An Introduction to the Source Code Control System

Eric Allman
Project Ingres
University of California at Berkeley

This document gives a quick introduction to using the Source Code Control System (SCCS). The presentation is geared to programmers who are more concerned with what to do to get a task done rather than how it works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section on "Further Information".

This is a working document. Please send any comments or suggestions to eric@Berkeley.Edu.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file". There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

This is version 1.21 of this document. It was last modified on 12/5/80.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes — equivalent to removing your changes later.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a ''release'' number and a ''level'' number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form **%*x*%**, where *x* is an upper case letter. For example, **%I%** is the SID of the latest delta applied, **%W%** includes the module name, SID, and a mark that makes it findable by a program, and **%G%** is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is ''%W%'' or whatever).

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin −i$i $i
        mv $i save/$i
end
```

This will put the named files into s-files in the subdirectory ''SCCS'' The files will be removed from the current directory and hidden away in the directory ''save'', so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory ''save''.

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print ''No Id Keywords (cm7)'', which is a warning message only.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for ''comments?'' before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like ''fixed compilation problem in previous delta'' or ''fixed botch in 1.3''. However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

---

[2]Actually, the SID of the final delta applied was 1.1.

[3]The ''edit'' command is equivalent to using the −e flag to *get*, as:

```
sccs get −e prog.c
```

Keep this in mind when reading other documentation.

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an ''install'' entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **−b** flag to ignore ''branches'' (alternate versions, described later) and the **−u** flag to only give files being edited by you. The **−u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info −ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c 1.2	08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string ''@(#)'' is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with ''@(#)''. This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
		prog.c	1.2	08/29/80
/usr/bin/prog:
		prog.c	1.1	02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%	%G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W%    %G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because ‘‘SccsId’’ is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID’s consistent across files

With some care, it is possible to keep the SID’s consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit −r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit −r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get −r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don’t know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the **−c** (cutoff) flag. For example,

```
sccs get −c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get −c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit −x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit −x1.3−1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit −x1.3−1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using **−x** (or **−i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of ‘‘a set of changes’’) can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prt command

When you created a delta, you presumably gave a reason for the delta to the ‘‘comments?’’ prompt. To print out these comments later, use:

```
sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2   80/08/29 12:35:31   bill   2   1   00005/00003/00084
removed "-q" option

D 1.1   79/02/05 00:19:31   eric   1   0   00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get −m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get −m −p prog.c | grep ´^1.3´
```

The **−p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the ‘‘diff’’ flags can be used. To pass the **−c** flag, use **−C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The ‘‘deledit’’ command is equivalent to ‘‘delget’’ except that the ‘‘edit’’ command is used instead of the ‘‘get’’ command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix −r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The −r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The −d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs −d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file ‘‘/usr/src/cmd/SCCS/who.c’’. The file ‘‘who.c’’ will always be created in your current directory regardless of the value of the −d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[6]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get –k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin –z prog.c
```

11. Using the Admin Command

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the **–f** flag. For example:

```
sccs admin –fd1 prog.c
```

sets the ''d'' flag to the value ''1''. This flag can be deleted by using:

```
sccs admin –dd prog.c
```

The most useful flags are:

b Allow branches to be made using the –b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.

y The ''type'' of the module. Actually, the value of this flag is unused by SCCS except that it replaces the **%Y%** keyword.

The **–t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **–t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive

[6]Or given up and decided to start over.

text, use ‘‘prt −t’’.

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a ‘‘branch.’’ Normally deltas continue in a straight line, each depending on the delta before. Creating a branch ‘‘forks off’’ a version of the program.

The ability to create branches must be enabled in advance using:

```
sccs admin −fb prog.c
```

The **−fb** flag can be specified when the SCCS file is first created.

12.1. Creating a branch

To create a branch, use:

```
sccs edit −b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get −r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don’t want to lose. The commands:

```
sccs edit −i1.5.1.1−1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs −d../xyz edit prog.c
```

