Source Code Control System User's Guide

L. E. Bonanni
C. A. Salemi

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

The Source Code Control System (SCCS) is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It provides facilities for storing, updating, and retrieving any version of a file of text, for controlling updating privileges to that file, for identifying the version of a retrieved file, and for recording who made each change, when and where it was made, and why. SCCS is a collection of programs that run under the UNIX† Time-Sharing System.

This document, together with relevant portions of the *UNIX User's Manual,* is a complete user's guide to SCCS, and supersedes all previous versions. The following topics are covered:

- How to get started with SCCS.
- The scheme used to identify versions of text kept in an SCCS file.
- Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- Protection and auditing of SCCS files, including the differences between the use of SCCS by *individual* users on one hand, and *groups* of users on the other.

Neither the implementation of SCCS nor the installation procedure for SCCS are described here.

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of UNIX commands that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important when programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the *changes;* each set of changes is called a "delta."

This document, together with relevant portions of the *UNIX User's Manual,* is a complete user's guide to SCCS. This manual contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- *How Deltas Are Numbered:* How versions of SCCS files are numbered and named.

---

† UNIX is a trademark of Bell Laboratories.

- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.
- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

2. SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions (appearing in the *UNIX User's Manual*) should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID), composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File: the "admin" Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

We wish to give custody of this file to SCCS. The following *admin* command (which is used to *administer* SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin -ilang s.lang
```

All SCCS files *must* have names that begin with "s.", hence, "s.lang". The **-i** keyletter, together with its value "lang", indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The *admin* command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below. In the following examples, this warning message is not shown, although it may actually be issued by the various command.

The file "lang" should be removed (because it can be easily reconstructed by using the *get* command, below):

```
rm lang
```

2.3 Retrieving a File: the "get" Command

The command:

```
get s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
```

This means that *get* retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is created.

The above *get* command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command (see below), the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The -e keyletter causes *get* to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file,* that will be read by the *delta* command. The *get* command prints the same messages as before, except that the SID of the version to be created through the use of *delta* is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file "lang" may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

2.4 Recording Changes: the "delta" Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

comments? added more languages

Delta then reads the *p-file*, and determines what changes were made to the file "lang". It does this by doing its own *get* to retrieve the original version, and by applying *diff*(1)[1] to the original version and the edited version.

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", *delta* outputs:

1.2
2 inserted
0 deleted
5 unchanged

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

2.5 More about the "get" Command

As we have seen:

get s.lang

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

get s.lang

get −r1 s.lang

get −r1.2 s.lang

The numbers following the **−r** keyletter are SIDs (see Section 2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the *get* command:

get −e −r2 s.lang

Because release 2 does not exist, *get* retrieves the latest version *before* release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. *Get* then outputs:

1. All references of the form *name*(*N*) refer to item *name* in Section *N* of *UNIX User's Manual*.

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

we will see, by *delta's* output, that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6 The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the *help* command:

