USING UNIX TOOLS FOR SMALL PROJECTS

Scott D. Carson
University of Virginia

Computer Science Report No. RM-86-04
June 27, 1986

Using UNIX™ Tools for Small Projects

Scott D. Carson
Department of Computer Science
University of Virginia
Charlottesville, VA

ABSTRACT

Even though large computer projects are often considered most important, the fact remains that a significant number of computer projects are small, one-person jobs such as programs or documents. The need for careful software management is not as great with small projects because communication problems are obviously reduced. However, to the extent that it is helpful for the implementor and beneficial to the implementor's organization, certain software development practices should be followed.

This paper describes a method for developing small projects using three UNIX tools: *compile*, *make*, and *RCS*. The paper does not define a software development methodology; instead, it proposes certain conventions that are general enough to apply to a variety of projects. Several example applications are discussed.

UNIX is a trademark of AT&T Bell Laboratories.

The objective of this paper is to show how a few software tools can be used together to form a simple, effective project development environment. The projects addressed are small, requiring a single person. They need not be restricted to programming projects; the methods apply to other projects such as documentation.

It is assumed that the reader is familiar with the existing documentation (manual pages) for the following tools:

compile - provides a mechanism for inserting "compilation" directives in files,
make - provides for the specification of dependencies among modules that make up a project [1], and
RCS - provides a version control mechanism for files [3].

Manual pages for these tools are included at the end of this paper.

We will examine three examples in this paper: a very small project, a slightly larger documentation project, and a programming project. The very small project is best implemented using the *compile* tool, while the two larger projects use *make* and *RCS* in concert.

1. A Very Small Project

The smallest projects developed on the computer use only a single file, and rarely require version control. Since there is only a single file involved, there is no need to specify dependencies among files. Examples include business letters, memoranda, and short programs. The only problem facing the implementor is to remember how to produce the "output" from the source file.

For instance, suppose that a memorandum is to be written. Further, suppose that the source text for this memorandum exists in a file called "memo", and that producing the typeset version of the memorandum requires the tools "tbl", "pic", and "qtroff". It is easy for the writer to remember the sequence of commands necessary to process the source file at the time when the memorandum is written, yet if interruptions occur the command sequence may be forgotten.

The solution to this problem is to use the *compile* program to specify the command sequence as part of the source text. The essential idea is as follows. First, the command sequence is inserted into the source file as a specially flagged line of text. Then, instead of actually typing the command sequence to process the file, the writer invokes the *compile* command. The *compile* program looks for this special line in the source file, extracts it, performs certain substitutions (see the manual page), and passes the line to the shell for execution.

Returning to the example, suppose that the command required to process the file "memo" is the following:

```
pic memo | tbl | qtroff -me
```

The *compile* program looks for a line that contains the keyword "$Compile:", and uses the rest of that line as a "compilation" directive. This special line is normally inserted into the source file as a comment. Thus, in a file that is to be processed by *qtroff*, the line would be the following:

```
\"  $Compile: pic %f | tbl | qtroff -me
```

Notice that the file name itself is not specified in the compilation directive. Instead, the special string "%f" is used to mean "the file specified as the argument to *compile*". Thus, to process the file "memo", one would type

```
compile memo
```

The *compile* program would then extract the compilation directive line, substitute the word "memo" for the string "%f", and execute the line as a command.

It is important to avoid naming the source file in the compilation directive, so that if the file is either renamed or named by another means then the *compile* program will still be able to proceed correctly. For instance, if the file name is changed to "memo.january.2", then typing the command

```
compile memo.january.2
```

produces the correct result. Additionally, if the file is named with an absolute path name, and if the user's current directory is elsewhere, then compile will substitute correctly. In other words, if the writer's current directory is "/usr/sdc" and the file is in "/usr/memos/memo.january.2", then typing the command

```
compile /usr/memos/memo.january.2
```

causes *compile* to execute the command

```
pic /usr/memos/memo.january.2 | tbl | qtroff -me
```

which has the desired effect.

Compile is equally suitable for programs written in *c* or other languages. Naturally, the compilation directive changes, but the overall process is the same. For example, a *c* program might contain the following "comment":

```
/*
 *      $Compile: cc -O -o %F %f -lcurses -ltermcap
 */
```

If the file containing the compilation directive is named "program.c", then typing

```
compile program.c
```

executes the command

```
cc -O -o program program.c -lcurses -ltermcap
```

The same comments about the use of file names in compilation directives apply.

