ULTRIX

digital

Guide to the Source Code Control System

Order Number: AA–ME84B–TE

ULTRIX

Guide to the Source Code Control System

Order Number: AA-ME84B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This guide introduces the `sccs`(1) preprocessor, then describes how to use it with related sccs commands in a software project.

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 Overview to the Source Code Control System

2 Using SCCS

3 Using SCCS on a Project

Figures

About This Manual

The objective of this guide is to provide you with information on the Source Code Control System (SCCS), and to explain how to use its commands and options.

Audience

This manual is written for the person who is responsible for archiving source files. This person may also be responsible for managing and maintaining an ULTRIX operating system. It assumes that this individual is familiar with some ULTRIX commands and with an editor such as `vi` or `ed`.

Organization

This guide consists of three chapters and an index. The chapters are:

Chapter 1: Overview to the Source Code Control System
Provides an overview of the Source Code Control System (SCCS), explaining SCCS terminology and SCCS keywords.

Chapter 2: Using SCCS
This chapter describes how to create SCCS files, change SCCS files, restore old versions of SCCS files, and audit changes to SCCS files. It also explains how to combine commonly used SCCS keywords and contains a quick SCCS keyword reference.

Chapter 3: Using SCCS on a Project
This chapter explains the various problems and situations you may encounter when using SCCS on a major project.

Related Documents

You should use this document in conjunction with the *ULTRIX Reference Pages*.

Conventions

The following conventions are used in this guide:

special — In text, each mention of a specific command, option, partition, pathname, directory, or file is presented in this type.

command(x)	In text, cross-references to the command documentation include the section number in the reference manual where the commands are documented. For example: See the cat(1) command. This indicates that you can find the material on the cat command in Section 1 of the *ULTRIX Reference Pages*.
literal	In syntax descriptions, this type indicates terms that are constant and must be typed just as they are presented.
italics	In syntax descriptions, this type indicates terms that are variable.
[ ]	In syntax descriptions, square brackets indicate terms that are optional.
. . .	In syntax descriptions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`function`	In function definitions, the function itself is shown in this type. The function arguments are shown in italics.
UPPERCASE	The ULTRIX system differentiates between lowercase and uppercase characters. Enter uppercase characters only where specifically indicated by an example or a syntax line.
`example`	In examples, computer output text is printed in this type.
`example`	In examples, user input is printed in this bold type.
%	This is the default user prompt in multiuser mode.
#	This is the default superuser prompt.
>>>	This is the console subsystem prompt.
.	In examples, a vertical ellipsis indicates that not all of the
.	lines of the example are shown.
.	

Overview to the Source Code Control System 1

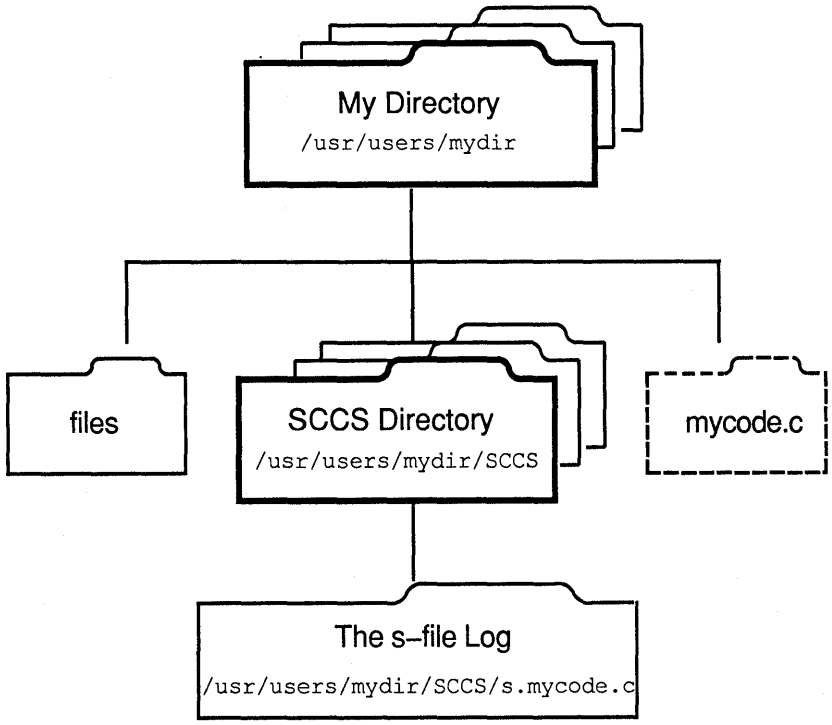
The Source Code Control System (SCCS) is a source management system, which maintains a record of each set of changes made to the files, including why the changes were made, who made them, and when they were made. Old versions can be recovered, and different versions can be maintained simultaneously. Also, in projects with more than one person, SCCS ensures that two people are not editing the same file at the same time.