```
help co1
```

This produces the following output:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, *help* is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

3. HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the *release* number when making a delta, to indicate that a major change is being made. When this is

done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

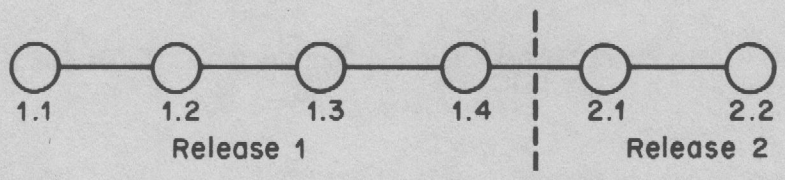

Figure 1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal *sequential* development of an SCCS file, in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a *branching* in the tree, in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of *four* components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

release.level.branch.sequence

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a *particular branch*. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

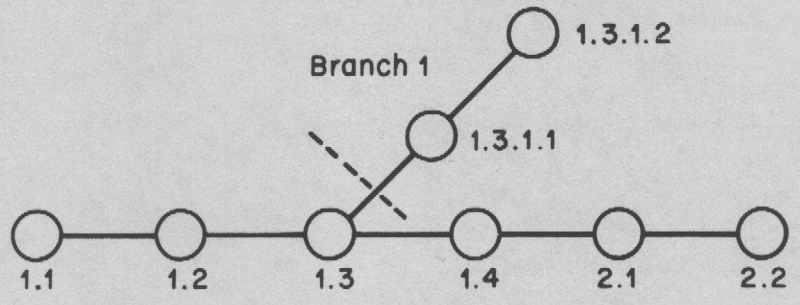

Figure 2. Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may

always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from *it*, all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

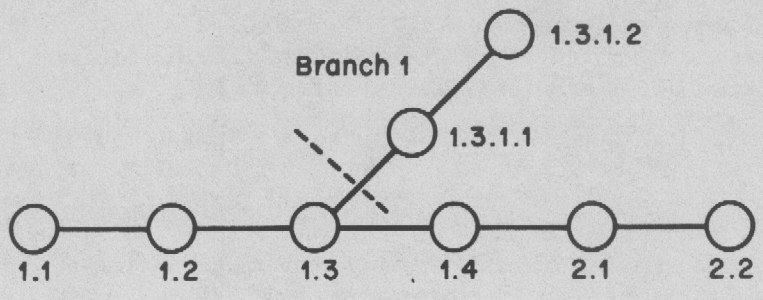

Figure 3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

4. SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below. SCCS commands accept two types of arguments: *keyletter* arguments and *file* arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (−), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable[2] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name "−" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the *find*(1) or *ls*(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

2. Because of permission modes (see *chmod*(1)).

All keyletters specified for a given command apply to *all* file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right.

Somewhat different argument conventions apply to the *help, what, sccsdiff,* and *val* commands (see Sections 5.5, 5.8, 5.9, and 5.11).

Certain actions of various SCCS commands are controlled by *flags* appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(1).

The distinction between the *real user* (see *passwd*(1)) and the *effective user* of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file,* which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file,* called the *z-file,* whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the *process number* of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files;* they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to the *help* command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of *that* file and to proceed with the next file, in order, if more than one file has been named.

5. SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *UNIX User's Manual,* and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

Because the commands *get* and *delta* are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.

admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the *get* command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

5.1 get

The *get* command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the *get* command is invoked, as discussed below.

The most common invocation of *get* is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated *g-file* (file "abc") is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, etc., and *not* for editing (i.e., *not* for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

5.1.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*, so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification (ID) keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the *g-file*. Thus, executing *get* on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by *get*, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by *get*, although the presence of the **i** flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see *get* (1).

5.1.2 Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of *get*.

The **−r** keyletter is used to specify an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get −r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get −r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a two- or four-component SID is specified as a value for the −r keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get −r3 s.abc
```

causes retrieval of the *trunk* delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, *get* retrieves the *trunk* delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get −r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get −r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The −t keyletter is used to retrieve the latest ("top") version in a particular *release* (i.e., when no −r keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get −r3 −t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3 Retrieval with Intent to Make a Delta

Specification of the −e keyletter to the *get* command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes *get* to check:

1. The *user list* (which is the list of *login* names and/or *group IDs* of users allowed to make deltas (see Section 6.2)) to determine if the login name or group ID of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. That the *release* (R) of the version being retrieved satisfies the relation:

   floor $\leq$ R $\leq$ ceiling

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.
3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.
4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the **j** flag in the SCCS file (multiple concurrent edits are described in Section 5.1.5).

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the **−e** keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a *writable g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are *not* substituted by *get* when the **−e** keyletter is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so that the message:

```
No id keywords (cm7)
```

is never output when *get* is invoked with the **−e** keyletter.

In addition, the **−e** keyletter causes the creation (or updating) of a *p-file*, which is used to pass information to the *delta* command (see Section 5.1.4).

The following is an example of the use of the **−e** keyletter:

```
get −e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the **−r** and/or **−t** keyletters are used together with the **−e** keyletter, the version retrieved for editing is as specified by the **−r** and/or **−t** keyletters.

The keyletters **−i** and **−x** may be used to specify a list (see *get*(1) for the syntax of such a list) of deltas to be *included* and *excluded*, respectively, by *get*. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be *not* applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. (Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*.) Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem

actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

☞ *The* **−i** *and* **−x** *keyletters should be used with extreme care.*

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of *get* with the **−e** keyletter, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by *get* executed with the **−e** keyletter. However, no processing related to the *p-file* takes place.

5.1.4 Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of *get* commands with the **−e** keyletter may be executed on the same file, provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed, see Section 5.1.5).

The *p-file* (which is created by the *get* command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":[3]

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of "get **−e**" causes the *creation* of the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* with a line containing the above information. Before updating, however, *get* checks that no entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved, unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a *writable g-file,* which is an SCCS error condition. In practice, such multiple executions are performed by different users,[4] so that this problem does not arise, since each user normally has a different working directory.

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get,* as well as the SID of the version to be eventually created by *delta,* as a function of the SID specified to *get.*

5.1.5 Concurrent Edits of the Same SID

Under normal conditions, *get*s for editing (**−e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, *delta* must be executed before a subsequent *get* for editing is executed at the same SID as the previous *get.* However, multiple concurrent edits (defined to be two or more *successive* executions of *get* for editing based on the same retrieved SID) *are* allowed if the **j** flag is set in the SCCS file. Thus:

---

3. Other information may be present, but is not of concern here. See *get*(1) for further discussion.
4. See Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files.

TABLE 1. Determination of New SID

Case	*SID Specified**	*−b Keyletter Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL +1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB +1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL +1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB +1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB +1).1
7.	R	—	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB +1).1
8.	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB +1).1
9.	R.L	no	No trunk successor	R.L	R.(L +1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB +1).1
11.	R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mB +1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS +1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB +1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S +1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB +1).1
16.	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB +1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB +1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

† The **−b** keyletter is effective only if the **b** flag (see *admin*(1)) is present in the file. In this table, an entry of "−" means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