2. A Larger Documentation Project

Single-person projects that require more than one file are inherently more complicated than those that require only one file. The first problem is that the files must be put together in some fashion to produce the result file or files. The second problem is that some degree of version control may be required, both to preserve old versions of the files and to build versions of the result files based on specific versions of the input files. These problems are solved using two UNIX tools: *make* and *RCS.*

Make

Make is a program that allows for the specification of dependencies among files that are put together to make result files. The dependencies and the instructions that create the result files from the input files are specified in a file known as a *Makefile*. Each entry in the *Makefile* consists of a result file name, followed by a list of files on which it depends, followed by the sequence of shell commands that are used to make the result file from the input files.

When the *make* command is typed, the *make* program looks for a file called *Makefile* in the current directory. *Make* extracts the result file names and the dependent file names from the *Makefile* and checks the dates of the result files against those of the dependent files. If any of the dependent files have been modified since the last time the result file was created, then the corresponding result file is re-created according to the appropriate sequence of commands.

For example, suppose that a document consists of four source files called "header", "introduction", "body", and "conclusion". The result file is to be known as "paper", and it is to be in printer-ready format. The command sequence used to format the paper is as follows:

```
cat header introduction body conclusion | pic | tbl | qtroff -me -t >paper
```

Whenever one of the four input files is changed the result file must be re-created. This is expressed in the *Makefile* as follows:

```
paper:  header introduction body conclusion
        cat header introduction body conclusion | pic | tbl | qtroff -me -t >paper
```

If any of the four input files is modified, then typing *make* causes the result file to be updated by executing the specified commands. If the result file has been created since the last modification to the input files, then *make* simply says

```
'paper' is up to date.
```

The *Makefile* may contain definitions for multiple result files. In fact, the result file name (left hand side) of a *make* dependency rule may represent a file that does not exist. For example, the following two definitions might also appear in the *Makefile*:

```
% ls
Makefile        RCS     body    conclusion      header  introduction
% ci header
RCS/header,v  <--  header
initial revision: 1.1
enter description, terminated with ↑D or '.':
NOTE: This is NOT the log message!
>> Troff macro definitions, page set up commands.
>> .
done
% rcs -U header
RCS file: RCS/header,v
done
% co header
RCS/header,v  -->  header
revision 1.1
done
%
```

Figure 1 - RCS Terminal Session

```
print: paper
       lpr -Plaser -t paper

spelling_errors:  header introduction body conclusion
        cat header introduction body conclusion | spell >spelling_errors
```

The printed version of the paper and the list of spelling errors can be obtained by typing *make print* and *make spelling_errors*, respectively.

RCS

RCS, the "Revision Control System", is a set of programs that help manage multiple versions of files. The text of each file is stored in another file known as an *RCS* file. The *RCS* file contains information sufficient to construct any previous version of the text file, along with a description of each version.

Most work with *RCS* can be accomplished with three commands:

ci - check in a new version of a file,
co - check out an existing version of a file, and
rcs - modify parameters of an RCS file.

```
#
#   Makefile for ''paper''
#
paper:  header introduction body conclusion
        cat header introduction body conclusion | pic | tbl | qtroff -me -t >paper

print: paper
       lpr -Plaser -t paper

spelling_errors:  header introduction body conclusion
        cat header introduction body conclusion | spell >spelling_errors

header:
        co header

introduction:
        co introduction

body:
        co body

conclusion:
        co conclusion

paper.release1:
        mkdir temp
        (cd temp ; \
         co -rrelease1 ../RCS/header,v ; \
         co -rrelease1 ../RCS/introduction,v ; \
         co -rrelease1 ../RCS/body,v ; \
         co -rrelease1 ../RCS/conclusion,v ; \
         cat header introduction body conclusion | pic | tbl | qtroff -me -t >../paper.release1)
        /bin/rm -rf temp
```

Figure 2 - Completed Makefile

The *ci* command is used to create a version of the file whose name is its argument. The corresponding *RCS* file is expected to be in either the current directory or a directory called "RCS" in the current directory. Normally, the latter situation is preferable. The name of the *RCS* file is formed by adding the suffix ",v" to the source file name.

If no *RCS* file exists for the specified source file, then *ci* creates a new one. At this time the user enters a short description of the file. Returning to the previous example, suppose that the files "header", "introduction", and so on exist in the current directory. Additionally, suppose that a directory "RCS" exists, also in the current directory. To establish an *RCS* file for the file "header", one would type *ci header*. The terminal session is shown in Figure 1.

Once a file has been checked in, it is inaccessible. The *co* command is used to retrieve versions of a file. By default, *co* checks out the newest version. Thus, to retrieve the file "header" for modification, one would type *co header*. After modification, the file would be checked in again, and a new version established.

The *rcs* command modifies information in *RCS* files. One of the most important uses of *rcs* is to execute the command *rcs -U*, which "turns off" a file-locking protocol intended to avoid multiple concurrent updates to the *RCS* file. Locking should be turned off (permanently) for each file when it is first checked in (Figure 1).