The SCCS system has two levels of operation, traditional SCCS commands and a preprocessor for these commands, `sccs`(1). This manual discusses the `sccs` preprocessor and how you use it to create, administer, and use SCCS libraries.

1.1 Introduction to SCCS

The following figure illustrates the SCCS directory structure:

Figure 1-1: SCCS Directory Structure



ZK–0146U–R

All versions of your program, plus the log and other information, are kept in a file called the `s-file`. The s-file is basically a log file that allows SCCS to save your file by saving the origional version and all the changes that have been made, along with comments describing why changes were made. There are three major operations that can be performed on the s-file:

- Getting 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 only intended for compilation, printing, and so forth; it is not intended to be edited or changed in any way. Thus, any changes made to a file retrieved in this way may be lost.

- Getting a file for editing.

  This operation also retrieves a version of the file from the s-file, but this version of the file can be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

- Merging a file back into the s-file.

  This is the companion operation to getting a file for editing. After you have edited a file, SCCS assigns a new version number to the file and saves any comments you make explaining the reasons for your changes.

1.2 SCCS Terminology

Before attempting to place files under SCCS, you should make sure that you understand the following SCCS terms:

- The s-file
- Deltas
- SCCS IDs
- ID keywords

The following sections describe these terms.

1.2.1 The 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; 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. The s-file also includes some header information for each version of the file, as well as any comments given by the person who created each new version explaining why they made the changes.

1.2.2 Deltas

Each set of changes to the s-file is called a `delta`. Although technically a delta only includes the latest changes, in practice it is quite common for each delta to be made with respect to all the deltas that have occurred before. This matches normal use, where the previous changes are not saved at all, so that all changes are automatically based on all other changes that have happened through the file's history. However, it is possible to get a version of the file that has selected deltas removed. This is equivalent to removing changes at a later time.

1.2.3 SCCS IDs (Version Numbers)

A Source Code Control System ID (SID) represents a delta and is usually 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 to the file or project is being made.

Because SCCS normally applies all past deltas to a file, the SID of the final delta that is applied to the file can be used to represent a version number of the file as a whole.

1.2.4 ID Keywords

When you get a version of a file and you intend to compile and install it (as opposed to editing it), SCCS expands some special keywords, called ID keywords, inline. You can use these ID keywords to include the current version number of the file or other information in the file. ID keywords are of the following form:

%x%

The variable *x* is an uppercase letter. For example, `%I%` is the SID of the latest delta applied, where as `%W%` includes the module name, the SID, and a mark that allows the SCCS program to find it, and `%G%` is the date of the latest delta applied.

When you get a file for editing, SCCS does not expand the ID keywords; this is so that after you put them back into the s-file, SCCS will expand them automatically on each new version. If they were expanded accidentally, your file would appear to be the same version from that point on, which would defeat the purpose of including the keywords. Also, if you installed a version of the program without expanding the ID keywords, it would be impossible to tell what version you are working on because the `sccs` program would only have the SID of `%W%`.

The following list summarizes the information provided by of the ID keywords. Refer to `admin(1)` in the *ULTRIX Reference Pages* for more information on keywords.

%Z% Expands to `@(#)` for the `what` command to find. This is a special string that signals the beginning of an keyword.

%M% The current module name, for example, `prog.c`.

%I% The highest SID applied.

%W% A shorthand for the following: `%Z%%M% <tab> %I%`.

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

%R% The current release number, for example, the first component of the `%I%` keyword.

%Y% Replaced by the value of the **–t** flag (set using the `admin` command).

