Robert P.

```
Leblang, David B. and Chase, Robert P.

"Computer-Aided Software Engineering in a Distributed
Workstation Environment",
SIGPLAN Notices, vol. 19, no. 5, pp. 104-112, May 1984.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium
on Practical Software Development Environments.
```

B.6 Glasser, Alan L.

```
Glasser, Alan L.

"The Evolution of a Source Code Control System"
Software Engineering Notes, vol. 3, no. 5, pp. 122-125, Nov. 1978.
Proceedings of the Software Quality and Assurance Workshop.
```

B.7 Brown, H.B.

```
Brown, H.B.

"The Clear/Caster System",
Nato Conference on Software Engineering, Rome 1970
```

B.8 Habermann, A. Nico

```
Habermann, A. Nico

"A Software Development Control System",
Technical Report, Carnegie-Mellon University,
Department of Computer Science, Jan. 1979.
```

B.9 Code Management System

```
Digital Equipment Corporation

"Code Management System",
Document No. EA-23134-82, 1982
```

B.10 Lampson, Butler W. and Schmidt, Eric E.

```
Lampson, Butler W. and Schmidt, Eric E.

"Practical Use of a Polymorphic Applicative Language",
Proceedings of the 10th Symposium on Principles of
Programming Languages,
pp. 237-255, ACM, January 1983.
```

B.11 Tichy, Walter F.

```
Tichy, Walter F.

"A Data Model for Programming Support Environments and its Application",
Automated Tools for Information System Design and Development, ed.
Hans-Jochen Schneider and Anthony I. Wasserman,
North-Holland Publishing Company, Amsterdam 1982.
```

B.12 Heckel, Paul

```
Heckel, Paul

"A Technique for Isolating Differences Between Files",
Communications of the ACM, vol. 21, no. 4, pp. 264-268, April
1978.
```

B.13 Tichy, Walter F.

```
Tichy, Walter F.

"The String-to-String Correction Problem with Block Moves",
ACM Transactions on Computer Systems, vol. 2, no. 4, pp. 309-321,
Nov. 1984.
```

Index

Table of Contents