Incorporating the *RCS* commands into the *Makefile* is easy. Rules are introduced for deriving each source file from its corresponding *RCS* file, so that each file is automatically checked out if it does not exist already. For example, the *Makefile* rule for the file "header" would look like this:

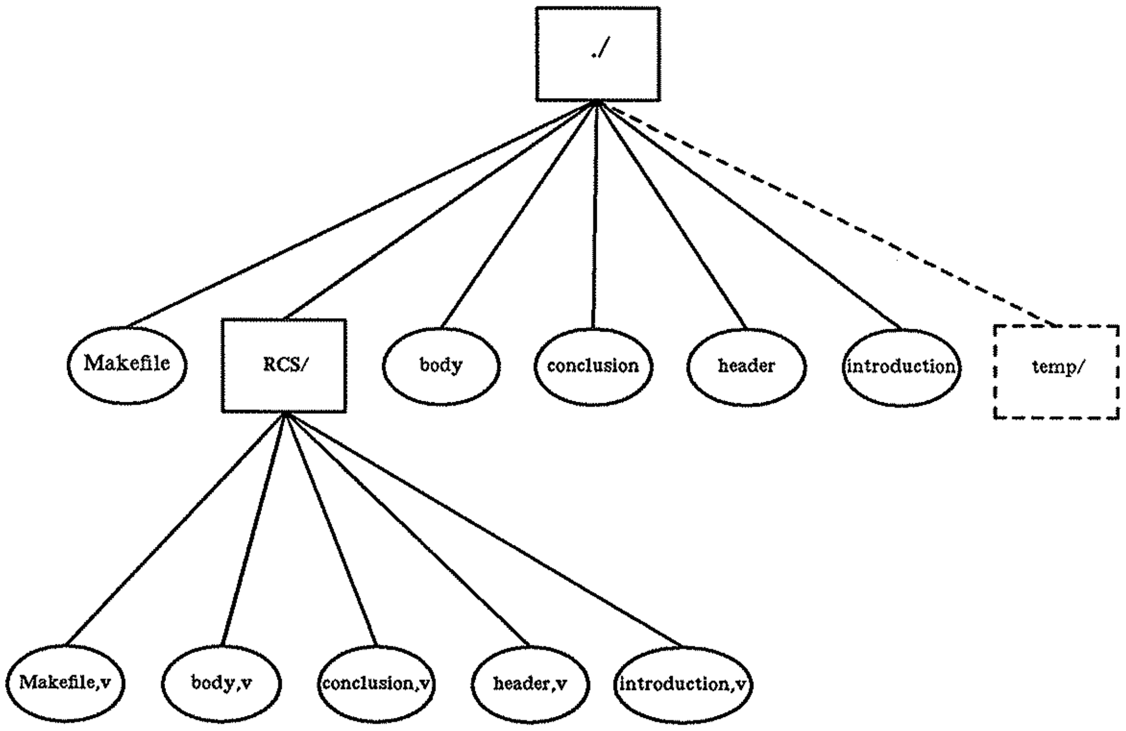

Figure 3 - Documentation Project Directory Structure

```
header:
        co header
```

Note that the file "header" is not dependent on the *RCS* file; "header" is only checked out if and only if it does not exist. This means, for instance, that a modification to the *RCS* file via the cs command will not cause *make* to check out another version automatically. The typical result of these conventions is that the latest version of each source file is present in the working directory.

The *rcs* command can be used to assign symbolic names to particular revisions of each file. These can be checked out collectively, by (symbolic) name. Thus, it is possible to identify particular versions of source files that make up a "release" of a result file.

For example, suppose that we want to identify version 1.2 of the file "header" as belonging to a collection known as "release1". The command *rcs -nrelease1:1.2* accomplishes the task. After a revision of each of the source files is identified as belonging to "release1", a *Makefile* rule can be defined to produce the appropriate version of the result file, "paper.release1". The *Makefile* rule looks like this:

```
paper.release1:
        mkdir temp
        (cd temp ; \
         co -rrelease1 ../RCS/header,v ; \
         co -rrelease1 ../RCS/introduction,v ; \
         co -rrelease1 ../RCS/body,v ; \
         co -rrelease1 ../RCS/conclusion,v ; \
         cat header introduction body conclusion | pic | tbl | qtroff -me -t >../paper.release1)
        /bin/rm -rf temp
```

Note that a temporary directory, "temp", is created and deleted. Note also that "paper.release1" has no dependencies in the *Makefile* rule; if it exists, then it is up to date. This makes intuitive sense: once "paper.release1" exists it need never be re-created because its source files cannot change.