1.3 For More Information

To assist you in maintaining existing shell scripts or programs that may use the component modules directly, the *ULTRIX Reference Pages* contain descriptions of the following SCCS modules:

`admin(1)`	`cdc(1)`	`comb(1)`
`delta(1)`	`edit(1)`	`get(1)`
`prs(1)`	`rmdel(1)`	`sccs(1)`
`sccsdiff(1)`	`sccshelp(1)`	`unget(1)`
`val(1)`	`vc(1)`	`what(1)`
`sccsfile(5)`		

Using SCCS 2

This chapter describes how to use SCCS. It describes some of the most commonly used SCCS commands, such as `admin`, `get`, and `delta`, explains combined SCCS commands, and also contains a quick reference of SCCS commands. The specific topics discussed are:

- Creating SCCS files
- Getting files for compilation
- Modifying SCCS files (creating deltas)
- Restoring old versions
- Auditing changes
- Combining SCCS keywords
- Summary of SCCS commands

2.1 Creating SCCS Files

To put C language source and header files into SCCS format, you need to use the `admin` and `sccs` commands. For example, To set up a file named `prog8.c` under SCCS in a directory called `/project`, here are the commands:

```
# cd /projects
# cp ~joe/prog.8c .
# mkdir SCCS
# sccs admin -iprog8.c prog8.c
```

This example assumes the version of the `prog8.c` file being put under SCCS is in user `joe`'s home directory.

If you have several files to place under SCCS, it might be easier for you to use a simple `csh` script:

```
# mkdir SCCS save
# foreach i (prog[1-5].c)
? sccs admin -i$i $i
? mv $i save/$i
? end
```

This example shows how to place the files `prog1.c`, `prog2.c`, `prog3.c`, `prog4.c`, and `prog5.c` into the SCCS directory. As SCCS places the files into the SCCS directory, it removes the files from the current directory and places them in the directory `save`. Next, you should pull the files from the SCCS directory back into the current directory. This is described in Section 2.2. When you are sure that SCCS has correctly created the s-files, 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` prints the following message, informing you that there are no ID keywords:

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

2.2 Getting Files for Compilation

To get a copy of the latest version of a file, type the following command:

```
# sccs get prog1.c
```

When SCCS gets a file, it prints a message similar to the following:

```
1.1
87 lines
```

This message means that version 1.1 of the file was retrieved (or more accurately, that the SID of the most recent delta applied was 1.1) and that the file contains 87 lines. The file `prog1.c` is placed in your working directory. Furthermore, the mode of the file is set to read-only to remind you that you are not supposed to make changes to this copy of the file.

Note

This copy of the file should not be changed, because SCCS is unable to merge the changes back into the s-file. If you do make changes, they will not be under source control. Thus, the next time someone uses the `get` command to get a copy of that file from SCCS, your changes will be lost.

2.3 Modifying SCCS Files (Creating Deltas)

This section describes how to obtain a copy of an SCCS source file for editing and how to merge the changes into the file.

2.3.1 Getting a Copy to Edit

To obtain a copy of source file to edit, use the following command, which gets a copy of the file with read-write permissions:

```
# sccs edit prog1.c
```

This `sccs` command is equivalent to the `get` command with the **–e** flag, and produces the same results. For example:

```
# sccs get -e prog1.c
```

The SCCS edit command prints the same messages that the SCCS `get` command prints, with the following addition:

```
New delta 1.2
```

When you have a copy of the file that you want to edit, you can edit it using a standard text editor. For example:

```
# vi prog1.c
```

2.3.2 Merging the Changes Back into the s-file

When you have made the desired changes to a file, you can put your changes back into the SCCS file using the `delta` command:

```
# sccs delta prog1.c
```

SCCS then prompts for comments that it will store in the s-file:

```
Comments? (^D to end)
```

You can now enter one or more lines of text to describe the changes that you have made and why. When you have finished entering your comments, press CTRL/D at the beginning of a new line. SCCS will then print a series of messages similar to the following:

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

These messages indicate that delta 1.2 was created, and that SCCS inserted five lines in the file, removed three lines, and left 84 lines unchanged. Changes to a line are counted as a line deleted and a line inserted. SCCS then removes the changed file (in the example, `prog1.c`). You can obtain an updated copy of the file using the `get` command.