```
get −e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get −e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of *delta*. In this case, a *delta* command corresponding to the first *get* produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the *delta* command corresponding to the second *get* produces delta 1.1.1.1.

5.1.6 Keyletters that Affect Output

Specification of the **−p** keyletter causes *get* to write the retrieved text to the standard output, rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get −p s.abc > arbitrary-file-name
```

The **−p** keyletter is particularly useful when used with the "!" or "$" arguments of the UNIX *send* (1C) command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file "compile" contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
~−s
~!get −p −rREL MOD
/*
//
```

will *send* the highest level of release 3 of file "s.abc". Note that the line "~−s", which causes *send*~(1C) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*~(1C) is to substitute "s.abc" for MOD and "3" for REL in the line "~!get −p −rREL MOD".

The **−s** keyletter suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the **−p** keyletter to "pipe" the output of *get*, as in:

```
get −p −s s.abc | nroff
```

The **−g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get −g −r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the **−g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get −e −g s.abc
```

The **−l** keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in *get*(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get −r2.3 −l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of "p" with the **−l** keyletter, as in:

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The **−g** keyletter may be used with the **−l** keyletter to suppress the actual retrieval of the text.

The **−m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **−n** keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The **−n** keyletter is most often used in a pipeline with *grep*(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get −p −n −s directory | grep pattern
```

If both the **−m** and **−n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the **−n** keyletter), followed by the line in the format produced by the **−m** keyletter. Because use of the **−m** keyletter and/or the **−n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the **−m** keyletter nor the **−n** keyletter may be specified together with the **−e** keyletter.

See *get*(1) for a full description of additional *get* keyletters.

5.2 delta

The *delta* command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the *delta* command requires the existence of a *p-file* (see Sections 5.1.3 and 5.1.4). *Delta* examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. *Delta* also performs the same permission checks that *get* performs when invoked with the **−e** keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file*, by comparing it (via *diff*(1)) with its own, temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing *delta*, because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed *get* with the **−e** keyletter more than once on the same SCCS file), the **−r** keyletter must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry[5]. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of *delta* is:

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new-line character. The user's response may be up to 512 characters long, with new-lines *not* intended to terminate the response escaped by "\".

---

5. The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.

If the SCCS file has a **v** flag, *delta* first prompts with:

MRs?

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR[6] numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?".

The **−y** and/or **−m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input:

```
delta −y"descriptive comment" −m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **−m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when *delta* is executed from within a *shell procedure* (see *sh* (1)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. This implies that if *delta* is invoked with more than one file argument, and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, *delta* outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by *delta* do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* is likely to find a description that differs from the user's perception. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If, in the process of making a delta, *delta* finds no ID keywords in the edited *g-file*, the message:

No id keywords (cm7)

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a *get* without the **−e** keyletter (recall that ID keywords are replaced by *get* in that case), or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

---

6. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*.[7] If there is only *one* entry in the *p-file*, then the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the **−n** keyletter is specified. Thus:

```
delta −n s.abc
```

will keep the *g-file* upon completion of processing.

The **−s** ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **−s** keyletter together with the **−y** keyletter (and possibly, the **−m** keyletter) causes *delta* neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **−p** keyletter. The format of this output is similar to that produced by *diff*(1).

5.3 admin

The *admin* command is used to *administer* SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

5.3.1 Creation of SCCS Files

An SCCS file may be created by executing the command:

```
admin −ifirst s.abc
```

in which the value ("first") of the **−i** keyletter specifies the name of a file from which the text of the *initial* delta of the SCCS file "s.abc" is to be taken. Omission of the value of the **−i** keyletter indicates that *admin* is to read the standard input for the text of the initial delta. Thus, the command:

```
admin −i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by *admin* as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the **−i** keyletter), the message is treated as an error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the **−i** keyletter.