The *Makefile* itself may also be checked in. A complete *Makefile* for this example is shown in Figure 2. A directory tree listing corresponding to the example is shown in Figure 3.

3. A C Programming Project

The same set of software tools can be used to maintain programming projects. Small programming projects typically consist of a main program, several separately-compiled source modules, and a series of "header" files that are "included" at compilation time.

Suppose, for example, that a software project is made up of five files:

main.c - the main program,
module1.c - function definitions,
module2.c - function definitions,
header1.h - defined constants, "included" by module1.c and main.c, and
header2.h - defined constants, "included" by module2.c and main.c.

The executable version of the program, "a.out", is constructed from the relocatable object version of each of the source files: "main.o", "module1.o", and "module2.o". These are created using the c compiler. Each object module depends on both its source file and the header files that it "includes". The *Makefile* rules are expressed as follows:

```
a.out:  main.o module1.o module2.o
        cc main.o module1.o module2.o
```

```
SOURCES=main.c module1.c module2.c header1.h header2.h

a.out:  main.o module1.o module2.o
        cc main.o module1.o module2.o

main.o: main.c header1.h header2.h
        cc -O -c main.c

module1.o: module1.c header1.h
        cc -O -c module1.c

module2.o: module2.c header2.h
        cc -O -c module2.c

$(SOURCES):
        co $@

a.out.release1:
        mkdir temp
        (cd temp ; \
         co -rrelease1 ../RCS/* ;\
         make ;\
         mv a.out ../a.out.release1)
        /bin/rm -rf temp
```

Figure 4 - C Programming Project Makefile

```
main.o: main.c header1.h header2.h
        cc -O -c main.c

module1.o: module1.c header1.h
        cc -O -c module1.c

module2.o: module2.c header2.h
        cc -O -c module2.c
```

As with the documentation project, *RCS* can be used to maintain multiple versions of each source file. Files are checked in and out with *ci* and *co.* The *rcs* command can be used to change the file locking mechanism and to define symbolic version names.

The *Makefile* rule for the file "main.c" is as follows:

```
    main.c:
            co main.c
```

Thus, main.c can be produced whenever absent by checking it out.

Figure 4 shows the complete *Makefile* for the c programming project. It contains the rules for producing the executable file, each object file, and each source file. In addition, the *Makefile* contains rules for producing an executable file, "a.out.release1", from versions of the source files symbolically named "release1".

Note that the *RCS* rules are expressed more compactly than in previous examples, using *make*'s macro definition mechanism. Also note the different procedure used to build the "release1" version; in this example *make* is used recursively. This means that the *Makefile* must be checked in, and that a version of it must be symbolically named "release1".