2.3.3 When to Make Deltas

It is probably not necessary to make a delta before every recompilation or test; if you do, you tend to get a lot of deltas with comments such as:

```
Fixed compilation problem in previous delta
```

```
Fixed botch in 1.3
```

However, you should make a delta for each file before you install a module for general use.

When files are out for edit, you should 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, make deltas for each file that is out for edit, use the `get` command to retrieve updated copies, and then recompile all the files.

2.3.4 Getting Information About SCCS Files

There are several SCCS commands that you can use to get specific information about SCCS files. These are:

- sccs info
- sccs check
- sccs delta 'sccs tell'
- sccs what

2.3.4.1 Using sccs info – To find out what files are currently being edited and the names of the users who have taken the files out for editing, type the following command:

```
# sccs info
```

2.3.4.2 Using sccs check – The `sccs check` command is nearly equivalent to the `info` command:

```
# sccs check
```

The `check` command, however, prints no message if nothing is being edited, and returns nonzero exit status if anything is being edited. You can use the `check` command in an `install` entry in a makefile to abort the install if any of the files are still out for edit.

2.3.4.3 Using sccs delta 'sccs tell' – If you know that all the files being edited should have deltas made, you can use the following command line:

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

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

All of these commands take a **–b** flag to ignore branches (alternate versions, described in Chapter 3) and the **–u** flag to print only the names of files being edited by you. The **–u** flag takes an optional *user* argument, which prints only the names of files being edited by that user. For example, the following command prints a list of files being edited by user `john`:

```
# sccs info -ujohn
```

2.3.4.4 Using sccs what – To find out what version of a program is being run on your system, use the following command line:

```
# sccs what prog1.c /usr/bin/prog
```

The `what` command searches through the specified file and prints all strings it finds that begin with `@(#)`. This works on all types of files, including binary files and libraries. For example, the command in the preceding example might result in the following messages being printed:

```
prog1.c:
     prog1.c   1.2  08/29/87
/usr/bin/prog:
     prog1.c   1.1  02/01/88
```

These results indicate that the source code in `prog1.c` will not compile into the same version as the existing binary file in `/usr/bin/prog`.

2.3.5 Inserting ID Keywords

The `get` command can insert ID keywords into your file and expand them automatically. For example, consider a line such as the following:

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

This would be replaced with something like this:

```
static char SccsId[] = "@(#)prog1.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 that signals the beginning of an SCCS ID keyword.

You can insert ID keywords anywhere, including in comments, but ID keywords that are compiled into the object module are especially useful, because they allow you to determine what version of the object is being run, as well as the source.

When you put ID keywords into header files, it is important to assign them to different variables. For example, you might include the following line in the file `access.h`:

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

You might also include the following line in the file `opsys.h`:

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

Note

If you had assigned the keywords to the same variables, you would have received errors when you compiled the program, because `SccsId` would be 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 convenient to include ID keywords in comments in header files.

2.3.6 Creating New Releases

To create a new release of a program, you can specify the release number you want to create in the `edit` command line. For example, the following command causes the next delta to be in release 2, that is, the next delta will be numbered 2.1:

```
# sccs edit -r2 prog1.c
```

Thus, future deltas will automatically be in release 2. To change the release number of an entire system, use the following command:

```
# sccs edit -r2 SCCS
```

2.4 Restoring Old Versions

This section describes how to revert to and restore different versions of an SCCS file and how to delete SCCS deltas.

2.4.1 Reverting to Old Versions

Suppose that after delta 1.2 was stable, you made and released delta 1.3. But assume this delta introduced a bug, so you made delta 1.4 to correct it. But delta 1.4 still contained bugs, and you decided you wanted to go back to the old version (delta 1.2). You could revert to delta 1.2 by specifying the SID with the `get` command:

```
# sccs get -r1.2 prog1.c
```

This creates a version of `prog1.c` that is delta 1.2, which you can then reinstall so that work can proceed.

In some cases you might not know the SID of the delta that you want. 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, the following command retrieves whatever version

was current as of July 22, 1988 at 12:00 noon:

```
# sccs get -c880722120000 prog1.c
```

You can strip off trailing components (which then default to their highest legal value), and insert punctuation in the obvious places. For example, the date in the example could also be specified as follows:

```
# sccs get -c"88/07/22 12:00:00" prog1.c
```

2.4.2 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 with the SCCS `rmdel` command:

```
# sccs edit rmdel 1.3 prog1.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 also exclude a range of deltas using a dash. For example, if you want to get rid of deltas 1.3 and 1.4, you can use the following command, which excludes all deltas from 1.3 to 1.4:

```
# sccs edit rmdel 1.3-1.4 prog1.c
```

Alternatively, you can use the following command to exclude a range of deltas from 1.3 to the current highest delta in release 1:

```
# sccs edit rmdel 1.3-1 prog1.c
```

Because you can exclude each delta at will, it is most useful to put each semantically distinct change into its own delta.

2.5 Auditing Changes

This section describes several SCCS commands that you can use to monitor changes made to the files. These commands include:

- The `prs` command – print delta comments
- The `get` command options – identify changes by printing comments
- The `diffs` and `sccsdiff` commands – identify changes

2.5.1 Printing delta Comments

The `prs` command enables you to print comments associated with the `deltas` of an SCCS file. To print `delta` comments for the `prog1.c` file, type the following command:

```
# sccs prs prog1.c
```

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

```
D 1.288/08/29 12:35:31   bill 2    1    00005/00003/00084
removed "-q" option

D 1.187/02/05 00:19:31   eric 1    0    00087/00000/00000
date and time created 86/06/10 00:19:31 by eric
```

2.5.2 Identifying Changes by Printing Comments

To find out why someone inserted lines in a file for example `prog1.c`), you can get a copy of the file that has each line preceded by the SID that created it, by typing the following command:

```
# sccs get -m prog1.c
```

You can then find out what a particular delta did by printing the comments the using the `prs` command.

To find out what lines are associated with a particular delta (for example, 1.3), use the following command:

```
# sccs get -m -p prog1.c | grep '^1.3'
```

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

2.5.3 Identifying the Changes

When you are editing a file such as `prog1.c`, you can find out what changes you have made using the following command:

```
# sccs diffs prog1.c
```

You can use most of the flags described in the `sccsdiff` command. To pass the **–c** flag, use **–C .**

To compare two versions that are in different deltas, use the `sccsdiff` command. For example, to see the differences between delta 1.3 and delta 1.6 of the file `prog1.c`, type the following command:

```
# sccs sccsdiff -r1.3 -r1.6 prog1.c
```

2.6 Combining SCCS keywords

There are several SCCS command sequences that many people use frequently. SCCS provides special compound commands and options to make it easier for you to execute these sequences. These commands are:

- `sccs delget`
- `sccs fix`
- `sccs unedit`
- `sccs -d`

These commands and their options are described in the following sections.

2.6.1 Using the delget Command

You often need to make a delta to some file. for example `prog1.c`, and then get that file. You can type the following command line to do this quickly:

```
# sccs delget prog1.c
```

This single command produces the same results as do the following two commands:

```
# sccs delta prog1.c
# sccs get prog1.c
```

The `deledit` command is equivalent to `delget` except that you use the `edit` command instead of the `get` command.

2.6.2 Using the fix Command

Frequently, there are small bugs in deltas, such as compilation errors, for which there is no reason to maintain an audit trail. To replace a delta in a file, type the following command:

```
# sccs fix -r1.4 prog1.c
```

This command gets a copy of delta 1.4 of the file `prog1.c` for you to edit and then deletes delta 1.4 from the SCCS file. When you do perform a delta of `prog1.c`, it will become delta 1.4 again.

Note

You must specify the `-r` flag, and the specified delta must be a leaf delta. There should be no deltas created subsequent to the creation of that delta. Also, the `fix` command does not work on an s-file with no deltas (–r1.1).

2.6.3 Removing Edit Reservations from a File

After you have edited a file such as `prog1.c`, if you find that you do not want to keep the edits, you can retain the previous version with this command:

```
# sccs unedit prog1.c
```

2.6.4 Using the –d Flag

If your files are in a working directory that is not the same directory that contains the SCCS files, you may be able to reduce the number of keystrokes by using a shell alias. For example, if you are in the `csh` shell, you can type the following command to create an alias named `syssccs`:

```
# alias syssccs sccs -d/usr/src
```

Once set, that alias allows you to issue commands such as the following:

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

This command looks for the file `/usr/src/cmd/SCCS/who.c`. The file `who.c` is always created in your current directory, regardless of the value of the `-d`flag.