---

7. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*, which is described in Section 4 above.

When an SCCS file is created, the *release* number assigned to its first delta is normally "1", and its *level* number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **−r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin −ifirst −r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **−i** keyletter.

5.3.2 Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (**−y** keyletter) and/or MR numbers[8] (**−m** keyletter) in exactly the same manner as for *delta*. If comments (**−y** keyletter) are omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**−m** keyletter), the **v** flag must also be set (using the **−f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* (see *sccsfile* (5)) in the SCCS file. Thus:

```
admin −ifirst −mmrnum1 −fv s.abc
```

Note that the **−y** and **−m** keyletters are only effective if a new SCCS file is being created.

5.3.3 Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* (see Section 6.2) may be initialized or changed through the use of the **−t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the **−t** keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin −ifirst −tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an *existing* SCCS file, the **−t** keyletter specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
admin −tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the **−t** keyletter as in:

```
admin −t s.abc
```

causes the *removal* of the descriptive text from the SCCS file.

The *flags* (see Section 6.2) of an SCCS file may be initialized and changed, or deleted through the use of the **−f** and **−d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See *admin* (1) for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords

8. The creation of an SCCS file may sometimes be the direct result of an MR.

contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The **−f** keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin −ifirst −fi −fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the *get* command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the %M% ID keyword.) Note that several **−f** keyletters may be supplied on a single invocation of *admin*, and that **−f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **−d** keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin −dm s.abc
```

removes the **m** flag from the SCCS file. Several **−d** keyletters may be supplied on a single invocation of *admin*, and may be intermixed with **−f** keyletters.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas (see Sections 5.1.3 and 6.2). This list is empty by default, which implies that *anyone* may create deltas. To add login names and/or group IDs to the list, the **−a** keyletter is used. For example:

```
admin −axyz −awql −a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **−a** keyletter may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The **−e** keyletter is used in an analogous manner if one wishes to remove ("erase") login names or group IDs from the list.

5.4 prs

Prs is used to print on the standard output all or parts of an SCCS file (see Section 6.2) in a format, called the output *data specification*, supplied by the user via the **−d** keyletter. The data specification is a string consisting of SCCS file *data keywords*[9] interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently being processed, and **:C:** is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see *prs*(1).

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs −d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

---

9. Not to be confused with *get ID keywords*.

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the **−r** keyletter. For example:

```
prs −d":F:: :I: comment line is: :C:" −r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the **−r** keyletter is *not* specified, the value of the SID defaults to the most recently created delta.

In addition, information from a *range* of deltas may be obtained by specifying the **−l** or **−e** keyletters. The **−e** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created *earlier*. The **−l** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created *later*. Thus, the command:

```
prs −d:I: −r1.4 −e s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command:

```
prs −d:I: −r1.4 −l s.abc
```

may produce:

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for *all* deltas of the SCCS file may be obtained by specifying both the **−e** and **−l** keyletters.

5.5 help

The *help* command prints explanations of SCCS commands and of messages that these commands may print. Arguments to *help,* zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. *Help* has no concept of *keyletter* arguments or *file* arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will *not* terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces:

ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
rmdel −rSID name ...

5.6 rmdel

The *rmdel* command is provided to allow *removal* of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The **−r** keyletter, which is mandatory, is used to specify the *complete* SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel −r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, *rmdel* checks that the *release* number (R) of the given SID satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$

Rmdel also checks that the SID specified is *not* that of a version for which a *get* for editing has been executed and whose associated *delta* has not yet been made. In addition, the login name or group ID of the user must appear in the file's *user list*, or the *user list* must be empty. Also, the release specified can not be *locked* against editing (i.e., if the **l** flag is set (see *admin*(1)), the release specified *must* not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7 cdc

The *cdc* command is used to *change* a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the *rmdel* command, except that the delta to be processed is *not* required to be a leaf delta. For example:

```
cdc −r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The *new* commentary is solicited by *cdc* in the same manner as that of *delta*. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing *cdc* and the time of its execution.

Cdc also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

5.8 what

The *what* command is used to find identifying information within *any* UNIX file whose name is given as an argument to *what*. Directory names and a name of "−" (a lone minus sign) are *not* treated specially, as they are by other SCCS commands, and no *keyletters* are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword (see *get*(1)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), new-line, or (non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char id[] "%Z%%M%:%I%";
```

and then the command:

```
get -r3.4 s.prog.c
```

is executed, and finally the resulting *g-file* is compiled to produce "prog.o" and "a.out", then the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
   prog.c:3.4
prog.o:
   prog.c:3.4
a.out:
   prog.c:3.4
```

The string searched for by *what* need not be inserted via an ID keyword of *get;* it may be inserted in any convenient manner.

5.9 sccsdiff

The *sccsdiff* command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **−r** keyletter, whose format is the same as for the *get* command. The two versions *must* be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the *pr*(1) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "−" (a lone minus sign) are *not* acceptable to *sccsdiff*.

The differences are printed in the form generated by *diff*(1). The following is an example of the invocation of *sccsdiff:*