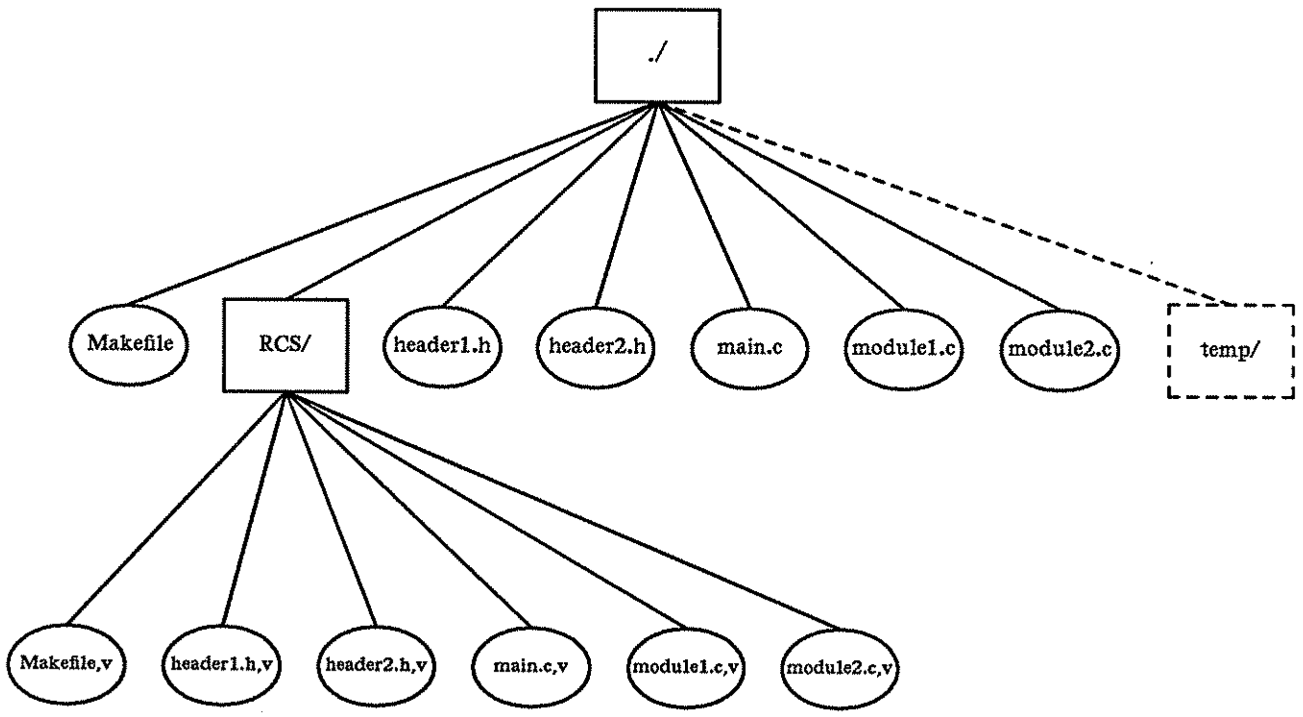

Figure 5 - C Programming Project Directory Structure

Figure 5 shows the directory structure for the c programming project. The temporary directory used to make "a.out.release1" is also shown.

4. Conclusion

The three software tools described in this paper can be used effectively for a variety of small computer projects. *Compile* is best used for very small projects, while *make* and *RCS* are suitable for larger projects. The effectiveness of the tools is independent of the particular application.

The ideas presented in this paper can be extended to encompass larger, multi-person projects (see [2]). However, additional considerations, such as file locking and directory structure, become more important. These concerns are addressed in the detailed documentation for the tools.

REFERENCES

[1] S. I. Feldman, "Make - a Program for Maintaining Computer Programs," Bell Laboratories, Murray Hill.*

[2] Peter J. Nicklin, "The SPMS Software Project Management System," Division of Structural Engineering and Structural Mechanics, Department of Civil Engineering, University of California at Berkeley, Berkeley, CA. *

*Included as part of the Berkeley UNIX manual set.

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

NAME

compile - detect compilation directives in files

SYNOPSIS

compile file ...

DESCRIPTION

Compile is a utility for detecting compilation directives within files. *Compile* searches through the first block of the named file(s), searching for a marker, currently of the form "$Compile:". When this marker is detected, the remainder of the line, up to a newline, or an occurence of two sequential unescaped dollar signs ('$'), is copied to the standard output with some character strings substituted.

The strings that are substituted are:

%f	- the full name of the file specified on the command line.
%d	- the *directory* part of the specified file, including the final '/'.
%x	- the *extension* on the specified file, if any. This is the string following the *last* occurence of a period ('.').
%p	- the *prefix* part of the specified file. This is the string which follows the final slash and precedes the first period. (This is useful for SCCS files.)
%F	- the filename-only part of the specified file. This does not include the path, prefix, or extension parts.

Also, some C-like escape sequences are substituted:

!n	- newline
!t	- tab
\\	- backslash
!<nnn>	- the character whose octal value is <nnn>

EXAMPLES

Compile would most commonly be used to produce input for the shell. The following line might occur in a C program source file:

```
/*
 * $Compile: cc -o %F -DFOO=1 %f
 */
```

If the file were called "foo.c", *compile* would produce on the standard output:

```
cc -o foo -DFOO=1 foo.c
```

Thus, the command might be used as follows:

```
compile foo.c | sh
```

Compile is in no way limited to "compiling" source language programs. It can be used on *nroff* source by adding a line near the top of an *nroff* source file, e.g.

```
\"  $Compile: nroff -ms -rO8 %f >%F.out
```

BUGS

Compile currently always selects the first marker. The marker cannot be specified by the user. Extensions and prefices are rather rigidly defined. These bugs will be fixed with future extensions. See the comment at the beginning of the source code for other planned enhancements.

ENHANCEMENTS

Implemented the suggestion of ihuxf!larry - compile actually executes the command it produces.

AUTHOR

S. McGeady
Tektronix, Inc.
(503) 685-2555

```
stevenm@tektronix            (CSNET)
stevenm.tektronix@rand-relay        (ARPA)
decvax!teklabs!stevenm              (UUCP)
ucbvax!teklabs!stevenm              (UUCP)
zehntel!tektronix!stevenm    (UUCP)
```

rule as described in the next paragraph exist is inferred. The default list is

```
.SUFFIXES: .out .o .c .e .r .f .y .l .s .p
```