2.7 SCCS Command Summary

You should always use the following commands as options to the `sccs command`. This list is not exhaustive. For more options, see the *ULTRIX Reference Pages*.

get — Get 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 — Do not 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*	Do not apply any deltas created after *date*.
edit	Get 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 that is out for 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.	
prs	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*.
		Useful flags that can be used with the **–f** flag are:
		b — Allow branches to be made using the **–b** flag with the `edit` command.
		d*SID* — Default SID to be used with the `get` or `edit` commands.
		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.
	–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 and implementation documents to ensure they get distributed with the s-file.
fix	Remove a delta and reedit it.	
delget	Do a `delta` command followed by a `get` command.	
deledit	Do a `delta` command followed by an `edit` command.	

Using SCCS on a Project 3

This chapter describes how to use SCCS while working on a large project with several team members. The topics discussed are:

- Setting up an SCCS directory for a Project
- Deciding when to edit a file
- Recovering a corrupted edit file
- Using SCCS with the `admin` command
- Maintaining different versions (branches)
- Using SCCS with the `make` command

3.1 Setting Up an SCCS Directory for a Project

This section describes the preferred method for setting up an SCCS directory and populating it with files for use on a project-wide basis. By following the steps outlined in this section, you can eliminate the need for superuser intervention in maintaining the project files.

A good approach for placing the files of any project under source control is first to assign a project librarian the responsibility of maintaining those files. The project librarian can then set up and maintain the SCCS directories and files.

It is wise to set up a generic account for the project librarian. By doing so, any user who has the password can act as the project librarian. This feature allows the project librarian duties to be passed from one individual to another.

The following example makes these assumptions:

- The project files are to be placed under source control in the directory `/project`, which has already been created and which has read/write permissions for the members of the project
- The project librarian has the account `projlib` and owns the `/project` directory
- The original versions of the project files are in user `joe`'s home directory

With the preceding assumptions made, here are the commands the project librarian should take to set up the SCCS directory for the project:

```
# cd /project
# mkdir SCCS
# mkdir bin
# chmod 775 SCCS
# chmod 775 bin
# cp /usr/ucb/sccs /project/bin/sccs.proj
# chmod 4555 /project/bin/sccs.proj
# ls -lg bin
```

```
total 50
-r-sr-xr-x  1 projlib     staff       51200 Aug 23 11:24 sccs.proj
# cp ~joe/file /project
# bin/sccs.proj create file
file:
No id keywords (cm7)
1.1
124 lines
No id keywords (cm7)
# ls -al *
   5 -rw-r--r--  1 projlib         4442 Aug 25 16:12 ,file
   5 -r--r--r--  1 projlib         4442 Aug 25 16:13 file

SCCS:
total 8
   1 drwxr-xr-x  2 projlib          512 Aug 25 16:13 .
   1 drwxr-xr-x  4 projlib          512 Aug 25 16:13 ..
   5 -r--r--r--  1 projlib         4583 Aug 25 16:13 s.file

bin:
total 52
   1 drwxrwxr-x  2 projlib          512 Aug 23 11:31 .
   1 drwxr-xr-x  4 projlib          512 Aug 25 16:13 ..
  50 -r-sr-xr-x  1 projlib        51200 Aug 23 11:24 sccs.proj
# rm ,file
```

The previous example shows how to set up a file named `file` in the SCCS directory `/project`. The `create` option to the `sccs` command (`sccs.proj` in the example) does the following:

1. Creates an s. file in the SCCS directory
2. Prepends a comma (,) to the file being placed under source control in the current directory (you should remove this once you are satisfied that the file has been correctly placed under SCCS)
3. Performs a `get` of the file being placed under source control

The project librarian owns all the s. files in the SCCS directory regardless of who created them. Therefore, the project librarian can manipulate the s. files at any time without needing to be the superuser.

Be sure that you use the modified `sccs` command from now on. Otherwise, problems can occur. In this example, you should always use the command `/projects/bin/sccs.proj`.