```
sccsdiff -r3.4 -r5.6 s.abc
```

5.10 comb

Comb generates a *shell procedure* (see *sh*(1)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is *not* recommended that *comb* be used as a matter of routine; its use should be restricted to a *very* small number of times in the life of an SCCS file.

In the absence of any keyletters, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **−c** keyletter specifies a *list* (see *get*(1) for the syntax of such a list) of deltas to be preserved. All other deltas are discarded.

The **−s** keyletter causes the generation of a shell procedure, which, when run, produces *only* a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that *comb* be run with this keyletter (in addition to any others desired) *before* any actual reconstructions.

It should be noted that the shell procedure generated by *comb* is *not* guaranteed to save any space. In fact, it is possible for the reconstructed file to be *larger* than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

5.11 val

Val is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

Val checks for the existence of a particular delta when the SID for that delta is *explicitly* specified via the **−r** keyletter. The string following the **−y** or **−m** keyletter is used to check the value set by the **t** or **m** flag respectively (see *admin*(1) for a description of the flags).

Val treats the special argument "−" differently from other SCCS commands (see Section 4). This argument allows *val* to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end-of-file. This capability allows for one invocation of *val* with different values for the keyletter and file arguments. For example:

```
val −
−yc −mabc s.abc
−mxyz −ypl1 s.xyz
```

first checks if file "s.abc" has a value "c" for its *type* flag and value "abc" for the *module name* flag. Once processing of the first file is completed, *val* then processes the remaining files, in this case "s.xyz", to determine if they meet the characteristics specified by the keyletter arguments associated with them.

Val returns an 8-bit code; each bit set indicates the occurrence of a specific error (see *val*(1) for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the **−s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

6. SCCS FILES

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

6.1 Protection

SCCS relies on the capabilities of the UNIX operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the *release lock* flag, the *release floor* and *ceiling* flags, and the *user list* (see Section 5.1.3).

New SCCS files created by the *admin* command are given mode 444 (read only). It is recommended that this mode *not* be changed, as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755, which allows only the *owner* of the directory to modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files (see Section 6.3). The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

SCCS files must have only *one* link (name), because the commands that modify SCCS files do so by creating a copy of the file (the *x-file*, see Section 4) and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files[10]. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel* and *cdc* commands.

The interface program must be owned by the SCCS administrator, and must have the *set user ID on execution* bit on (see *chmod*(1)), so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to *inherit* the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose *login* names or *group* IDs are in the *user list* for that file (but who are *not* its owner) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*. The project-dependent interface program, as its name implies, must be custom-built for each project.

---

10. Previously, the UNIX system allowed only 256 unique user IDs. Thus, several users often had to share user IDs and, therefore, file permissions. The current UNIX system allows 65,536 unique user IDs; it is recommended that each user have a unique user ID.

6.2 Format

SCCS files are composed of lines of ASCII text[11] arranged in six parts, as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as its type, its SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile* (5); the *checksum* is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various UNIX commands, such as *ed*(1), *grep*(1), and *cat*(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when it is desired to simply "look" at the file.

☞ *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

6.3 Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file, or portions of it (i.e., one or more "blocks") can be destroyed. SCCS commands (like most UNIX commands) issue an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (possibly by having lost one or more blocks, or by having been modified with, for example, *ed*(1)). *No* SCCS command will process a corrupted SCCS file except the *admin* command with the **−h** or **−z** keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the *admin* command with the **−h** keyletter on all SCCS files:

```
admin −h s.file1 s.file2 ...
   or
admin −h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not

11. Previous versions of SCCS up to and including Version 3 used non-ASCII files. Therefore, files created by earlier versions of SCCS are incompatible with the current version of SCCS.

detect *missing* files. A simple way to detect whether *any* files are missing from a directory is to periodically execute the *ls*(1) command on that directory, and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX operations group and request that the file be restored from a backup copy. In the case of minor damage, repair through use of the editor *ed*(1) may be possible. In the latter case, after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

January 1981