The rule to create a file with suffix *s2* that depends on a similarly named file with suffix *s1* is specified as an entry for the 'target' *s1s2*. In such an entry, the special macro $* stands for the target name with suffix deleted, $@ for the full target name, $< for the complete list of prerequisites, and $? for the list of prerequisites that are out of date. For example, a rule for making optimized '.o' files from '.c' files is

```
.c.o: ; cc -c -O -o $@ $*.c
```

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, 'CFLAGS' is used for *cc*(1) options, 'FFLAGS' for *f77*(1) options, 'PFLAGS' for *pc*(1) options, and 'LFLAGS' and 'YFLAGS' for *lex* and *yacc*(1) options. In addition, the macro 'MFLAGS' is filled in with the initial command line options supplied to *make*. This simplifies maintaining a hierarchy of makefiles as one may then invoke *make* on makefiles in subdirectories and pass along useful options such as **-k**.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target '.SILENT' is in *makefile*, or the first character of the command is '@'.

Commands returning nonzero status (see *intro*(1)) cause *make* to terminate unless the special target '.IGNORE' is in *makefile* or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target is a directory or depends on the special name '.PRECIOUS'.

Other options:

-i Equivalent to the special entry '.IGNORE:'.

-k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

-n Trace and print, but do not execute the commands needed to update the targets.

-t Touch, i.e. update the modified date of targets, without executing any commands.

-r Equivalent to an initial special entry '.SUFFIXES:' with no list.

-s Equivalent to the special entry '.SILENT:'.

FILES

makefile, Makefile

SEE ALSO

sh(1), touch(1), f77(1), pc(1)
S. I. Feldman *Make - A Program for Maintaining Computer Programs*

BUGS

Some commands return nonzero status inappropriately. Use **-i** to overcome the difficulty.
Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*.

NAME

make - maintain program groups

SYNOPSIS

make [**-f** makefile] [option] ... file ...

DESCRIPTION

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **-f** option is present, 'makefile' and 'Makefile' are tried in order. If *makefile* is '-', the standard input is taken. More than one **-f** option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target. If a name appears on the left of more than one 'colon' line, then it depends on all of the names on the right of the colon on those lines, but only one command sequence may be specified for it. If a name appears on a line with a double colon **::** then the command sequence following that line is performed only if the name is out of date with respect to the names to the right of the double colon, and is not affected by other double colon lines on which that name may appear.

Two special forms of a name are recognized. A name like *a*(*b*) means the file named *b* stored in the archive named *a*. A name like *a*((*b*)) means the file stored in archive *a* containing the entry point *b*.

Sharp and newline surround comments.

The following makefile says that 'pgm' depends on two files 'a.o' and 'b.o', and that they in turn depend on '.c' files and a common file 'incl'.

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o: incl a.c
        cc -c a.c
b.o: incl b.c
        cc -c b.c
```

Makefile entries of the form

```
string1 = string2
```

are macro definitions. Subsequent appearances of $(*string1*) or ${*string1*} are replaced by *string2*. If *string1* is a single character, the parentheses or braces are optional.

Make infers prerequisites for files for which *makefile* gives no construction commands. For example, a '.c' file may be inferred as prerequisite for a '.o' file and be compiled to produce the '.o' file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred according to selected suffixes listed as the 'prerequisites' for the special name '.SUFFIXES'; multiple lists accumulate; an empty list clears what came before. Order is significant; the first possible name for which both a file and a

NAME

rcsintro - introduction to RCS commands

DESCRIPTION

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc.

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

Functions of RCS

- Storage and retrieval of multiple revisions of text. RCS saves all old revisions in a space efficient way. Changes no longer destroy the original, because the previous revisions remain accessible. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
- Maintenance of a complete history of changes. RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of checkin, and a log message summarizing the change. The logging makes it easy to find out what happened to a module, without having to compare source listings or having to track down colleagues.
- Resolution of access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.
- Maintenance of a tree of Revisions. RCS can maintain separate lines of development for each module. It stores a tree structure that represents the ancestral relationships among revisions.
- Merging of revisions and resolution of conflicts. Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged affect the same sections of code, RCS alerts the user about the overlapping changes.
- Release and configuration control. Revisions can be assigned symbolic names and marked as released, stable, experimental, etc. With these facilities, configurations of modules can be described simply and directly.
- Automatic identification of each revision with name, revision number, creation time, author, etc. The identification is like a stamp that can be embedded at an appropriate place in the text of a revision. The identification makes it simple to determine which revisions of which modules make up a given configuration.
- Minimization of secondary storage. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

Getting Started with RCS

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

```
ci  f.c
```

This command creates the RCS file f.c,v, 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 checkin commands will ask you for a log entry, which should summarize the changes that you made.

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

```
co  f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it back in by invoking

```
ci  f.c
```

Ci increments the revision number properly. If *ci* complains with the message

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

then your system administrator has decided to create all RCS files with the locking attribute set to 'strict'. In this case, you should have locked the revision during the previous checkout. Your last checkout should have been

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

Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout 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 CHECKIN 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 off the RCS file need not have a lock for checkin; 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 working 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 3 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 checkin (in case you want to continue editing), invoke

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

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

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

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

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

Automatic Identification

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

```
$Header$
```

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

```
$Header:  filename  revision_number  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[] = "$Header$";
```

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 checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co*(1) for details.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/05/11 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci(1), co(1), ident(1), merge(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1), rcsfile(5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control

System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

ci - check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

Ci stores new revisions into RCS files. Each file name ending in ',v' 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.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of *co* (1)).