When using the old version, be sure to use the **−b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info −b
```

when in the directory ‘‘xyz’’.

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the **–i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called ''all'' and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all files from the current directory that can be regenerated from SCCS files.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= –i –s

prog: prog.o
	$(CC) $(LDFLAGS) –o prog prog.o
prog.o: prog.c prog.h

example: example.o
	$(CC) $(LDFLAGS) –o example example.o
example.o: example.c

.DEFAULT:
	sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS=  prog.c prog.h example.c

LDFLAGS= −i −s

prog: prog.o
        $(CC) $(LDFLAGS) −o prog prog.o
prog.o: prog.h

example: example.o
        $(CC) $(LDFLAGS) −o example example.o

sources: $(SRCS)
$(SRCS):
        sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say ‘‘make sources’’, and (3) the makefile is less likely to do confusing things since it won’t try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn’t know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.c d.s x.h y.h z.h
TARG= /usr/lib

# programs
GET=   sccs get
REL=
AR=    −ar
RANLIB=        ranlib

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a

install: lib.a
        sccs check
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@

print: sources
        pr *.h *.[cs]
clean:
        rm −f *.o
        rm −f core a.out $(LIB)
```

The ‘‘$(REL)’’ in the get can be used to get old versions easily; for example:

```
make b.o REL=−r1.3
```

The *install* entry includes the line ‘‘sccs check’’ before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS=	a.o b.o c.o d.o
SRCS=	a.c b.c c.y d.s x.h y.h z.h

GET=	sccs get
REL=

a.out: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
	$(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
	touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with ‘‘sccs’’. This list is not exhaustive; for more options see *Further Information*.

get Gets files for compilation (not for editing). Id keywords are expanded.

- −r*SID* Version to get.
- −p Send to standard output rather than to the actual file.
- −k Don’t expand id keywords.
- −i*list* List of deltas to include.
- −x*list* List of deltas to exclude.
- −m Precede each line with SID of creating delta.
- −c*date* Don’t apply any deltas created after *date*.

edit Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

- −r*SID* Same as *get*. If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID*.
- −b Create a branch.
- −i*list* Same as *get*.
- −x*list* Same as *get*.

delta Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.

unedit Remove a file that has been edited previously without merging the changes into the s-file.

prt Produce a report of changes.

- −t Print the descriptive text.
- −e Print (nearly) everything.

info Give a list of all files being edited.

- −b Ignore branches.
- −u[*user*] Ignore files not being edited by *user*.

check Same as *info*, except that nothing is printed if nothing is being edited and exit status is returned.

tell Same as *info*, except that one line is produced per file being edited containing only the file name.

clean Remove all files that can be regenerated from the s-file.

what Find and print id keywords.

admin Create or set parameters on s-files.

- −i*file* Create, using *file* as the initial contents.
- −z Rebuild the checksum in case the file has been trashed.
- −f*flag* Turn on the *flag*.
- −d*flag* Turn off (delete) the *flag*.
- −t*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or ‘‘design & implementation’’ documents to insure they get distributed with the s-file.

Useful flags are:

b	Allow branches to be made using the −b flag to *edit.*
d*SID*	Default SID to be used on a *get* or *edit*.
i	Cause ‘‘No Id Keywords’’ error message to be a fatal error rather than a warning.
t	The module ‘‘type’’; the value of this flag replaces the **%Y%** keyword.

fix Remove a delta and reedit it.

delget Do a *delta* followed by a *get*.

deledit Do a *delta* followed by an *edit*.

2. Id Keywords

%Z% Expands to ‘‘@(#)’’ for the *what* command to find.

%M% The current module name, *e.g.*, ‘‘prog.c’’.

%I% The highest SID applied.

%W% A shorthand for ‘‘%Z%%M% <tab> %I%’’.

%G% The date of the delta corresponding to the ‘‘%I%’’ keyword.

%R% The current release number, *i.e.*, the first component of the ‘‘%I%’’ keyword.

%Y% Replaced by the value of the **t** flag (set by *admin*).