Note

If users use the `/usr/ucb/sccs` command instead of the modified version-- `/project/bin/sccs.proj` in the example-- to place files in SCCS, the command will fail. To eliminate the possibility of forgetting to use the new `sccs` command over the one distributed with the system, rename the new version `sccs` in the project directory and then be sure the project directory is listed first in your PATH. In the preceding example, you can rename `/project/bin/sccs.proj` to `/project/bin/sccs` and then be sure to list the `/project/bin` directory first in the `PATH` variable of your environment file (usually `.login` or `.profile`).

3.2 Deciding when to Edit a File

SCCS prevents people from modifying a file at the same time by locking an s-file while it is out of SCCS for edit. As a result, you should not reserve files for editing unless they are actually being edited at the time, because this prevents other people on the project from making necessary changes. For example, the following might be an appropriate way to proceed:

```
# sccs edit a.c g.c t.c
# vi a.c g.c t.c (test the experimental version)
# sccs delget a.c g.c t.c
# sccs info
Nothing being edited
# make install
```

When you use the `sccs info` command as shown in the example, SCCS prints the message `Nothing being edited` to confirm that the latest changes have been merged into the SCCS s-file.

As a general rule, you should use the `delta` command to merge all changes to source files before you install a program for general use. Doing this ensures that it is possible to restore any version in use at any time.

3.3 Recovering a Corrupted Edit File

Sometimes you may find that you have corrupted or destroyed a file that you were trying to edit. Unfortunately, you cannot just remove the file and use the `edit` command again because, SCCS keeps track of the fact that someone is editing that file and will not let you take it out for editing again. Neither can you get a copy of the file using the `get` command, because that would expand the ID keywords. Instead, type the following command:

```
# sccs get -k prog.c
```

This command gets a copy of the file `prog.c` without expanding the ID keywords. Therefore you can use the `delta` command with the resulting file.

Alternately, you can use the `unedit` command followed by the `edit` command.

In particularly bad circumstances, the SCCS file itself may become corrupted. The most common way this happens is that someone edits the file incorrectly. Because SCCS keeps a checksum, you get errors every time you read the file. To fix this checksum, type the following command:

```
# sccs admin -z prog.c
```

This command fixs the clears the checksum for the file `prog.c`.

3.4 Using SCCS with the admin Command

There are a number of parameters that you can set using the `admin` command. The most useful of these are flags, which you can by using the `-f` flag. For example, the following command sets the `-d` flag to the value 1 for the file `prog.c`:

```
# sccs admin -fd1 prog.c
```

You can delete the `-d` flag for the file `prog.c` by typing the following command:

```
# sccs admin -dd prog.c
```

The following are the most often used flags:

b	Allow branches to be made using the -b flag with the edit command.
d*SID*	Default SID to be used on a get or edit command. If this is just a release number, this flag 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 an s-file that has the ID keywords inserted as constants instead of internal forms.
y	The type of the module. Actually, SCCS does not use the value of this flag except that it replaces the %Y% keyword.

You can use the -t*file* flag to store descriptive text from file. This descriptive text might be the documentation or a design and implementation document. Using the -t flag ensures that if you send the SCCS file, you will also send the documentation. If you omit file, the descriptive text is deleted. To see the descriptive text, use the prs -t. command.

You can safely use the admin command on files any number of times. You do not need to first reserve the file the using the get or edit commands before you can use the admin command.

3.5 Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain a version of a program for an extended period while normal maintenance continues on the version in production. You can do this by creating 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.

You can create branches using the admin command. For example:

```
# sccs admin -fb prog.c
```

You can specify the -fb flags when you first create the SCCS file.

3.5.1 Creating a Branch

To create a branch for the file prog.c, type the following command:

```
# sccs edit -b prog.c
```

This creates a branch with (for example) SID 1.5.1.1. The deltas for this version are numbered 1.5.1.*n*

3.5.2 Getting Files from a Branch

Deltas in a branch are usually not included when you use the get command. To get these versions for the file prog.c, type the following command:

```
# sccs get -r1.5.1 prog.c
```

3.5.3 Merging a Branch Back into the Main Trunk

If you are working on multiple projects, it is likely that you will have to merge branches at some point. For example, if you are currently working on the prog.c file in a 1.5 branch and someone has created a 1.6 delta that must also contain your work, you can merge the two deltas with the following commands:

```
# sccs edit -i1.5.1.1-1.5.1 prog.c
# sccs delta prog.c
```

If some of the changes conflict, SCCS prints the following error message:

```
inex conflicts at line #
```

You should examine the generated result carefully, since SCCS includes all of the conflicting information in the file. You should then edit the file and remove the unwanted information from the lines identified in the error message.

The following example shows how to merge the file `prog.c` into a branch. You might use this technique to maintain a different version of a program:

1. Create a directory to contain the new version. In this example, the original directory is `xyz` and the new directory is `newxyz`:

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

2. Edit a copy of the program on a branch:

   ```
   # sccs -d../xyz edit prog.c
   ```

3. When using the old version, be sure to use the `-b` flag to the `info`, `check`, `tell`, and `clean` commands to avoid confusion. For example, if your working directory is `xyz`, type the following command:

   ```
   # sccs info -b
   ```

4. To save a copy of the program (still on the branch) back in the s-file, type the following command, which performs a delta on the branch and gets the file out for edit for you:

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

5. When the experiment is complete, merge the file back into the s-file using the `delta` command:

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

At this point you must decide whether this version should be merged back into the trunk (the default version), which may have undergone changes. If so, you can merge it in using the `-i` flag with the `edit` command, as previously described.

3.5.4 Use Branches Sparingly

It is not unusual to have several concurrent development streams in progress at a given time. However, you should use discretion when creating multiple branches. Keep in mind that the more branches there are, the more complex the SCCS structure becomes. With too many branches, tracking the files becomes very complicated.

3.6 Using SCCS with the make Command

SCCS and the `make` command can work together. This section contains sample makefiles for some common applications.

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

`a.out` (or some output file that 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 directory, changing file modes and converting archive files (using the `ranlib` command) as necessary.

`sources` Creates all the source files from SCCS files.

`clean` Removes all unnecessary files from the directory.

`print` Prints the contents of the directory.

The `clean` entry removes any files that you can regenerate from the SCCS files. It is wise to to keep the source files available at all times and the only time that you should remove them is when the directory is being archived in SCCS. To remove the files, type the following command:

```
# sccs clean
```

This command line removes all the files for which an s-file exists, but which is not being edited.

The following sections provide examples to help you understand how to use SCCS. They are only partial and may omit some of the basic entries previously listed when they seem obvious.

3.6.1 Maintaining Single Programs

Frequently there are directories with several largely unrelated programs (such as simple commands). You can put these programs into a single makefile, for example:

```
LDFLAGS= -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 $<
```

As shown in the previous example, 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. The following makefile shows another way of accomplishing the same task:

```
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 number of advantages to this approach:

- The explicit dependencies of the `.o` on the `.c` files are not needed.
- There is an entry called `sources`, so if you want to get all the sources you can do so by typing:

  ```
  # make sources
  ```

- The makefile is more predictable, because it will not try to get files that do not exist.

3.6.2 Maintaining a Library

It is best to update libraries that are largely static using explicit commands, because the `make` command does not know about updating them properly. However, you can handle libraries that are in the process of being developed quite adequately. The problem is that the object files (`.o`) have to be kept both inside and outside the library. For example, here is a makefile:

```
# 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)
```

With this example makefile, you can use the `$(REL)` argument in the line containing the `get` command 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 (none of them is being edited) and will abort the `make` command if this condition is not met.

3.6.3 Maintaining a Large Program

The following example is a simple makefile that uses the `sccsget` command to obtain the source files for the recompilation process.

```
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 those in the previous example.) This makefile requires that copies of the source and object files be kept during development. It is also wise to include lines of the following form so that the modules will be recompiled if any header files change:

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

Because the `make` command does not do transitive closure on dependencies, you may find lines like the following in some makefiles:

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

This would be used in cases where the file `z.h` contains a line like the following in order to bring the modification date of `z.h` in line with the modification date of `x.h`:

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

If you have a makefile such as this, you can remove the `touch` command. You can achieve the equivalent effect by doing an automatic `get` on `z.h`.

See `make`(1) in the *ULTRIX Reference Pages* for further information about the `make` command and makefiles.

Index

L

M

P

R

S

T

U

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to the Source Code Control System
AA-ME84B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line