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

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *ci* looks for the RCS file first in the directory ./RCS and then in the current directory.

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, unless locking is set to *strict* (see *rcs* (1)). A lock held by someone else may be broken with the *rcs* command.

Normally, *ci* checks whether the revision to be deposited is different from the preceding one. If it is not different, *ci* either aborts the deposit (if **-q** is given) or asks whether to abort (if **-q** is omitted). A deposit can be forced with the **-f** option.

For each revision deposited, *ci* prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. If several files are checked in, *ci* asks whether to reuse the previous log message. If the std. input is not a terminal, *ci* suppresses the prompt and uses the same log message for all files. See also **-m**.

The number of the deposited revision can be given by any of the options **-r**, **-f**, **-k**, **-l**, **-u**, or **-q** (see **-r**).

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 (see **-t** below).

-r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, *ci* derives 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 the caller holds no lock, but he is the owner of the file and locking is not set to *strict*, then the revision is appended to the trunk.

If *rev* indicates 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* indicates a branch instead of a revision, 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*.

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

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

-k[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see *co* (1)), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. 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 its original number, date, author, and state.

-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.

-u[*rev*] works like **-l**, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

-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.

-m*msg* uses the string *msg* as the log message for all revisions checked in.

-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.

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

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

-t[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *ci* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if **-t** is not given. The prompt is suppressed if std. input is not a terminal.

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 always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

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.

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *Ci* always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

co - check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

Co retrieves revisions from RCS files. Each file name ending in ',v' is taken to be an RCS file. All other files are assumed to be working files. *Co* retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

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

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 ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *co* looks for the RCS file first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may 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. Locking a revision currently locked by another user fails. (A lock may be broken with the *rcs* (1) command.) *Co* with locking 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. *Co* without locking is not subject to accesslist restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the *selected branch*. The selected branch is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options **-l**, **-p**, **-q**, or **-r**.

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

-l[*rev*] locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option **-r** for handling of the revision number *rev*.

-p[*rev*] prints the retrieved revision on the std. output rather than storing it in the working file. This option is useful when *co* is part of a pipe.

-q[*rev*] quiet mode; diagnostics are not printed.

-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 may be given in free format and are converted to local time. Examples of formats for *date*:

22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21 (free format),
Fri, April 16 15:52:25 EST 1982 (output of ctime).

Most fields in the date and time may be defaulted. *Co* determines the defaults 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 current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-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. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the **-n** option of the commands *ci* and *rcs*.

-s*state* retrieves the latest revision on the selected branch whose state is set to *state*.

-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.

-j*joinlist* generates a new revision which is the join of the revisions on *joinlist*. *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 options **-l**, ..., **-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* prints a warning and includes the overlapping sections, delimited by the lines <<<<<<< *rev1*, =======, and >>>>>>> *rev3*.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option **-l** is present, the initial *rev1* is locked.

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 may 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.

Keywords and their corresponding values:

$Author$	The login name of the user who checked in the revision. qms2. Class$
$Date$	The date and time the revision was checked in.
$Header$	A standard header containing the RCS file name, the revision number, the date, the author, and the state.
$Locker$	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 file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after *$Log:...$*. This is useful for accumulating a complete change log in a source file.
$Revision$	The revision number assigned to the revision.
$Source$	The full pathname of the RCS file.
$State$	The state assigned to the revision with *rcs -s* or *ci -s*.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

```
co  io.c;     co RCS/io.c,v;     co  io.c,v;
co  io.c  RCS/io.c,v;     co  io.c  io.c,v;
co  RCS/io.c,v  io.c;     co  io.c,v  io.c;
```

FILE MODES

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to *strict* (see *rcs* (1)).

If a file with the name of the working file exists already and has write permission, *co* aborts the checkout if **-q** is given, or asks whether to abort if **-q** is not given. If the existing working file is not writable, it is deleted before the checkout.

FILES

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option **-d** gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character '\&' into the keyword.

BUGS

The option **-j** does not work for files that contain lines with a single '.'.

NAME

rcs - change RCS file attributes

SYNOPSIS

rcs [options] file ...

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.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rcs* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

-i creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, *rcs* tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

-a*logins* appends the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

-A*oldfile* appends the access list of *oldfile* to the access list of the RCS file.

-e[*logins*] erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, the entire access list is erased.

-c*string* sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword Log during checkout (see *co*). This is useful for programming languages without multi-line comments. During *rcs -i* or initial *ci*, the comment leader is guessed from the suffix of the working file.

-l[*rev*] locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with *ci* or *rcs -u* (see below).

-u[*rev*] unlocks the revision with number *rev*. If a branch is given, the latest revision on that branch is unlocked. If *rev* is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may 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 with a line containing a single '.' or control-D.

-L sets 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.

-U sets 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. The default (**-L** or **-U**) is determined by your system administrator.

-n*name*[*:rev*]
associates the symbolic name *name* with the branch or revision *rev*. *Rcs*

prints an error message if *name* is already associated with another number. If *rev* is omitted, the symbolic name is deleted.

-N*name*[:*rev*]
: same as **-n**, except that it overrides a previous assignment of *name*.

-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 may have branches or locks.

-q
: quiet mode; diagnostics are not printed.

-s*state*[:*rev*]
: sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed; If *rev* is a branch number, the latest revision on that branch is assumed. Any identifier is acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, *ci* sets the state of a revision to *Exp*.

-t[*txtfile*]
: writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *rcs* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. If the **-i** option is present, descriptive text is requested even if **-t** is not given. The prompt is suppressed if the std. input is not a terminal.

DIAGNOSTICS

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

FILES

The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. *Rcs* creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, *rcs* always creates a new file. On successful completion, *rcs* deletes the old one and renames the new one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

co (1), ci (1), ident(1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

rcsdiff - compare RCS revisions

SYNOPSIS

rcsdiff [-**b**] [-**cefhn**] [-**r***rev1*] [-**r***rev2*] file ...

DESCRIPTION

Rcsdiff runs *diff* (1) to compare two revisions of each RCS file given. A file name ending in ',v' is an RCS file name, otherwise a working file name. *Rcsdiff* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). Pairs consisting of both an RCS and a working file name may also be specified.

The options **-b**, **-c**, **-e**, **-f**, and **-h** have the same effect as described in *diff* (1); option **-n** generates an edit script of the format used by RCS.

If both *rev1* and *rev2* are omitted, *rcsdiff* compares the latest revision on 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.

EXAMPLES

The command

```
rcsdiff   f.c
```

runs *diff* on the latest trunk revision of RCS file f.c,v and the contents of working file f.c.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), co (1), diff (1), ident (1), rcs (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

rcsmerge - merge RCS revisions

SYNOPSIS

rcsmerge **-r***rev1* [**-r***rev2*] [**-p**] file

DESCRIPTION

Rcsmerge incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If **-p** is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. *Merge* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in *co -j*. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when 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.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), co (1), merge (1), ident (1), rcs (1), rcsdiff (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Rcsmerge does not work for files that contain lines with a single '.'.

NAME

rcsmerge - merge RCS revisions

SYNOPSIS

rcsmerge **-r***rev1* [**-r***rev2*] [**-p**] file

DESCRIPTION

Rcsmerge incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If **-p** is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. *Merge* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in *co -j*. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when 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.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), co (1), merge (1), ident (1), rcs (1), rcsdiff (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Rcsmerge does not work for files that contain lines with a single '.'.

NAME

rlog - print log messages and other information about RCS files

SYNOPSIS

rlog [options] file ...

DESCRIPTION

Rlog prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rlog* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

Rlog prints the following information for each RCS file: RCS file name, working file name, head (i.e., the number of the latest revision on the trunk), 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. Without options, *rlog* prints complete information. The options below restrict this output.

-L ignores RCS files that have no locks set; convenient in combination with **-R**, **-h**, or **-l**.

-R only prints the name of the RCS file; convenient for translating a working file name into an RCS file name.

-h prints only RCS file name, working file name, head, access list, locks, symbolic names, and suffix.

-t prints the same as **-h**, plus the descriptive text.

-d*dates* prints 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*, (inclusive). A range of the form <*d* or *d*> selects all revisions dated *d* or earlier. A range of the form *d*< or >*d* selects all revisions dated *d* or later. 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 *co* (1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon.

-l[*lockers*] prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

-r*revisions*

prints information about revisions given in the comma-separated 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.

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

-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.

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

EXAMPLES

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

The first command prints the names of all RCS files in the subdirectory 'RCS' which 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.

DIAGNOSTICS

The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.2 ; Release Date: 83/05/11 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS