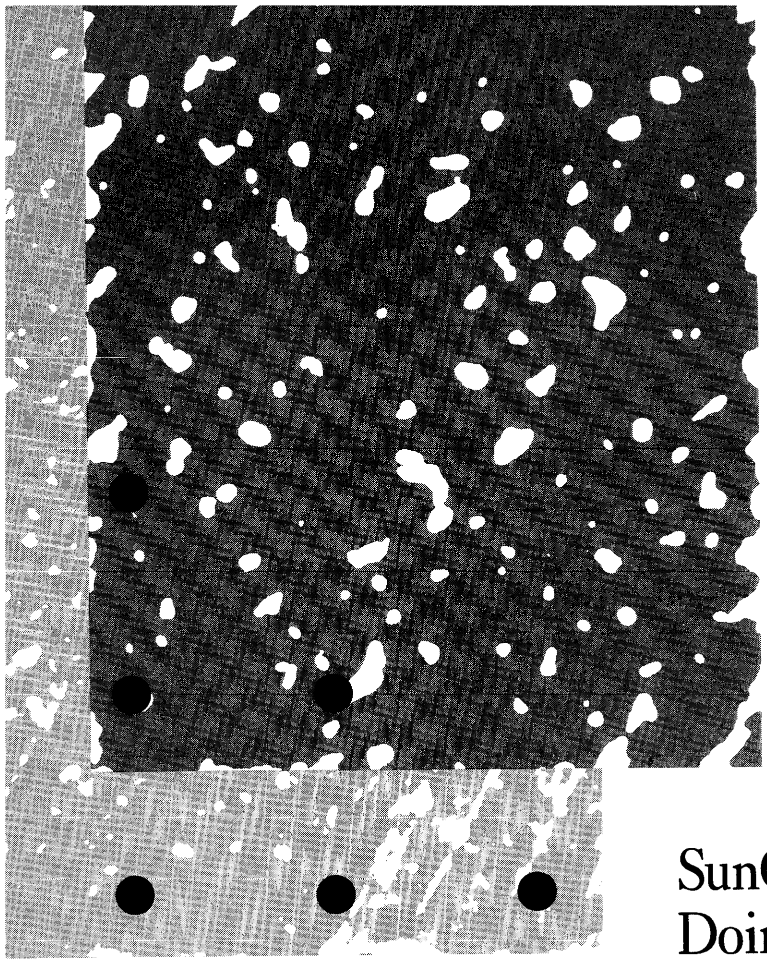


SunOS User's Guide: Doing More

Contents

Tables

Figures

Preface

This manual describes some of the more sophisticated features the SunOS™ operating system provides, and how to use them to simplify complicated tasks. If you have not already done so, you should read *SunOS User's Guide: Getting Started* before using this manual.

Chapter 1 provides details about maintaining file security.

Chapter 2 describes commands relating to other users on the system, including `root`, the "superuser."

Chapter 3 introduces tools for sophisticated file management.

Chapter 4 continues the discussion of the C shell and its timesaving features begun in *SunOS User's Guide: Getting Started*.

Appendices A, B, and C introduce you to writing shell scripts, in both the C shell and the Bourne shell.

1 Securing Your Files

1.1. The Importance of Security

If you work on a computer linked to a network, many other users could have access to your files. This is a fine thing when you want your data to be shared or modifiable by others; but it's problematic when you want to protect the integrity or confidentiality of your files.

Your system adminstrator has primary responsibility for maintaining the security of the system. But as a user, you can protect your own machine and files from unauthorized use, and at the same time allow access to the good guys.

1.2. Maintaining Password Security

In *SunOS User's Guide: Getting Started* you learned how to enter and to change your password. Your password is the first line of defense against unauthorized use of your workstation. Here are some things to consider when choosing and using your password:

- Do not select a password that is easily guessable. The name of your spouse, your hobby, your phone number, your first name, your birthday, the brand of your automobile, your favorite sport are *not* good choices for passwords. Do not use a word in the on-line dictionary, since such words can be tried automatically.
- Good passwords are at least six characters long, aren't based on personal information, and have non-alphabetic characters in them.
- Never write your password down on paper. A password that is impossible to remember is worse than one too easily guessable, because you'll end up writing it down.
- Don't use the same password for every account you have.
- Do not let anyone watch your fingers as you type your password. (Remember that passwords do not appear on the terminal screen.)
- Change your password whenever you think it may have been compromised. Changing it from time to time anyway is not a bad idea.

Password Aging

If your system is using *password aging* (implemented with options to the `passwd` command), your password may have either a maximum or a maximum *and* minimum lifespan. The lifespan of your password is set by your system adminstrator.

When the maturity date (or maximum age) of the password has elapsed, the `login` program will require you to change your password. You will see the message

```
Your password has expired. Choose a new one.
```

The system then automatically runs the `passwd` program and prompts you for a new password.

If the minimum age of your password has been set for, say, two weeks, and you try to change your password before that time has elapsed, the system will respond with the message

```
Sorry, less than 2 weeks since the last change.
```

To display aging information on your password, use the `-d` option to the `passwd` command:

```
venus% passwd -d

username    4-20-90     14  60
```

The display shows, in order, the date of creation of the current password, the minimum age, and the maximum age. (This information will be displayed only if password aging as been implemented.)

For more information on password aging, see *System and Network Administration*. Type **man passwd** to see the on-line man page.

1.3. Locking Your Terminal Screen

If you're running the SunView window system, you can lock your terminal screen against unauthorized access while preserving the state of the SunView display.

The `lockscreen` program clears the workstation screen and then, typically, provides a moving graphics display to reduce phosphor burn. `lockscreen` will require your password before restoring your window display. See the *SunView User's Guide* for more information.

1.4. Controlling Access Permissions

As explained in some detail in *SunOS User's Guide: Getting Started*, every file and directory is controlled by a series of access permissions. These permissions determine who can read, write, and execute your files. Users are broken down into three categories: the file owner, the owner's group, and everyone else.

Permissions can be changed to suit by using the `chmod` and `umask` commands. File ownership can be changed with the `chown` command. (Note that `chown` requires that you become superuser; see Chapter 2.) Be sure that you've set the appropriate permissions on your files: do you want your mail files accessible by anyone? should your group be able to read but not write to some of your work files? etc.

In particular, make sure your initialization files — .login, .cshrc, .profile, .mailrc, .sunview, .exrc, among others — are owned and writable only by you. And make sure your home directory is writable only by you.

Finally, note that programs often use the directories /tmp and /var/tmp to stash temporary files. These directories are readable by everyone — so unless the files you put there are unreadable, they will be open to everyone.

1.5. Encrypting Files

Encrypting With crypt

You can use crypt[1] to encode the contents of confidential files. This command rolls a key through the text, disguising it. You supply the key, which is used both to encrypt and decrypt the file. Unfortunately, this method of encryption is vulnerable to attack: encrypted text can be examined for patterns that will reveal the key. But the longer the key, the greater the security against such attacks. However, in practice keys are limited to a maximum of eight characters.

Here's how you would use crypt to make a sensitive file more secure:

```
venus% crypt < filename > .filename
Enter key: (typed key invisible)
venus% rm filename
```

You don't need to put a dot before the output file name, but it helps to keep the file out of sight if an intruder runs ls on the directory. Don't call the file something .crypt, because that makes it obvious how the file was created. It's a good idea to use chmod to change permissions to make the encrypted file readable and writable only by its owner. Note that in this example the original, unencrypted file was removed.

It's important to remember the key. Without it, you can't decrypt the file.

crypt does not prompt you twice for the key; you should type it in carefully— avoiding typographical errors that would make the text unrecoverable.

You can also use crypt to decode a file:

```
venus% crypt < filename > newfile
Key: (type key)
venus%
```

This will create a new file, called newfile, containing the unencrypted text you started out with.

If you want to look at the decoded contents without creating a new file, a command of the form:

[1] SunOS encryption facilities are only available to customers within the United States of America.

```
venus% crypt < filename | more
```

will, after asking for the key, display them on the screen.

You can also use vi to edit the data directly, without creating an intermediary file. Start vi with the -x option:

```
venus% vi -x filename
Enter key: (typed key invisible)
(editor continues as usual)
```

vi automatically re-encrypts the file (using the same key) before writing to disk.

Encrypting With des

You can use the des command for greatly improved security for sensitive files. The disadvantage of des encryption is that you can't use vi or another editor to edit the file. On the other hand, since des uses different algorithms for encrypting and decrypting, a file is very secure — unless someone can guess the key. (It's up to you to ensure that the key is unguessable.)

Here's how to encrypt a file with des:

```
venus% des -e filename > filename.des
Enter key: (typed key invisible)
des: WARNING: using software DES algorithm
venus% rm filename
```

If your machine has DES hardware assist, you won't see the warning message.

Don't forget the key! You won't be able to decrypt the data without it.

Here's how to decrypt a file with des:

```
venus% des -d filename.des > newfile
Enter key: (typed key invisible)
des: WARNING: using software DES algorithm
```

If you forget or mistype the key, des will warn you that decryption failed, and the resultant output file will be illegible.

2

Other Users

2.1. Other Users

From the system's standpoint, every user has a login name, a password, an identification number, or *userid*, a group membership, a user's name or other pertinent data, a home directory, and a default shell. This information is kept in the file `/etc/passwd`. To find out who can log in to your system, look in this file.

Figure 2-1 *The* `/etc/passwd` *File*

```
root:0XtYHFnkYou3Y:0:10:Operator:/:/bin/csh
daemon:*:1:1::/:
uucp:eXs0qzRjUOS8Y:4:8::/var/spool/uucppublic:
cindy:Lu8UBYYbPNEpw:26:20:Cindy Smith:/home/brno/cyndi:/bin/csh
carter:SQxRMoQbqQ0Hk:612:20:Jamie Carter:/home/brno/carter:/bin/
jimg:lUvG9UKY0uE/A:1131:60:Julie Gomez:/home/brno/jimg:/bin/csh
ben:bAwVM.A6LiXFo:1132:30:Ben Benson:/home/brno/ben:/bin/csh
karla:mceur1TqKdcDQ:1172:30:Karla Caracas:/home/brno/karla:/bin/
```

Fields corresponding to the above categories are separated by colons, and described in the following table (using the last line above as a sample entry).

Table 2-1 *Information Contained in* `/etc/passwd`

Field	*Sample*
login name	karla
encrypted password	mceur1TqKdcDQ
user ID number	1172
group ID number	30
commentary	Karla Caracas
home directory	/home/brno/karla
login shell	/bin/csh

The first line of this file contains an entry for `root`, the operator of the system. When logged in as `root`, the operator can access any file or device on the system, perform system maintenance, and edit system files such as this. (For more on `root`, see Section 2.2.) The next two entries allow for certain networking functions to be performed, and the subsequent lines correspond to individual users.

If you are using the Network Information Service (NIS), then a line reading `+::0:0:::` in `/etc/passwd` gives login privileges on your machine to anyone in the NIS domain. To find out more about the NIS , and users with access over the network, refer to *SunOS User's Guide: Getting Started* and *System and Network Administration.*

For a more complete treatment of `/etc/passwd`, see the *SunOS Reference Manual* or type **man 5 passwd**.

Users Currently Logged In

The system tries to provide equivalent performance to everyone using it. To find out who is logged in, type `who`. `who` shows you the login-name of each user on the system, the terminal that person is using, when they logged in, and, if logged in from a remote machine, the name of that machine. See *SunOS User's Guide: Getting Started* for more detailed information about using remote machines.

From time to time, you may want to see what others are doing. The `w` command tells you what command is running on each user's terminal. In addition, it shows you the amount of time since the user last typed something in (idle), the total CPU time spent by each user so far (JCPU), the CPU time spent by the command now running (PCPU). To get a detailed list of everyone's processes, use the command

```
ps -au
```

```
venus% ps -au
USER       PID %CPU %MEM   SZ  RSS TT STAT  TIME COMMAND
landon   19755 49.8 10.0  212  140 0c R     0:03 ps -au
wilkie   19751 42.4 15.8  366  226 17 S     0:12 vi mail.record
dewey    19754  4.8  8.3  232  114 09 S     0:02 /usr/lib/sendmail -bm c2
goldwatr 18732  0.0  0.0  186    0 19 IW    0:44 mail
ford     19752  0.0  2.2   70   24 16 S     0:00 pmsg
bush     18085  0.0  0.0  300   86 p0 IW    0:10 vi eco
venus%
```

The `-a` option tells `ps` to show you information about all processes, not just your own. The `-u` option gives a more detailed display that includes the name of the user who owns the process. The `-au` option is simply the combination of these two.[2] For information about the remaining columns, refer to `ps` in the *SunOS Reference Manual.*

Changing Identity With `su`

If you know someone else's password, you can temporarily assume that person's system identity by using the `su` (*superuser*) command. A common reason for doing so is to get access to files that you don't own. Suppose that a colleague has moved a file into one of your directories that you want to edit:

[2] Single-letter options that can be combined like this are sometimes referred to as *flags*.

```
venus% ls -l
total 34
-r--r--r--  1 sam      1697 Aug  2 13:35 env.b
-r--r--r--  1 sam      1244 Aug  2 13:50 chapter.1
-r--r--r--  1 jd       3623 Aug  2 13:50 program.source
```

First, use cp to make a copy of the file. You will own the copy, and can edit it. To get rid of the version you don't own, switch your userid and delete it:

```
venus% cp program.source my.source
venus% su jd
Password: ...
venus% rm program.source
venus%
```

To revert to your previous ID, type Ctrl-D (or the command exit).

If, after switching userids, you want to find out what your effective login identity is, type whoami:

```
venus% whoami
jd
venus% exit
venus% whoami
sam
```

The command who am i reveals your original login identity when you use su to temporarily become someone else.

Note that it is generally considered very bad manners to su into someone else's identity without explicit permission and notification. Consult your system administrator for usage at your site.

2.2. Becoming root, the Superuser

Each machine has a *superuser*, a user who has powers and permissions quite above and beyond those of ordinary users. This superuser is often known as root. A person with superuser status can edit files which are off-limits to other users, such as /etc/passwd (the password file) or /etc/hosts.equiv (the list of other machines on a network that your machine trusts). root can also use some restricted commands, such as mount or reboot.

Originally, the UNIX operating system, on which SunOS is based, was designed for many users to be working on a single, more-or-less centralized machine. One person, the system administrator, was in charge of maintaining, configuring, and upgrading the system — hence the name *superuser*.[3]

[3] This is still the setup for people using "time-sharing" machines.

With a network of independent workstations like Suns, however, each person may have the ability to become `root` on his or her own machine. They can take care of many of the tasks that were formerly the province of the superuser, such as making connections to printers or mounting remote filesystems. In a workstation environment, then, a superuser and a system administrator are not necessarily the same thing: a system administrator is now someone who maintains *shared* machines and networks.

For example, suppose you are a diskless client of the server `chiqui`. That means that you have your own workstation — call it `venus` — and you keep your files on the machine `chiqui`. You can become superuser on your own machine. But maintenance and configuration of `chiqui` is left to your system administrator. On the other hand, if you are running a stand-alone system (one with a disk), then you are the system administrator and you become `root` to carry out all system administrator tasks.

If you type `su` with no name, it attempts to switch you to `root`, also referred to as the *superuser*. When you become the superuser, the last character of the prompt changes from a percent sign (`%`) to a pound sign (`#`):

```
venus% su
Password: ...
venus# (root prompt)
...
venus# exit (exits root)
venus%
```

As `root`, you can kill any process running on your machine. You have read and write privileges on every file on your machine's disk (or disk partition) and you can change the ownership of these files.[4]

To quit being `root` and return to your own identity, type **`exit`**.

You must become `root` to perform system maintenance tasks such as adding new users, adding new terminals or printers, etc. Refer to *System and Network Administration* for more information on performing these tasks.

[4] Files mounted from a remote host belong to that machine. You must be logged in as `root` on the remote host to get superuser privileges for files that reside on it. Refer to *SunOS User's Guide: Customizing Your Environment* to find out more about remote hosts and mounted file systems.

3

Managing Your Files

The SunOS operating system has good facilities to help you locate files, monitor changes to important files, and manage your space on the disk.

3.1. Locating Files

To locate a file in the file system hierarchy, you may need to know its absolute pathname. When trying to locate a file, chances are that you are either looking for the pathname of a particular command, or you are looking for a certain text file. The operating system provides several ways to locate commands. These are presented first, followed by methods for locating text files.

Looking Up a Command With `whereis` and `which`

To find the pathname of a standard SunOS command, type in `whereis` followed by the command name (`whereis` also displays the pathname of the `man` entry):

```
venus% whereis csh
csh: /bin/csh /usr/man/man1/csh.1
```

You can also use `which` to look up a command. This is useful when you have commands that are aliased or when your system contains commands in addition to the standard set. If the command is an alias, `which` shows you its definition. If the command is in a directory listed in your `path` variable, `which` displays its pathname. If there is more than one version of a command in those directories, `which` displays the version that the system finds first. This is the same version that the system performs when you type in the command.

```
venus% which ls
ls:      aliased to ls -F

venus% which chesstool
/usr/games/chesstool
```

Looking Up a Command's Description With `whatis`

Typing `whatis`, followed by the name of a command, will give you a brief description of what that command does:

```
venus% whatis whatis
whatis (1)  - display a one-line summary about a keyword
```

`whatis` will not work if your system administrator has not run the `catman` command after installing the system.

Looking Up Files With `find`

Starting with a named directory, `find` searches for files that meet conditions you specify. A condition could be that the filename match a certain pattern, that the file is owned by a certain user (or belong to a certain group), or that the file has been modified within a certain time frame.

Unlike most SunOS commands, `find` options are several characters long and the name of the starting directory must precede them on the command line.

`find` *`directory options`*

Each option describes a criterion for selecting a file. A file must meet all criteria to be selected. So the more options you apply, the narrower the field becomes. The `-print` option indicates that you want the results to be displayed. (As described later on, you can use `find` to run commands. You may want `find` to omit the display of selected files in that case.)

The `-name` *filename* option tells find to select files that match *filename*. Here *filename* is taken to be the rightmost component of a file's full pathname. For example, the rightmost component of the file `/usr/lib/calendar` is `calendar`. This portion of a file's name is often called the *basename*.

To see which files within the current directory and its subdirectories end in *s*, type:

```
venus% find . -name '*s' -print
./programs
./programs/graphics
./programs/graphics/gks
./src/gks
...
venus%
```

Other options include:

`-name` *`filename`*
: select files whose rightmost component matches *filename*. Surround *filename* with single quotes if it includes filename substitution patterns.

`-user` *`userid`*
: select files owned by *userid*. *userid* can be either a login name or user ID number.

`-group` *group*	select files belonging to *group*.
`-mtime` *n*	select files that have been modified within *n* days.
`-newer` *checkfile*	select files modified more recently than *checkfile*.

You can combine options within (escaped) parentheses (`\(...\)`) to specify an order of precedence for criteria. Within escaped parentheses, you can use the `-o` flag between options to indicate that `find` should select files that qualify under either category, rather than just those files that qualify under both:

```
venus% find . \( -name AAA -o -name BBB \) -print
./AAA
./BBB
```

You can invert the sense of an option by prepending an escaped exclamation point. `find` then selects files for which the option does *not* apply:

```
venus% find . \!-name BBB -print
./AAA
```

Running Commands With `find`

You can also use `find` to apply commands to the files it selects with the

```
-exec command '{}' \;
```

option. This option is terminated with an escaped semicolon (`\;`). The quoted braces are replaced with the filenames that `find` selects.

You can use `find` to remove automatically temporary work files. If you name your temporary files consistently, you can use `find` to seek them out and destroy them wherever they lurk.[5] For example, if you name your temporary files `junk` or `dummy`, this command will find them and remove them:

```
find . \(-name junk -o -name dummy \) -exec rm '{}' \;
```

Looking at File Types With `file`

Sometimes you want to see what sort of data a file contains without having to look at its contents. In particular, if the file is a compiled program (*object-file*), trying to display its contents can produce spectacular and disconcerting results on your screen. `file` quickly tells you whether a file contains, for example, plain text, `troff` sources, C program sources, executable files, or tape-format archives. (There are a number of kinds of files; see under *file* in the *SunOS Reference Manual.*)

[5] For good housekeeping, you may want to get rid of such files on a regular basis without having to think about it. If you put a command like this in your `.logout` file, then the system will clean up unwanted files for you whenever you log out. You can also use the `crontab` facility; see *SunOS Reference Manual.*

```
venus% file *
AAA:  empty
document:  nroff, troff, or eqn input test
troff.output:  troff (CAT) output
program:   demand paged pure executable
scratch:  ascii text
```

3.2. Looking at Differences Between Files With `diff`

It often happens that different people with access to a file make copies of it and then edit their copies. `diff` will show you the specific differences between versions of a file and provide you with an indication of how the contents of one can be edited to produce the other. The command

```
diff leftfile rightfile
```

scans each line in *leftfile* and *rightfile* looking for differences. When it finds a line (or lines) that differ, it determines whether the difference is the result of an addition, a deletion, or a change to the line, and how many lines are affected. It tells you the respective line number(s) in each file, followed by the relevant text from each.

If the difference is the result of an addition `diff` displays a line of the form

```
l[,l]  r[,r]
```

where *l* is a line number in *leftfile* and *r* is a line number in *rightfile*.

If the difference is the result of a deletion, `diff` uses a `d` in place of `a`; if it is the result of a change on the line, `diff` uses a `c`.

The relevant lines from both files immediately follow. Text from *leftfile* is preceded by a *left* angle-bracket (<). Text from *rightfile* is preceded by a *right* angle-bracket (>).

This example shows two sample files, followed by their `diff` output:

```
venus% cat sched.7.15
Week of 7/15

Day:  Time:        Action Item:          Details:

T     10:00        Hardware mtg.         every other week
W     1:30         Software mtg.
T     3:00         Docs. mtg.
F     1:00         Interview
venus% cat sched.7.22
Week of 7/22

Day:  Time:        Action Item:         Details:

M     8:30         Staff mtg.           all day
T     10:00        Hardware mtg.        every other week
W     1:30         Software mtg.
T     3:00         Docs. mtg.
venus% diff sched.7.15 sched.7.22
1c1
< Week of 7/15
---
> Week of 7/22
4a5
> M     8:30         Staff mtg.         all day
8d8
< F     1:00         Interview
```

3.3. Monitor Changes With sccs

When you want to protect an ASCII file from accidental deletion, keep track of changes to it, or allow more than one person to modify it, you can monitor the file using sccs. sccs (or "source code control system") is a utility program that protects important files by allowing only one person at a time to make changes, by maintaining a record of those changes, and by rebuilding the current (or any previous) version upon request.

Putting a File Under sccs Control (sccs create)

To put a file under sccs control, perform the following steps:

(1) cd to the directory containing the file(s) to be protected. If a subdirectory named SCCS is not already present, create it. If you want to allow other users access to the files, change the permissions of the current directory and those of the SCCS subdirectory to 775.[6]

For information on changing permissions with chmod and umask, see *SunOS User's Guide: Getting Started*.

```
venus% cd project
venus% mkdir SCCS
venus% chmod 775 . SCCS
```

[6] Unless you are sure that you do *not* want them to have access, it is normally a good idea to change permissions of both directories to allow it, at least for other members of your user group.

(2) Type in a command of the form:

```
sccs create filename
```

filename is the name of a file or files to monitor. This is how you would put all you files under SCCS:

```
venus% sccs create *
```

For each file that you indicate on the command line, sccs produces a special file called a *history* file, and puts it in the SCCS subdirectory. The history file has a name of the form:

```
s.filename
```

History files are also referred to as "s.files."

and contains a complete record of all lines changed throughout the life of the file. sccs maintains a checksum on all history files, so *do not* edit them!

sccs may respond with the warning:

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

This message can safely be ignored when you are auditing your own files.

When working with files that are part of a large project, sccs ID keywords can be important. For more information about sccs as a tool for managing large programming projects, refer to *Programming Utilities and Libraries.*

(3) Remove the backup file(s) that sccs leaves behind. These files are created by sccs as a safety precaution, and are no longer necessary once the create operation is complete. Names of these backup files begin with a comma (,).

```
venus% rm ,*
```

Once under sccs control, you have to check a file out before you can make changes to it. Files that aren't checked out through sccs have permissions set to read-only for everyone (444).

Which Files Are Checked Out? (sccs info)

To see which files in the working directory are checked out, use the sccs info command. If no files are checked out, sccs responds with the message:

```
Nothing being edited
```

If there are files checked out, it lists those that are, the current version number of each, the version number each will have when checked in again, the name of the user who checked out each, and the date and time of check-out:

```
csh.1: being edited: 1.4 1.5 sam 85/09/04 16:32:15
```

Recovering the Current Version (`sccs get`)

Because several people may have write access to the directory, it is possible that a file in the working directory may be deleted accidentally. Files that ***aren't*** under `sccs` control are gone for good once they are removed, but you can easily restore files under `sccs` from their history files using the `sccs get` command:

```
sccs get filename
```

If you want to recover the current version of all files in the directory, use the command:

```
sccs get SCCS
```

Checking a File Out (`sccs edit`)

Only one person at a time can check a file out. This assures you that changes won't be lost, garbled, or intermixed between the edits of different users.

To check out a file, type in `sccs edit` followed by the file or files you wish to check out. `sccs` will respond with the current version number, the new version (delta) number, and the number of lines in the file.

```
venus% sccs edit program
1.1
new delta 1.2
220 lines
venus%
```

Once checked out, you can edit the file using `vi` or another editor.

When you check out a file, `sccs` changes the ownership of the file to you, gives you write permission (owner only), and places a *lock* file containing your userid, the version number, and other information in the `SCCS` directory.[7] When you check the file back in, the lock file is removed and the permissions are set to read only, but you retain ownership of the file.

Looking at Current Changes (`sccs diffs`)

While still checked out, you may want to review the changes you have made so far. To do so, type:

```
sccs diffs filename
```

`sccs` responds with standard `diff` output, using `sccs`'s current version as the "leftfile" and the *filename* as the "rightfile." (See Section 3.2.)

Checking a File In (`sccs delget`)

When you are done making changes, you can check in the new version of the file by typing the command:

```
sccs delget filename
```

`delget` is a contraction for `delta`, the command to incorporate a new version into the history file, and `get`, the command to recover the newest version (that you are just now checking in).[8]

[7] The lock file has a name of the form: `p.`*filename*, and referred to as a "p-file."

[8] If `sccs` responds with an error message, it does not perform the `get` action, and you may have to recover

When you use `delget` (or `delta`) to check in the file, `sccs` asks you for a line of comments. These comments are included in the history file, and should briefly summarize the changes you have made. After adding your comments and pressing [Return], `sccs` responds with the new version number, the number of lines inserted, deleted, and unchanged, and the total number of lines.

```
venus% sccs delget program
comments? added remarks for more readable code
1.2
43 inserted
18 deleted
287 unchanged
1.2
348 lines
```

A replaced line shows up as an insertion and deletion.

Backing Out With No Changes (`sccs unedit`)

To check a file back in without any changes, type in:

```
sccs unedit filename
```

Looking at the File's History (`sccs prt`)

To review a file's history, use the command:

```
sccs prt filename
```

This command shows you the version number, comment lines, date checked in, and user responsible for each version of the file:

```
venus% sccs prt program
SCCS/s.program:

D 1.2 85/09/04 12:51:07 sam 2 1 00042/00008/00357
MRs:
COMMENTS:
added remarks for more readable code

D 1.1 85/08/30 16:54:57 sam 1 0 00365/00000/00000
MRs:
COMMENTS:
date and time created 85/08/30 16:54:57 by sam
```

Comparing Versions (`sccs sccsdiff`)

To compare previous versions of a file, use the command

```
sccs sccsdiff -rx.y -rm.n filename
```

Where *x.y* and *m.n* are version numbers to be compared. This command produces standard `diff` output.

files using `sccs get SCCS`.

Restoring a Previous Version (`sccs get -r`)

If you want to back out a version of the file that is already checked in, you must perform the following steps:

(1) Recover the previous version. You can look up its number using `sccs prt` *filename*. To rebuild the previous version, type in a command of the form:

```
sccs get -rx.y filename
```

where *x.y* is the desired version number.

(2) Rename the recovered version of the file

```
mv filename temp
```

(3) Check the file out with `sccs edit`.

(4) Replace the checked-out version with the old version:

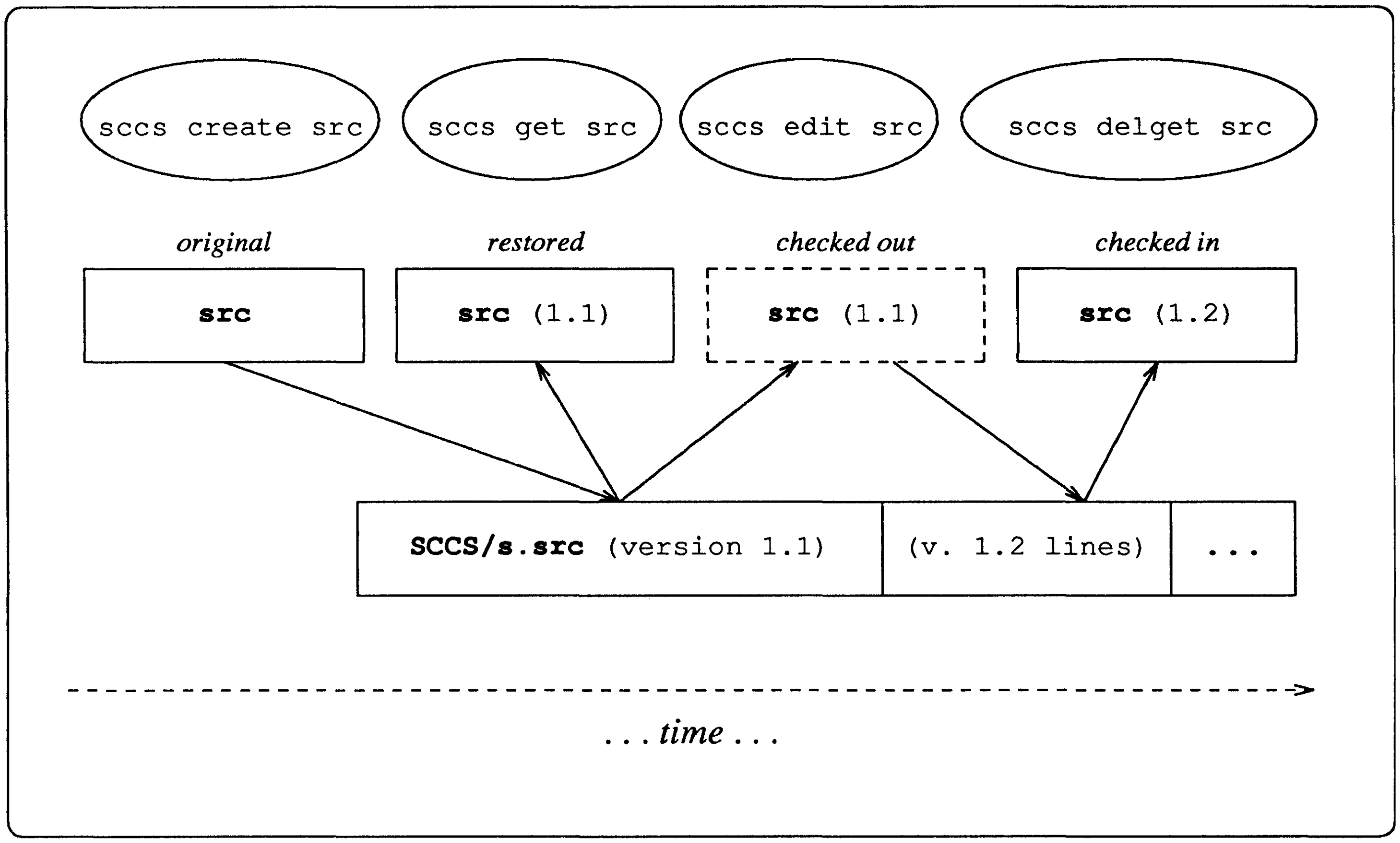
```
mv temp filename
```

(5) Check the file back in with `sccs delget`.

To assure that it all worked properly, compare the latest version with the desired previous version using `sccs sccsdiff`.

The typical flow of events when making changes to a file under `sccs` control is:

Figure 3-1 *Flow of Events With* `sccs`*-Controlled Files*



Solving Problems With `sccs`

`sccs` is a complicated and verbose utility. There may be times when it responds with an error message even though things worked properly. Its error messages are sometimes difficult to interpret. If you are not sure that `sccs` succeeded in doing what you asked, you can take certain steps to verify whether it has:

`ls -l SCCS`
: will show an s.file for each file under `sccs` control.

`sccs info`
: will show which files are checked out and to whom.

`sccs prt` *`filename`*
: will show your comments in the first three lines when you have checked in a file successfully.

If you attempt to check a file out and you get the message:

```
ERROR [SCCS/s.filename]: writable 'filename' exists (ge4)
```

this usually means that someone has the file checked out already. You can verify this using `sccs info`. If `sccs info` does not list the file as being edited, then the lock file in the `SCCS` directory has been deleted. When this happens `sccs` will not allow anyone to check the file either in or out.

To correct this problem, first run `sccs diffs` on the file to see if it differs from the version last checked in. If so, it is a good idea to contact the file's owner to find out if the changes made should be kept. If so, then copy the file to a new filename, remove the writable original, and check the file out using `sccs edit`. Then move the new filename back to the original name (overwriting the checked-out version), and check the new version back in using `sccs delget`.

If the changes need not be saved, you can correct the problem by simply removing the writable file, restoring the current version using `sccs get` and then checking it out using `sccs edit`.

3.4. Automating Complicated Tasks With `make`

Performing complicated tasks — such as producing object code for programs or formatting large documents — involves processing different files through various programs at the proper times and in the proper order. This can be a lot to remember. `make` simplifies these complications by following a record of the steps involved, called a *makefile*, that you create.

The makefile contains a list of the steps called *targets;* each target contains a list of SunOS commands. A target can be qualified by a list of other targets upon which it depends. One target is said to depend on another if the latter must be completed before the former can be performed successfully. The latter target is called a *dependency.*

For example, an `SCCS` subdirectory must be created before you can put files under `sccs`. And you must put a file under `sccs` with `sccs create` before you can check that file out. So the command `sccs edit` depends in practice on the commands `mkdir SCCS` and `sccs create` for its own success.

`make` uses the list of targets as a recipe to produce a desired program, document, or other object file called a *target file,* or simply *target.*

make performs only those steps that are required to bring the target files up to date. The makefile lists the various steps involved and how they depend on one another, and make examines the list to see which target files are outdated.

A target is considered to be outdated when the source file used to produce it has changed since the target file itself was last produced. make then performs only those steps required to replace any outdated target files.

make has a facility to perform *macro substitution*.[9] This allows you to abbreviate long lists and to predefine parameters that often change, so that with a few simple edits the same procedure can be used to produce other, similar objects.

Makefiles

Like a recipe card, a makefile is composed of two sections. The first section is a list of macro definitions. These are described in detail later on. The second section outlines steps in the procedure and their relationships to one another. In make parlance, each step is called a *target*.

Each target has a name. If that target's function is to produce an object file of some sort, then the name of the target should be the same as the name of the file it produces. If the target performs some sort of housekeeping step, then it can have any name you like.

A target may also have a list of *dependencies* associated with it. make uses this list to determine whether files produced by the target are up to date.

Finally, each target has a list of SunOS commands to perform. When performing a step, make performs each command in turn, starting a Bourne shell[10] for each command line.[11]

The following is an example of a makefile to put the contents of a directory under sccs control. The file consists of just two targets and no macro definitions:

[9] Like an alias, a *macro* is a string of text that is replaced by its definition, or *expansion* when encountered in an input file (or command line).

[10] Because it runs a Bourne shell, certain C shell constructs, such as foreach, don't work. Refer to sh in the *SunOS Reference Manual* for more information about the Bourne shell.

[11] Since each command line is executed in its own shell, you must use the command-separation character ; and the command-line continuation character \ Return to build command *routines*.

Figure 3-2 *Sample Makefile To Put Files Under* `sccs`

```
# makefile: for putting files under sccs

#         no macro definitions

#         target definitions

put.under:  SCCS
    # these lines begin with a required tab character
    -sccs create *
    -rm ,*
    -sccs get SCCS

SCCS:
    -mkdir SCCS
    -chmod 775 SCCS .
```

The targets are `put.under` and `SCCS`. The target `put.under` depends on the target `SCCS`. If the `SCCS` directory is not already present and up to date (directories always are), `make` performs the commands listed under `SCCS` first.

The format of each target is significant. The name of the target must be followed by a colon and the list of dependencies, if any. (If this list is longer than one line, you can split it in two by leaving a backslash (\) at the end of the first line.) The list of commands immediately follows the target name, and each command line begins with a Tab.

Comments begin with a # and can be placed to the right of commands on any line (not ending in a backslash). At least one blank line separates target definitions from one another.

When you prepend a – to a command, `make` ignores a nonzero (error) return code from that command. Normally, `make` halts whenever a command it runs exits with a nonzero status. Adding the dashes in this case tells `make` to continue putting new files under `sccs` control, even though it may encounter older files already there.

Because `make` checks for dependencies, you can write makefiles in a top-down fashion. The step that produces the final output should appear first. Steps that it depends upon can appear next, followed by steps that *they* depend on.

Running `make`

When the makefile is ready, simply type in `make`.

`make` looks for a file in the working directory named `makefile` or `Makefile`,[12] checks for dependencies, beginning with the first target it encounters, and then performs commands in their proper order:

```
venus% make
mkdir SCCS
chmod 775 SCCS .
sccs create *

SCCS:
ERROR: directory 'SCCS' specified as 'i' keyletter value (ad29)

makefile:
No id keywords (cm7)
[messages from sccs]
rm ,*
sccs get SCCS
[messages from sccs]
venus%
```

The error message

```
ERROR: directory 'SCCS' specified as 'i' ...
```

indicates that sccs attempted to create a history file for the directory SCCS. Because we used a dash as the first character of the command line, make continued processing.

Testing Makefiles

Most makefiles take a bit of debugging. To find out what commands make will perform without actually running them, use the –n option:

```
venus% make -n
sccs create *
rm ,*
sccs get *
```

In the above makefile, put.under depends upon SCCS. When you ran make the first time, the SCCS directory was created. When you ran make -n subsequently, make did not indicate that it would perform that step (since it was up to date anyway). If you were to remove the SCCS directory and then run make, it would perform commands in the SCCS target once again.

Defining Macros in the Makefile

The next example is a makefile used to format and print a document made up of several source files. With macro substitution, copies of a makefile such as this can be used for different documents:

[12] You can specify the name of some other makefile, using the -f *filename* option, as in make -f buildit, where buildit is a different Makefile.

Figure 3-3 *Sample Makefile for Printing a Document*

```
# Makefile: for printing a document

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output
```

A change to the list of sources, the printer, or the macro package can be made in one place and take effect throughout the makefile. For large and complex procedures, this is a big advantage.

By placing the `troff` output in an intermediate file,[13] you can avoid having to reformat the document every time you want to print a copy. By making `print` depend upon the file `troff.output`, you can be sure that you always get the latest formatted version.

By making `troff.output` depend on the list of sources (the expansion of the `$(SOURCES)` macro), you can be sure that when you change any one of the sources, `make` will rebuild `troff.output`, and the change will be reflected when you print the document.

Selecting a Target

You can select any target in the makefile by specifying it as an argument to `make` on the command line. If a target does not appear in the list of dependencies for the target you select (or the first target by default), `make` will not perform it. So you can record several independent procedures within the same makefile. For example, this makefile can be used either to put new source files under `sccs` or to print a finished document.

[13] `troff` intermediate output files are *not* text files. They will produce strange results if you try to look at them on the screen, and they should *not* be placed under `sccs`. It would be a good idea to put the source files under `sccs` instead.

Figure 3-4 *A Makefile With Independent Procedures*

```
# Makefile: for printing a document
#           and putting sources under SCCS

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output

# ----------------------------------------------------------

put.under: SCCS
# the next three lines begin with a tab
        -sccs create 'ls | grep -v troff.output'
        -rm ,*
        -sccs get *

SCCS:
        mkdir SCCS
        chmod 775 SCCS .
```

Using this makefile, if you type in `make` (or `make print`), you will get the document (typing `make` does everything in the makefile). If you type in

```
make put.under
```

your sources will be put under `sccs`.

3.5. Managing Disk Storage

Space on the disk is a limited resource. Therefore, it's a good idea to keep track of how much space you use, especially if your system is running with disk quotas.[14]

The SunOS operating system provides facilities to monitor your disk usage and locate big directories that are candidates for housekeeping. Even so, it can be unwise to delete old files willy-nilly. Since you might not know what gems you may have locked away there, the system also provides a facility to make tape archives of important files. Tape archives are especially good for large files that

[14] A disk quota is a limit on the amount of space (information) a user is allowed to use on the disk at any one time.

you need to keep but don't often use. If you make a tape archive before cleaning house, you can be sure that you won't lose anything important. You can use `df`, `du`, `find`, and `ls -l` to locate such files, and then you can use `tar` to move them onto a tape for storage off-line, as described in the following sections.

Looking at Disk Usage With `df`

`df` shows you the amount of space used up on each disk that is mounted (directly accessible) to your system. It is very simple to use, just type

```
df
```

to see the capacity of each disk mounted on your system, the amount available, and the percentage of space already used up:

```
venus% df
Filesystem            kbytes    used   avail capacity  Mounted on
waldo:/export/root/donkey
                       75651   17302   50783    25%    /
waldo:/usr            106303   54120   41552    57%    /usr
krakow:/usr/krakow    547697  374274  118653    76%    /home/krakow
athens:/usr/athens    266107  212150   27346    89%    /home/athens
toupee:/usr/view      326031  213711   79716    73%    /home/view
salsa:/usr/salsa      352022  299302   72344    75%    /home/salsa
waldo:/home/waldo     371967  327280    7490    78%    /home/waldo
waldo:/export/share   106303   54120   41552    57%    /usr/share
popeye:/usr/games      54387   48649     299    89%    /usr/games
krakow:/usr/tools      39095   31396    3789    89%    /usr/tools
croaker:/usr/demo      62855   50736    5833    90%    /usr/demo
athens:/usr/doc       105843   87253    8005    82%    /usr/doc
```

Filesystems at or above 90 percent of capacity should be cleared of unnecessary files. You can do this either by moving them to a disk or tape that is less full, using `cp` to copy them and `rm` to remove them. Or you can simply remove them outright. Of course, you should only perform housekeeping chores on files that you own.

Directory Usage and `du`

You can use `du` to display the usage of a directory and all its subdirectories in kilobytes; that is, units of 1024 bytes or characters.

`du` shows you the disk usage in each subdirectory. To get a list of subdirectories in a filesystem, `cd` to the pathname associated with that filesystem, and run the following pipeline:

```
venus% du | sort -r -n
5314     .
1155     ./Documents.new
818      ./SCCS
234      ./Programs.new
230      ./Reference.new
204      ./Reference.old
123      ./Library.new
89       ./Library.old
87       ./Users.Guide.old
49       ./Reports.old
27       ./Documents.old
5        ./Programs.old
```

This pipeline, which uses the *reverse* and *numeric* options of `sort`, pinpoints large directories. Use `ls -l` to look at the size (in bytes) and modification times of files within each directory. Or use `find` to locate files that exceed a given size. Old files, or text files over 100KB, often warrant storage off-line.

3.6. Making a Tape Archive With `tar`

To make a tape archive:

(1) Mount (insert) a fresh tape on the tape drive. If you don't know how to do this, see your system administrator or consult *System and Network Administration* for details.

(2) `cd` to a directory you wish to archive. If you wish to archive an entire hierarchy of files, `cd` to the topmost directory in that hierarchy. `tar` will archive the directory and all its subdirectories.

(3) Type the `tar` command:

```
tar cvf drive
```

Most options to the `tar` command do not take minus signs (-).

The `c` option tells `tar` to *create* a new tape archive and *overwrite* the previous contents of the tape. The `v` stands for *verbose*. `tar` tells you everything that it is doing. The `f` tells `tar` to put the archive on the file *drive* (the tape drive is considered a file). Your system administrator can tell you the name of a tape drive to use.

Tapes can be reused. If you do not wish to overwrite the previous contents, you can use `r` rather than `c`. With `r`, `tar` skips to the end of the previous archive, and then adds files to the end. If you want to conserve space on the tape, you can use `u`.[15] With `u`, `tar` replaces files whose contents have changed with their newest version, adds new files onto the end, and leaves untouched files alone.

drive can be a regular file. Since `tar` output takes up less space than do text files, a tape archive on disk can provide some space savings and a bit more convenience than using an actual tape. For even more space reduction, run the tape

[15] The `r` and `u` options do not work with quarter-inch cassettes; they work only with half-inch tape drives. See the `mt` command for quarter-inch tapes.

archive file, or *tarfile* through `compact`.[16]

Looking at the Contents of a Tape Archive

To examine the contents of a `tar` tape archive, use the `t` option:

```
tar tvf drive
```

To search for a specific file on the tape, pipe the output of `tar t` through `grep`. (See *SunOS User's Guide: Getting Started* for a discussion of `grep`.)

Extracting Files From a Tape Archive

To extract files from a tape archive, `cd` to the directory in which to place the file, mount the tape, and then use the `tar x` option:

```
tar xvf drive filename
```

If you omit the filename, `tar` extracts the contents of the entire tape. If you specify a a list of filenames, `tar` extracts the named files.

[16] The command `uncompact` restores the tarfile to its original state, and you can then use `tar` to retrieve files from within the tarfile just like you would from a tape drive.

4

More on the C Shell

This chapter continues the discussion of the C shell begun in *SunOS User's Guide: Getting Started*, where many of the basic concepts and commands associated with the C shell were introduced. If you have not read that discussion, you should do so now.

In the sections below, you will broaden your knowledge of command line editing. You will also learn the rudiments of using variables in the C shell. Command routines called scripts are introduced in Appendix B.

The SunOS operating system also offers the Bourne shell, which runs faster and has a simpler syntax for writing scripts. See Appendix C.

4.1. Command Line Editing (Continued)

SunOS User's Guide: Getting Started presented several ways of editing a command line and repeating all or part of a command. There, you learned to substitute one string for another with `^`*`old`*`^`*`new`* and `^`*`old`*`^`*`new`*`^:p`.

A word on the command line that begins with an exclamation is referred to as an *event designator*. An event designator can stand for a previous command or selected words from a previous command line.

As you have already seen, you can repeat the most recent command by typing two exclamation points (`!!`). And you have seen how to specify the last word of a command with `!$`.

There are several other ways to modify a recent command and thus perform a variety of tasks with just a few keystrokes. For instance, the history mechanism lets you repeat any command in the history list by typing an exclamation point, followed by its command line number,

`!`*`n`*

For example:

```
venus% history
    1  ls
    2  cd
    3  date
    4  history
venus% !3
date
Wed Aug 30 16:23:51 PDT 1989
venus%
```

You can specify the *n*th command *back,*

```
!-n
```

as in:

```
venus% !-3
cd
venus%
```

You can repeat an event by typing an exclamation point, followed by the first few characters that match it,

```
!string
```

where *string* is all or part of the command you're repeating

The history mechanism performs the first match it encounters. You may have to add a few characters to get the desired event. In this example the user wants to repeat the `clear` command (to clear the screen):

```
venus% history
11   ls -l
12   clear
13   cp *.dit /tmp
14   history
venus% !c
cp *.dit /tmp
[Ctrl-C]
venus%
```

Because the user typed in too few characters to specify the event precisely, `!c` matched the most recent event beginning with *c*, namely `cp`. The observant user interrupts it with [Ctrl-C] and then types in `!cl` to match the desired event:

```
venus% !cl
clear
(the screen clears)
```

There is, however, a limit to how many characters you may add to "disambiguate" the command: that limit is the first space. Whatever is typed in after the first space is interpreted as a further argument.

For example:

```
venus% history
   20  ls -a
   21  ls -l
   22  ls /etc
   23  history
venus% !ls -a
ls /etc -a
-a not found
venus%
```

Sometimes it's easier to match against a string of characters *embedded* within the event. To repeat a command in this way, use:

```
!?str?
```

where *str* is the embedded string to search for. For example:

```
venus% !?tmp?
cp *.dit /tmp
venus%
```

Selecting Words Within Events

Suppose that you want to apply several commands to a long list of files and you don't want to have to retype the list every time. `!*` repeats all arguments to the previous command (all but the first word of the command line). `!^` expands to the first argument.

For example, if the last command was

```
echo first
```

`!^` or `!:^` would expand to `first`. `!:`*n* expands to the *n*th argument (*n*+1 word). `!:$` expands to the last argument of the selected event.

`!:0` expands to the *zero* argument, which in the SunOS operating system is the command itself. So, for example, if you type

```
venus% more file1
```

you can then type:

```
venus% !:0 file2
more file2 (command expanded and performed)
```

You can select a specific word from a specific event by appending a *word designator* to its event designator. A word designator has the form of a colon, followed by a character. `:*` expands to all arguments in the event.

Let's refer back to the history list we've been using:

```
venus% history
    1  ls
    2  cd
    3  date
    4  history
    . . .
    11  ls -l
    12  clear
    13  cp *.dit /tmp
    14  history
    . . .
    20  ls -a
    21  ls -l
    22  ls /etc
    23  history
venus%
```

Using this list, to `mv` all files with the suffix `.dit` into the directory `/tmp`, you would type:

```
venus% mv !?tmp?:*
mv *.dit /tmp (command expanded and performed)
venus%
```

Modifying Selected Words and Events

You can edit the text of an event or word by appending an *event modifier* to it. A modifier starts with a colon, followed by one or more characters that indicate the actions to perform. `:s/`*`old`*`/`*`new`*`/` substitutes *new* for *old* in the first word where there is a match for *old*. When inserted between the colon and the modifier, a `g` indicates that the modifier applies to all designated words, not just the first. So to `mv` all `.dot` files into the directory `/tmp`, you would type:

```
venus% mv !?tmp?:*:gs/dit/dot/
mv *.dot /tmp (command expanded and performed)
```

As you learned in *SunOS User's Guide: Getting Started* , `:p` indicates that the event or word is to be expanded and echoed, but not performed. You can place several modifiers in an event or word designator. For instance:

`mv !?tmp?:*:gs/dot/dit/:p`

is echoed as

`mv *.dit /tmp`

but not performed.

For more information about event designators, word designators, and event modifiers, refer to Appendix A, C Shell Special Characters.

4.2. Variable Substitution

A *variable* is a named location in which to store text that you'd like the C shell to remember for you. You can use the `set` command to associate a variable name with a word to remember. A placeholder, composed of a dollar sign (`$`), followed by the name of a variable, is replaced with the contents of that variable by the C shell. Thus you can use a variable name, preceded by a `$`, as an abbreviation for its contents.

To assign a value to a variable, type in a command like:

```
venus% set testdir = ~/programs/test
```

To display that variable's contents:

```
venus% echo $testdir
~/programs/test
```

Suppose that you are working with files in two directories, each with very long, and very different pathnames:

```
/home/sam/sources/gfx/lines/module3
/home/bin/c/gfx/lines/module3
```

You can abbreviate these pathnames as follows:

```
set src = /home/sam/sources/gfx/lines/module3
set bin = /home/bin/c/gfx/lines/module3
```

Then, when you want to perform commands on files in these directories, you can use `$src` instead of `/home/sam/sources/gfx/lines/module3`, and `$bin` instead of `/home/bin/c/gfx/lines/module3` on the command line:

```
venus% cd $bin;pwd
/home/bin/c/gfx/lines/module3
venus% cd $src;pwd
/home/sam/sources/gfx/lines/module3
```

The `set` command with no arguments prints a list of all C shell variables and their current values. To see the value of a single variable, use a command of the form:

```
echo $variable
```

Storing Lists in C Shell Variables

In addition to single words, you can store a list of words in a C shell variable by enclosing the list in parentheses when you use the `set` command. One example of this is the `path` variable that you set in your `.cshrc` file.

Another might be:

```
venus% set mdirs = (/home/dakota/kitchen /home/dakota/gym)
venus% ls $mdirs
/home/dakota/gym:

aerobics      basketball    cars          dance

/home/dakota/kitchen:

anchovies     bagel         cabbages      doughnuts
venus%
```

Suppose that you just want to list those files in these directories which start with the letter *b*:

```
venus% ls $mdirs/b*
/home/dakota/gym/basketball

/home/dakota/kitchen:

anchovies     bagel         cabbages      doughnuts
venus%
```

This failed: `ls` lists the files starting with *b* in `/home/dakota/gym`, and *all* the files in `/home/dakota/kitchen`. This is because the */b** got appended to `mdirs` as a whole, and not to to each individual part of the variable. So typing

```
ls $mdirs/b*
```

is equivalent to typing

```
ls /home/dakota/kitchen /home/dakota/gym/b*
```

(You can operate on each member of a variable list by using the `foreach` command, described in the next section.)

You can select a specific word from the list by appending an *index* to the *call*[17] to the variable as follows:

```
$var[n]
```

where *var* is the name of the variable and *n* is a number indicating the position of the word within the list. Using the above example, the word `/home/dakota/gym` is the second word in the list. So the command:

[17] A call to a variable is the string you use to indicate that what you really want is the value it contains, in this case the name of the variable preceded by a dollar sign.

```
echo $mdirs[2]
```

displays the value

```
/home/dakota/gym
```

You can also specify a range:

```
venus% echo $mdirs[1-2]
/home/dakota/kitchen  /home/dakota/gym
venus%
```

But if you enclose a number in the braces that is higher than the count of words in the variable, you will get an error message. You can use filename substitution to simplify entering a list. The command:

```
set man = (/usr/man/{man,cat}?)
```

yields the following value:

```
venus% echo $man
/usr/man/man1 /usr/man/mann /usr/man/cat1
/usr/man/cat2 /usr/man/cat3 /usr/man/cat4
/usr/man/cat5 /usr/man/cat6 /usr/man/cat7
/usr/man/cat8 /usr/man/catl /usr/man/catn
```

which is a complete list of all the directories containing man page sources and formatted files.

Processing Lists With `foreach`

The `foreach` command provides a means to apply a set of commands successively for every word in a list. It prompts you for a set of commands, uses an *index* variable to store the current word while executing each pass through the commands, and repeats the list of commands once for each word in the list.

The syntax of the `foreach` command is:

```
foreach index (list)
```

where *index* is the name of the variable and *list* is a list of words. After you type [Return], `foreach` prompts for a command with a question mark. It continues to prompt for commands until you type the command `end` by itself after the question mark. This signifies the end of the loop.[18]

A *loop* is a set of commands to be repeated successively.

In a previous example, we tried unsuccessfully to list all the files beginning with the letter *b* in the directories contained in the variable `$mdirs`. `foreach` allows you to do this:

```
venus% foreach i ($mdirs)
? ls $i/b*
? end
basketball
bagel
venus%
```

In the next example, `*` is the filename metacharacter that represents all the files in a directory, and the `-n` option to `echo` is used to put all the output on the same line:

```
venus% foreach file (*)
? echo -n $file
? echo -n ", "
? end
```

The result is like using `ls`, except the files all appear on the same line, with a comma we specifically provided:

```
... file1, file2, file3, file4, ...
```

You can use variable substitution, as well as filename substitution symbols within the list.[19] Using the variable `man` defined above, the following `foreach` loop gives you a count of the source files and then the formatted files within each section of the man pages. As the loop proceeds, the value of the index variable (written as `$dir`) changes with each pass:

```
venus% foreach dir ($man)
? echo -n $dir
? ls $dir | wc -l
? end
/usr/man/manl          0
/usr/man/mann          0
/usr/man/cat1        486
/usr/man/cat2        176
/usr/man/cat3       1106
/usr/man/cat4         97
/usr/man/cat5        117
/usr/man/cat6         77
/usr/man/cat7         21
/usr/man/cat8        277
/usr/man/catl          0
/usr/man/catn          0
```

[19] This also works with the `set` command.

Predefined Variables

The C shell maintains a set of predefined variables. Some of these, like `noclobber`, are used by the C shell to affect the way it behaves. Others keep track of information that the C shell needs to know about. `home`, for instance, keeps a record of your home directory. If you change the value of `home`, and then use `cd` with no argument, the C shell attempts to change directories to that new value:

```
venus% set home=/
venus% cd;pwd
/
venus% set home=nonesuch
venus% cd;pwd
cd: Can't change to home directory.
venus% echo $home
nonesuch
venus% cd ~
nonesuch: No such file or directory
```

Environment Variables

The C shell also maintains a set of variables, called *environment* variables; you should be familiar with them from reading *SunOS User's Guide: Customizing Your Environment.* Environment variables are passed along to any commands or subshells. They are created and modified using the `setenv` command, which has a different syntax than has `set`:

```
setenv name value
```

There is no equal sign between the name of the variable and its value, as there is with `set`. And only one word (or string within quotes) can be assigned to an environment variable.

Environment variables are passed to all commands and programs run from within the current shell. C shell variables are only effective within the *current* shell.

Typically, the names of environment variables are given in all capitals. In some cases, there is a lower-case equivalent used by the C shell.

Others include `user` and `USER`, `term` and `TERM`, `shell` and `SHELL`, and `path` and `PATH`.

The environment variable `HOME` is such a case. When you use the `set` command to change the value of the (`home`) shell variable, the equivalent environment variable is also changed. When you use `setenv` to change the environment variable, however, the value of the `home` shell variable is not affected:

```
venus% set home=bogus
venus% echo $home
bogus
venus% echo $HOME
bogus
venus% setenv HOME /home/sam
venus% echo $home
bogus
venus% echo $HOME
/home/sam
venus% set home=$HOME
venus% echo $home
/home/sam
venus%
```

To get a list of all environment variables and their current values, use the command `printenv`.

Appendix

A

C Shell Special Characters

Characters with special meaning to the C shell:

`?`	Single character wild card.
`*`	String wild card, zero or more characters.
`.`	Abbreviation for current working directory.
`..`	Abbreviation for the parent of the current directory.
`~`	Abbreviation for your home directory.
`~` *user*	Abbreviation for the home directory of *user*.
`[...]`	Matches any single character listed within the brackets.
`[`*x*`-`*y*`]`	Matches any character within the range of *x* and *y*.
`{`*str*`, ...}`	*Grouping*. Matches each *str* successively. Filename substitution is applied to each *str* before matching occurs. Thus, `{x,*y*,?z*}` matches a filename `x`, all filenames containing the letter `y`, and all filenames having `z` as the second character. Groups enclosed with braces can be nested.
`&`	Places the command in the background.
Ctrl-Z	Stops the foreground job, placing it stopped in the background.
`%[`*n*`]`	Brings the current (stopped) job or the specified background job to the foreground.
`%[`*n*`] &`	Continues, in the background, the current or specified stopped job.
`>`*filename*	Redirects the standard output to *filename*. If *filename* already exists, its previous contents are lost. When set, the shell variable `noclobber` prevents redirection to existing files or character special devices.
`>!` *filename*	Forces the standard output to *filename*, even when `noclobber` is set.

>& *filename*
Routes diagnostic (standard error) output to *filename*, along with the standard output.

>& ! *filename*
Forces diagnostic and standard output to *filename*.

>> *filename*
Appends the standard output to *filename*. When `noclobber` is set, the file must already exist.

>> ! *filename*
Forces the standard output to *filename*, even when `noclobber` is set. Creates a new file if necessary.

>>& *filename*
Appends the diagnostic as well as standard output to *filename*. When `noclobber` is set, the file must already exist.

>>& ! *filename*
Forces appending of diagnostic and standard output to *filename*, even when `noclobber` is set.

cmd **|** *cmd*
Pipe. Uses the standard output of the left-hand *cmd* as standard input for the right-hand *cmd.*

cmd **|&** *cmd*
Uses both standard and diagnostic output of the left-hand *cmd* as standard input for the right-hand *cmd.*

(. . .) *Command grouping.* Commands and pipelines surrounded by parentheses are executed in a subshell and treated as a unit by the current C shell.

(. . .) >& *filename*
Redirects the standard output (if any) and the diagnostic output of the enclosed command(s) to *filename*. This is especially useful if the enclosed commands redirect the standard output to a file (thus sending the standard output and the standard error to separate destinations).

< *filename*
Opens *filename* as the standard input.

cmd **<<** *word*
Here document. Indicates that a command (typically interactive) is to accept *its* commands from the same device or file (usually a script) as the shell. *word* is interpreted literally as the *end-of-input* mark for the command. The C shell parses, but does not execute, each text line between the here document and a line containing *word* by itself. After applying command, filename, and variable substitution, the C shell passes each line on to *cmd.* To suppress all substitution, include a \, ", or ′ in *word.*

;	Separates commands on one input line.
\	At the end of a line, escapes the newline character and continues the command to the next input line.
\	Escape the special meaning of the character it precedes.
´...´	The C shell treats the enclosed text as one word, preventing variable and history substitution (except !, which can be escaped by \).
"..."	The C shell treats the enclosed text as one word, breaking words only at enclosed newlines.[20] History and variable substitution is performed *before* escape characters are interpreted.
\`*command*\`	Replaces the backquoted command or pipeline (including the backquote marks) with its output. Output is broken into words at blanks, tabs and newlines, except for the final newline. Unless the right-hand backquote is followed by a space, the last word of the substitution is prepended to the following word on the command line.

Escaped history substitution event designators and word designators (described below) can be used to indicate command line arguments within an alias definition.

^*l*^*r*[^]	Substitutes the string *r* for the string *l* in the previous command line. The final ^ is required only if history substitution modifiers are appended.
!	Begins a history substitution. To escape its special meaning, precede the ! with a backslash (\). An ! is also escaped when followed by a blank, tab, newline, (or =.

The following designators select an event (command line) from the history list. Word designators and modifiers can be appended for command-line editing.

!!	The previous command.
!*n*	Command line number *n*.
!–*n*	Selects the event whose number is *n* less than the current one.
!*str*	The most recent command beginning with *str*.
!?*str*[?]	The most recent command containing *str*. The closing question mark is only required when word designators or modifiers are appended.
!*	All arguments from the previous command, but not argument zero (the command name).

[20] An *enclosed newline* is a carriage return within quotes; i.e., an *escaped* newline.

!^	The first *argument* from the previous command. If, for instance, the command was `echo first`, then `!^` would expand to `first`.
!$	The last argument from the previous command.
!:*n*	The *n*th argument from the previous command.
!#	The contents of the *current* command line typed in so far.
!{*str***}** ...	Restrict the event designation to *str*; text following the brackets is appended to the last word of the expansion *after* substitution takes place.

Word designators can be appended to the history substitution character (`!` for the previous event) to a quick substitution, or to an event designator.

:*	All arguments, except argument zero.
:^	The first argument .
:$	The last argument.
:*n*	The *n*th argument.
:%	The word matched by most recent `!?` search.
:*x–y*	Argument *x* through argument *y*.
:–*y*	abbreviates **:0–***y*.
:*x***	Argument *x* through the last argument.
:*x–*	Argument *x* through the next-to-last argument.
:#	The contents of the *current* command line typed in so far.

The following modifiers can be used in any sequence to modify a selected event or word. A colon is required to separate modifier(s) from event or word designators.

[:]**p**	Prints the new command but does not execute it.
[:]**h**	Removes a trailing pathname component, leaving the head.
[:]**t**	Removes all leading pathname components, leaving the tail.
[:]**r**	Removes a filename extension (*.xxx*).
[:]**e**	Removes all but the extension.
[:]**s/***l***/***r***/**	Substitutes *r* for *l*. *l* is a literal string, not a regular expression. Any character may be used as the delimiter in place of `/`. The character `&` in the right hand side is replaced by the left hand string. A null *l* uses the previous string either from a *l* or from a `?` event search.
[:]**&**	Repeats the previous substitution.
[:]**q**	Quotes the substituted words, preventing further substitutions.

[:]x	Like :q, but breaks words at blanks, tabs and newlines.
:g*m*...	*Global prefix.* When prefixed any of the above modifiers, *m*, the modifier(s) apply to all words in the specified event. Normally, each word must be modified separately.

After the input line is aliased and parsed, and before each command is executed, the C shell performs variable substitution on words that start with an unescaped $, according to the list below. A $ is escaped by preceding it with a backslash (\), or when followed by a blank, tab, or end-of-line.

Shell variables have names consisting of up to 20 letters, digits and underscore characters, starting with a letter.

Environment variables can be expanded but not modified.

$*var* Is replaced with the value of *var*.

${*var***}**... The brackets indicate that the enclosed string is the variable name. The value of the named variable is prepended to the text that follows on the command line.

${ *var* **[***selector***] }**
Select words from within *var*. *selector* can be one of:

n	a number.
x–*y*	two numbers separated by a – to specify a range.
x –	Word *x* through the last word.
–*y*	The first word through word *y*.
*	all words in the value.
$*var*	the value of another variable, in which case variable substitution is applied to the *selector* first, and then to the entire word.

$#*var*	The number of words in the variable.
${#*var***}**	Same as $#*var*
$0	The name of the file from which command input is being read. An error occurs if the name is not known.
$*n*	The *n*th word in the argument list; equivalent to $argv[*n*] .
${*n***}**	Same as $*n*.
$*	All words in the argument list; equivalent to $argv[*].
$?*var*	
${?*var***}**	replaced with 1 if *var* is set, or 0 if not.
$?0	replaced with 1 if the current input filename is known, 0, otherwise.
$$	replaced with the process ID (PID) of the (parent) shell.

$<	replaced with text taken from the standard input, with no further interpretation. Used to read from the keyboard in a C shell script.

The modifiers `[:]h`, `[:]t`, `[:]r`, `[:]q`, and `[:]x` can be applied to the substitutions above. See "Modifiers" under "History Substitution," above, for a description.

If braces `{...}` appear in the variable substitution, modifiers must be enclosed within them.

The current implementation allows only one modifier within each variable substitution.

The following variable substitutions can not be modified: **$?**, **$$**, and **$<**.

Expressions

Expressions appear within the `@`, `exit`, `if`, and `while` builtin commands.

Null or missing terms are interpreted as `0`.

Results of all expressions are *strings* that represent decimal numbers. Results of logical expressions are `1` (for true) or `0` (for false).

(...)	Parentheses indicate grouping of operators and terms within an expression, overriding the standard precedence of operators.
==	True if the string on the left is equal to the string on the right (after all substitutions are performed).
!=	True if the string on the left is not equal to the string on the right.
=~	True if the string on the left is matched by the pattern on the right.
!~	True if the string on the left is not matched by the pattern on the right.
<	True if the number on the left is less than the number on the right.
<=	True if the number on the left is less than or equal to the number on the right.
>	True if the number on the left is greater than the number on the right.
>=	True if the number on the left is greater than or equal to the number on the right.
||	Logical *or* connective.
&&	Logical *and* connective.
{...}	*Command successful.* True if the command surrounded by brackets exits with status code `0`.

An operator of the form

flag filename

is true if the attribute *flag* applies to *filename*, with respect to the current user. *flag* can be one of:

-r	read access

`-w`	write access
`-x`	execute access
`-e`	existence
`-o`	ownership
`-z`	zero size
`-f`	plain file
`-d`	directory
`!` *flag*	true if *flag* does *not* apply.

If the file does not exist, or is inaccessible, then all inquiries yield false as a result.

`+`	Addition.
`–`	Subtraction.
`*`	Multiplication.
`/`	Division.
`%`	Remainder after division.
`0`*str*	A string with a leading zero is interpreted as an octal numeral.
`<<`	Bitwise *shift left* operator.
`>>`	Bitwise *shift right* operator.
`|`	Bitwise *or* operator.
`^`	Bitwise *exclusive or* operator.
`&`	Bitwise *and* operator.

B

C Shell Scripts

You can put a sequence of SunOS commands in a file called a *script*. By using the `source` *filename* command, or by setting the execute permissions and typing in the filename as if it were a command, you can tell the C shell to read and perform commands in the file.

NOTE *We recommend that you use the Bourne shell for writing shell scripts. The Bourne shell has a simpler command syntax, faster execution time, and provides better security. Refer to Appendix C for information about writing Bourne shell scripts.*

This appendix outlines features that you can use when writing scripts for the C shell.

C Shell Invocation

C shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the C shell is more general in scope. Scripts do not check for dependencies, for instance. But there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

When a script is invoked by name, the system looks at the very first line of the file to decide how to run it:

- If the first line of the script starts with a `#!`, followed by the name of a program, the system uses that program to perform commands in the script.
- If the first line starts with a # (hash sign), the system uses the C shell to run the script.
- If the first line does *not* start with a # (hash sign), the system uses the Bourne shell to run the script.

To run a script with no C shell startup processing, the first line should be of the form:

```
#! csh -f script
```

Command-Line Arguments in Scripts

To pass command-line arguments as parameters to a script, type its name, followed by any arguments you wish. The C shell places words following the name in the variable `argv`, the *arguments list*. Command-line arguments are treated as words contained in this variable, or you can use the equivalent variables: `$1` through `$`*n* where *n* is the number of arguments in the list.

Variables in Scripts

A number of notations are available for accessing words in variables, and other variable attributes. The notation:

```
$?name
```

expands to `1` if a named variable exists (using the `set` command) or to `0` otherwise:

```
venus% set var=(a b c)
venus% echo $?var
1
venus% unset var
venus% echo $?var
0
```

All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the count of words in the variable *name*:

```
venus% set var=(a b c)
venus% echo $#var
3
venus% unset var
venus% echo $#var
var: Undefined variable.
```

There is a special C shell variable, `$$`, which represents the process number of the shell itself. Why would you want a variable like this? Because the shell's process number is unique on the system, you can use it as part of a file's name if you want to create unique temporary files from inside the shell. Part of your script might create a file called `/tmp.$$`, for example; this file will not be confused with any other that might already exist.

The redirection characters:

```
$<
```

indicate that a line is to be read from the terminal. To write out the prompt `yes or no?` without a newline and then read the answer into the variable `a`:

```
echo -n "yes or no?"
set a=($<)
```

In this case `$#a` would be `0` if either a blank line or Ctrl-D were typed in response.

A minor difference between `$`*n* and `$argv[`*n*`]` is that `$argv[`*n*`]` yields an error if *n* is larger than the word count `$#argv`, while `$n` never yields a subscript-out-of-range error. This is for compatibility with older shells.

It is never an error to give a subrange of the form *var*`[`*n*–`]`. If there are less than *n* words in the given variable, then no words are selected.

A range of the form *var*`[`*m*–*n*`]` likewise returns a value without an error, even when *m* exceeds the number of words, provided that *n* is in range.

Expressions

All of the arithmetic operations of the C language are available in the C shell with the same precedence that they have in C. These operations are useful for evaluating expressions in branches and loops. The operations `==` and `!=` compare strings, and the operators `&&` and `||` implement the logical *and* and *or* operations, respectively. The operators `=~` and `!~` are similar to `==` and `!=`, allowing for pattern matching as with filename substitution.

File Enquiries

The expression:

```
-e filename
```

returns `1` if the file exists and `0` otherwise. Similar primitives provide other tests:

`-r` returns `1` if read-access is allowed for the user running the script.

`-w` returns `1` if write-access is allowed for the user.

`-x` returns `1` if execute-access is allowed.

`-o` returns `1` if the user owns the file.

`-z` returns `1` if the file has zero length.

`-f` returns `1` if a plain file.

`-d` returns `1` if a directory.

Pathname Processing Primitives

There are also primitives to apply to pathnames to strip off unneeded components:

`:t` (*tail*) removes all but the rightmost component (or *basename*) of the pathname.

`:r` (*root*) removes suffixes beginning with a dot (`.`).

`:e` (*end*) removes prefixes ending with a dot.

`:h` (*head*) removes the last component, leaving the pathname of the directory in which the file resides.

Here's an example of how these apply to a file:

If you had a file called `/usr/include/sys/types.h`, then `:t` would remove all but `types.h`; `:r` would leave you with `/usr/include/sys/types`; `:e` would leave you with just `h`; and `:h` would give you `/usr/include/sys`.

Return Codes

It is possible to test whether a command terminates normally by using a primitive of the form `{ command }`, which returns `1` if the command exits normally (with exit status `0`), or `0` if the command terminates abnormally (with a nonzero return code).

If more detailed information about the status of a command is required, it can be executed and the variable `status` examined in the next command. Since every command returns a value to `status`, you must save values of interest on the very next line of the script:

```
set checkpoint=$status
```

where *checkpoint* is a suitable variable name.

Sample C Shell Script

The following script, `copyc`, copies files named as arguments into a backup directory:

Figure B-1 *A Sample C Shell Script*

```
#
# copyc copies files named on the command line
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

Basic Control Structures: `if` and `foreach`

This script uses the `foreach` command, which causes the C shell to execute the commands between it and the corresponding `end` with the named variable taking on each of the values given between `(` and `)`. The named variable — in this case `i` — is set to successive words in the list. Within this loop you can use the `break` command to stop executing the loop and `continue` to terminate one iteration and begin the next. After the `foreach` loop, the iteration variable (`i` in this case) has the value it had during the last iteration.

The variable `noglob` is set to prevent filename expansion from being performed on members of `argv`. This is a good idea, in general, if the arguments to a C shell script are filenames that have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a `$` variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form:

```
if ( expression ) then
command
...
endif
```

The placement of the keywords here is *not* flexible. The word `then` *must* appear on the same line as `if`, when used with a block of commands.

The C shell does *not* accept the formats:

```
if ( expression )
then
```

or

```
if ( expression ) then command endif
```

For individual conditional commands, the C shell has another form of the `if` statement:

```
if ( expression ) command
```

which can also be written as

```
if ( expression ) \
     command
```

The newline is escaped here for the sake of appearance. The command must not involve `|`, `&` or `;` and must not be another control command. The final \ must immediately precede the end-of-line. This is the only form of the `if` command that can be used within an alias definition.

The more general `if` statement also admits a sequence of `else–if` pairs followed by a single `else` and an `endif`.

```
if ( expression ) then
     commands
else if ( expression ) then
     commands
...
else
     commands
endif
```

Introducing Comments With

The # character introduces a C shell comment in a script (but not from the terminal), and the C shell ignores all subsequent characters the line.

Other C Shell Control Structures

The C shell also has the control structures `while` and `switch`, which are similar to those in C.

```
while ( expression )
    commands
end
```

and

```
switch ( word )

case str_1:
    commands
    breaksw

  . . .

case str_n:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

See the `csh` man page for details. C programmers should note that `breaksw` exits from a `switch`, while `break` exits a `while` or `foreach` loop.

Finally, `csh` allows a `goto` statement, with labels looking as they do in C, that is:

```
loop:
    commands
    goto loop
```

Here Documents

A *here document* is a special notation used to pass instruction along to commands that normally run interactively. The here document begins with a <<*eot* and ends with a line containing *eot* by itself. *eot* can be any string.

Here is a script that runs `ed` to delete leading blanks from every line in each file in the argument list. In this case, the *eot* string is "woof":

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'woof'
1,$s/^[	 ]*//
w
q
'woof'
end
```

(The brackets in the script contain a tab and a space.)

The notation << `'woof'` means that the standard input for the `ed` command is the text in the C shell script file up to the next line consisting of exactly `'woof'`. The fact that the `woof` is enclosed in quote characters prevents the C shell from substituting variables on the intervening lines. In general, the C shell uses the word following << to terminate the text to be given to the command. If any part of the word following the << is quoted, these substitutions are not performed. In this case, since the form `1,$` was used in the editor script, you needed to ensure that the `$` is not variable-substituted. You can also ensure this by preceding the `$` here with a \, for instance:

```
1,\$s/^[ ]*//
```

but quoting the `woof` terminator is a more reliable way of achieving the same effect.

Catching Interrupts With `onintr`

If your script creates temporary files, you can use `onintr` to catch interrupts, so that the script can delete them before halting.

```
onintr label
```

where *label* is a label in your program that is followed by your housekeeping commands. If the C shell receives an interrupt, it performs a `goto` *label*, and executes those commands.

Exit

You can also use the `exit` command (which is built in to the C shell) to terminate the script. If you wish to exit with a nonzero status, do the following:

```
exit(status)
```

where *status* is the status you want to exit with.

C

Bourne Shell Scripts

You can use the Bourne shell to perform a set of SunOS commands contained in a file called a *script*.

Bourne shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the Bourne shell is more general in scope. Scripts do not check for dependencies, for instance. But there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

To run a Bourne shell script (for which you have execute permission), type in its filename as if it were a command. When you do, the system looks at the very first line of the file to decide which shell should run the script:

- If the first line does *not* start with a # (hash sign), the system uses the Bourne shell to run the script.
- If the first line starts with a # (hash sign) and is *not* followed by a `!` (exclamation mark), the system uses the C shell to run the script.
- Finally, if the first line of the shell script starts with a `#!` combination and is followed immediately by a name, the system looks for a program of that name to run the shell script. If you supply arguments on the command line, these are passed along to variables in the Bourne shell called *arguments*. The first argument after the name of the script is placed in variable `1`. The second is placed in variable `2`, and so forth.

NOTE *You can often simplify testing of Bourne shell scripts (or commands to run within them) by using the Bourne shell interactively. To do so, type in the command* `/bin/sh`, *and enter commands as described in this chapter. Use* [Ctrl-D] *to exit and return to the C shell. Most of the examples below make use of the Bourne shell interactively, as well as within scripts.*

Bourne Shell Variables

The Bourne shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. You may assign values to variables by writing the variables name, an equal sign, and a value (with no spaces between). For example:

```
$ user=fred box=m000 acct=mh0000
```

assigns values to the variables *user*, *box* and *acct*. To set a variable to the null string, you can say:

```
$ cheese=
```

The value of a variable is substituted by preceding its name with `$` — for example:

```
$ name=fred
$ echo $name
fred
$
```

You can use variables to provide abbreviations for strings that are used frequently throughout a script. A script containing the following lines

```
b=/home/fred/bin
...
mv pgm $b
```

moves the file `pgm` from the current directory to the directory `/home/fred/bin`. A more general notation is available for parameter (or variable) substitution, as in:

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed immediately by a letter or digit:

```
tmp=/tmp/ps
ps >${tmp}a
```

directs the output of `ps` to the file `/tmp/psa`.

Variables can be concatenated onto each other. If the variable `x` is set to *hello*, then `$x.foo` will be equal to `hello.foo`.

Bourne Shell Initial Variables

Except for `$?`, the variables defined in table C-1 are set initially by the Bourne shell. `$?` is set after executing each command.

Table C-1 *Variables Initialized by the Bourne Shell*

Variable	***Explanation***
`$?`	The exit status (return code) of the last command executed, as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned.
`$#`	The number of arguments (in decimal).
`$$`	The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example, `tmp.$$` will not be confused with any other file.
`$!`	The process number of the last process run in the background (in decimal).
`$-`	The current Bourne shell flags, such as `-x` and `-v` .

Variables With Special Meaning to the Bourne Shell

Some variables have a special meaning to the Bourne shell; avoid them in general use.

`$MAIL` When the Bourne shell is used interactively, it looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the Bourne shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file `.profile` in your home directory. For example:

The file `.profile` in your home directory is the setup file for the Bourne shell — equivalent to the combination of the `.cshrc` and `.login` files for the C shell.

```
MAIL=/var/spool/mail/fred
```

`$HOME` Your home directory; this variable is also typically set in `.profile`.

`$PATH` A list of directories that contain commands (the *search path*). Each time the Bourne shell executes a command, a list of directories is searched for an executable file by that name. If `PATH` is not set, then the current directory, `/bin`, and `/usr/bin` are searched by default. `$PATH` consists of directory names separated by `:`. For example,

```
PATH=/home/fred/bin:/bin:/usr/bin:
```

specifies that `/home/fred/bin`, `/bin`, and `/usr/bin` are to be searched in that order, followed by the current directory (the null string after the last : in the example above; a dot (.) is equivalent to the null string). This allows you to have your own private commands accessible independently of the current directory. If the command name starts with a `/`, then this directory search is not used.

`$PS1` The primary Bourne shell prompt string, by default, `$`.

`$PS2` The Bourne shell prompt when further input is needed, by default, >.

`$IFS` The set of characters to be interpreted as blanks (field separators) when parsing command lines.

The `test` Command

Although the `test` command is not part of the Bourne shell, scripts frequently use it. `test` can be used to check on the status of files, to compare strings and algebraic expressions, and to perform integer calculations. For instance:

```
test -f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general `test` evaluates a predicate and returns the result as its exit status. Here is the list of things you can test for.

`-b` *file*	true if *file* exists and is a block special device.
`-c` *file*	true if *file* exists and is a character special device.
`-d` *file*	true if *file* exists exists and is a directory.
`-f` *file*	true if *file* exists and is not a directory.
`-g` *file*	true if *file* exists and is setgid.
`-h` *file*	true if *file* exists and is a symbolic link.
`-k` *file*	true if *file* exists and its sticky bit is set.
`-l` *string*	the length of *string*.
`-n` *string*	true if the length of *string* is nonzero.
`-r` *file*	true if *file* exists and is readable.
`-s` *file*	true if *file* exists and has a size greater than zero.
`-t` [*fildes*]	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
`-w` *file*	true if *file* exists and is writable.
`-x` *file*	true if *file* exists and is executable.
`-z` *string*	true if the length of *string* is zero.
string-1 `=` *string-2*	true if the strings *string-1* and *string-2* are equal.
string-1 `!=` *string-2*	true if the strings *string-1* and *string-2* are not equal.
string	true if *string* is not the null string.
n1 `-eq` *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons `-ne`, `-gt`, `-ge`, `-lt`, or `-le` may be used in place of `-eq`, where `ne` means "not equal", `-ge` means "greater than or equal to", `-lt` means "less than," and so on.

Alternative Form of the `test` Command: [. . .]

You can also call `test` by surrounding the expression to be tested with brackets (`[ ]`). (The left bracket is a command name, the right bracket is a argument signifying the end of the expression.) The left bracket must be followed by a blank, and the right bracket must be preceded by one. This form is most often used with the `if` command described later on.

Getting Started: A Simple Script

Here is a very simple Bourne shell script to look up names in a list of names and telephone numbers contained in a file called `names.list`. Let's call the lookup script `name`:

```
$ cat name
#! /bin/sh
grep -i $1 names.list
$
```

This is about as simple as you can get. Let's run the `name` script looking for people called *Ted*:

```
$ name ted
Ted Applehead            teda@seeds               7534
Ted Monsterpie           random@house             7256
$
```

Later on, we will show a more sophisticated version of `name` and expand on this script to demonstrate other features of the Bourne shell.

Control Flow in the Bourne Shell: `for`

A frequent use of Bourne shell script is to loop through the arguments (`$1, $2, ...`) executing commands once for each argument. Here's an expanded version of the `name` script from above. The original version of `name` can only look for one person's name. Now we want to expand it to look for more than one name at a time. Let's look at the new version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person names.list
done
$
```

Here we set a variable called `person` to the value of each argument, one at a time, then we call out the value of `person` in the `grep` command. Now we can look for more than one name at a time:

```
$ name ben madge
Ben Tortcake             tort@icky                7258
Madge Hittite            celtics@garden           7214
$
```

General Form of the `for` Loop

The `for` loop notation is recognized by the Bourne shell and has the general form

for *name* **in** *w1 w2* ...
do *command-list*
done

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like `do` and `done` are only recognized following a newline or semicolon. *name* is a variable that is set to the words *w1 w2* ... in turn each time the *command-list* following `do` is executed. If `in` *w1 w2* ... is omitted, then the loop is executed once for each argument; that is, `in $*` is assumed.

An example of the use of the `for` loop is the *create* command whose text is

```
for i; do >$i; done
```

(Remember that `cat` >*filename* creates a file where none exists.)[21] The command:

```
$ create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. Use the notation *>file* on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before `done`.

Control Flow in the Bourne Shell: `case`

The `case` notation provides a multi-way branch.For example, suppose you wrote a script called `append` that contained the following lines:

```
case $# in
    1)  cat >>$1 ;;
    2)  cat >>$2 <$1 ;;
    *)  echo 'usage: append [ from ] to' ;;
esac
```

esac, you may have noticed, is *case* backwards.

When called with one argument as

```
$ append file
```

`$#` is the string "`1`" and the standard input is copied onto the end of *file* using the *cat* command. To append the contents of *file1* onto *file2*, say:

```
$ append file1 file2
$
```

If the number of arguments supplied to *append* is other than 1 or 2, a message is displayed indicating proper usage.

The general form of the `case` command is:

[21] In fact, in the Bourne shell, you don't need `cat`; typing >*filename* by itself creates a file.

```
case word in
     pattern-1)  command-list-1;;
     pattern-2)  command-list-2;;
     . . .
esac
```

The Bourne shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found the associated *command-list* is executed, and execution of the case is complete. Since * is the pattern that matches any string, you can use it for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example, the commands following the second * are never executed.

```
case $# in
     *) . . . ;;
     *) . . . ;;
esac
```

Another example of the use of the case construction is to distinguish between different forms of an argument. The following example is a fragment of a cc (C compiler) command:

```
for i
do case $i in
     -[ocs]) . . . ;;
     -*) echo 'unknown flag $i' ;;
     *.c)    /lib/c0 $i . . . ;;
     *)  echo 'unexpected argument $i' ;;
     esac
done
```

What does this do? It checks for the options (or *flags*) -o, -c, or -s; if it gets some other flag, it reports it as unknown. It checks to see if it gets a file ending in .c and processes it when it does; if it gets anything else it reports an unexpected argument.

Matching Multiple Patterns in One Case

To allow the same commands to be associated with more than one pattern the case command provides for alternative patterns separated by a '|'. For example:

```
case $i in
     -x|-y)  . . .
esac
```

is equivalent to

```
case $i in
     -[xy])  . . .
esac
```

The usual quoting conventions apply, so that

```
case $i in
     \?)    . . .
```

will match the character ?.

Here Documents in the Bourne Shell

Sometimes a shell script requires data. Instead of having the data in some file somewhere in the system, the data can be included as part of the shell script. Such a collection of data is called a *here document* — the data (document) is right *here* in the shell script. One advantage of a here document is that shell parameters can be substituted in the document as the shell is reading the data.

The general form of a here document is like this:

```
        lines of shell commands
        . . .
command-name << end-marker
lines of data
belonging to the
here document
        . . .
end-marker
        . . .
        more lines of shell commands
```

The name Command Using Here Document

Let's revisit the name script discussed in earlier sections. Instead of having the names and numbers in one file and the shell script in another file, you can keep both the script and the list in the same file — that is, in the script. Here's another version of the name command:

```
$ cat name
#! /bin/sh
grep -i $1 <<woof
Ted Applehead           teda@seeds              7534
Bernice Barns           boat@carib              7441
    . . .
    more names
    . . .
David Smiter            acme@nadir              7435
Ben Tortcake            tort@icky               7258
Dave von Noknock        dave@dove               7296
woof
$
```

In this example the Bourne shell takes the lines between <<woof and woof as the standard input for grep. The string woof is arbitrary, the document being terminated by a line that consists of the string following <<.

Now you'll notice that in *this* version of name we're back to being able to only look up one name at a time. We *could* combine the multiple-name version with the here-document version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person <<woof
Ted Applehead           teda@seeds               7534
Bernice Barns           boat@carib               7441
    . . .
    more names
    . . .
David Smiter            acme@nadir               7435
Ben Tortcake            tort@icky                7258
Dave von Noknock        dave@dove                7296
woof
done
$
```

The problem with this approach is that the shell reads up the list of names every time around the `for` loop. This could become excruciatingly slow. In a later section we show another version of `name` using temporary files for faster performance.

Parameter Substitution in Here Documents

Parameters are substituted in the here document before it is made available to whatever command as illustrated by the following script, called `edg` (`ed` globally).

```
ed $3 <<woof
g/$1/s//$2/g
w
woof
```

Then the command line:

```
$ edg string1 string2 file
```

is equivalent to the command:

```
$ ed file
g/string1/s//string2/g
w
```

and changes all occurrences of *string1* in *file* to *string2*. You can prevent substitution by using \ to quote the special character `$` as in

```
ed $3 <<woof
1,\$s/$1/$2/g
w
woof
```

This version of `edg` is equivalent to the first except that `ed` displays a `?` if there are no occurrences of the string `$1`. Quoting the terminating string prevents substitution entirely within a here document, for example:

```
grep $i <<\#
. . .
#
```

In this case the shell does not try to replace the # with anything.

The document is presented without modification to `grep`. If parameter substitution is not required in a *here* document, this latter form is more efficient.

Control Flow in the Bourne Shell: `while`

The actions of the `for` loop and the `case` branch are determined by data available to the Bourne shell. Also provided are a `while` or `until` loop and an `if then else` branch whose actions are determined by the exit status returned by commands. A `while` loop has the general form

```
while command-list-1
do command-list-2
done
```

The value tested by the `while` command is the exit status of the last simple command following `while`. Each time round the loop *command-list-1* is executed; if a zero exit status is returned then *command-list-2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do . . .
    shift
done
```

is equivalent to

```
for i
do . . .
done
```

`shift` is a Bourne shell command that renames the arguments `$2, $3, . . .` as `$1, $2, . . .` and discards `$1`.

Another kind of use for the `while/until` loop is to wait until some external event occurs and then run some commands. In an `until` loop the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. Presumably another process will eventually create the file.

Control Flow in the Bourne Shell: `if`

A general conditional branch of the form

```
if command-list
then    command-list
else    command-list
fi
```

is also available to test the value returned by the last simple command following if.

We can illustrate a very simple use of the `if` command by expanding on our `name` script from before. The relevant change is in the first few lines (remember that `-lt` means *less than*):

```
$ cat name
#! /bin/sh
if test $# -lt 1
then
        echo Usage: $0 name ...
        exit 1
fi
grep -i $1 <<woof
Ted Applehead           teda@seeds               7534
Bernice Barns           boat@carib               7441
    . . .
    more names
    . . .
David Smiter            acme@nadir               7435
Ben Tortcake            tort@icky                7258
Dave von Noknock        dave@dove                7296
woof
$
```

The change here is the `if` command — the original version of the script didn't check that the user supplied any parameters at all. This version checks the number of parameters (`$#`) using the `test` command and displays a *usage* message if there are no parameters to remind the user of the correct way to use the script.

We mentioned earlier that the `test` command can also be written as `[`. Here is the first couple of lines of the `name` script above rewritten in that way:

```
$ cat name
#! /bin/sh
if [ $# -lt 1 ]; then
        echo Usage: $0 name ...
        exit 1
fi
grep -i $1 <<woof
    . . .
woof
$
```

The `if` command may also be used in conjunction with the `test` command to test for the existence of a file as in

```
if test -f file
then    process file
else    do something else
fi
```

Here is an example of the `test` command in action. This is an extract from the `diff3` shell script:

```
$ cat -n /usr/bin/diff3
     1  #! /bin/sh
     2  e=
     3  case $1 in
     4  -*)
     5    e=$1
     6    shift;;
     7  esac
     8  if test $# = 3 -a -f $1 -a -f $2 -a -f $3
     9  then
    10    :
    11  else
    12    echo usage: diff3 file1 file2 file3 1>&2
    13    exit
    14  fi
    15  trap "rm -f /tmp/d3[ab]$$" 0 1 2 13 15
    16  diff $1 $3 >/tmp/d3a$$
    17  diff $2 $3 >/tmp/d3b$$
    18  /usr/lib/diff3 $e /tmp/d3[ab]$$ $1 $2 $3
```

The relevant line is number 8, which reads

```
if test $# = 3 -a -f $1 -a -f $2 -a -f $3
```

This says that if the number of parameters (`$#`) is equal to 3 and all three parameters are files, the script can continue; otherwise, the script displays an error message and stops. (The `-a` is a *logical and operator*; it joins statements just like the word *and*.)

`elif`: Multiple-Test Version of `if`

A multiple-test `if` command of the form

```
if ...
then    ...
else    if ...
    then    ...
    else    if ...
        ...
        fi
    fi
fi
```

may be written using an extension of the `if` notation:

```
if condition-1
then    actions-1
elif    condition-2
then    actions-2
elif    condition-3
. . .
fi
```

The sequence

```
if command-1
then    command-2
fi
```

may be written this way (the `&&` is a logical *and*):

```
command-1 && command-2
```

This means that *command-2* will be executed only if *command-1* succeeds.

Conversely,

```
command-1 || command-2
```

executes *command-2* only if *command-1* fails (the `||` is a logical *or*). In each case the value returned is that of the last simple command executed.

Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first, *command-list* is simply executed. (The semi-colon is necessary to indicate the end of *command-list*.) The second form executes *command-list* as a separate process. For example,

```
$ (cd x; rm junk )
$
```

executes `rm junk` in the directory `x` without changing the current directory of the invoking shell.

The commands

```
$ cd x; rm junk
$
```

have the same effect but leave the invoking shell in the directory `x`.

Debugging Bourne Shell Scripts

The Bourne shell provides two tracing mechanisms to help in debugging shell scripts. The first is invoked within a script as

```
set -v
```

(v for *verbose*) and displays lines of the script as they are read. It is useful to help isolate syntax errors. It may be invoked within a script, or when the script is run, by saying

```
$ sh -v proc . . .
$
```

where *proc* is the name of a Bourne shell script. This flag may be used in conjunction with the `-n` flag, which prevents execution of subsequent commands. `-n` serves as a *breakpoint*, allowing you to stop a script at a convenient point in the debugging, instead of having the whole script execute. Note that saying `set -n` at a terminal will render the terminal useless until an end-of-file is typed.

The command

```
set -x
```

produces an execution trace. Following parameter substitution, each command is displayed as it is executed. The `-v` and `-x` flags are similar; `-x` puts a + sign in front of the line shown being executed and it only displays executing lines, not control lines. This means that a `for` or `while` loop line will be displayed with `-v` but not with `-x`. The following shows the difference:

```
$ cat test
echo hello
for i in one two
do echo $i
done
$  sh -v test
echo hello
hello
for i in one two
do echo $i
done
one
two
$  sh -x test
+ echo hello
hello
+ echo one
one
+ echo two
two
$
```

Notice how, in the second example, *one* and *two* are substituted in for `$i`. Both flags may be turned off by saying

```
set -
```

and the current setting of the Bourne shell flags is available as `$-`.

Keyword Parameters in the Bourne Shell

Bourne shell variables may be given values by assignment or when a shell script is invoked. An argument to a Bourne shell script of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the script begins. The value of *name* in the invoking shell is not affected. For example,

```
$ user=fred command
```

executes *command* with `user` set to `fred`. The `-k` flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as arguments `$1, $2, ....`

You can also use the *set* command to set arguments from within a script. For example,

```
set - *
```

sets `$1` to the first filename in the current directory, `$2` to the next, and so on. Note that the first argument (-) ensures correct treatment when the first filename begins with a `-`.

Parameter Transmission in the Bourne shell

When a Bourne shell script is called, both arguments and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a Bourne shell script by specifying in advance that such parameters are to be *exported*. For example,

```
export user box
```

marks the variables *user* and *box* for export to scripts. When a shell script is called, copies are made of all exported variables for use within the invoked script. For example:

```
$ name=fred
$ export name
$ cat test
echo $name
$ sh test
fred
$
```

Modification of such variables within the script does not affect the values in the calling shell. (It is generally true of a Bourne shell script that it may not modify the state of its caller without explicit request on the part of the caller. Shared file descriptors are an exception to this rule.)

Names whose values are intended to remain constant may be declared *readonly*. The form of this command is the same as that of the `export` command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution in the Bourne Shell

If a Bourne shell parameter is not set, the null string is substituted for it. For example, if the variable d is not set

```
$ echo $d
```

or

```
$ echo ${d}
```

will echo nothing. A default string may be given as in

```
$ echo ${d-.}
```

which will echo the value of the variable d if it is set and '.' otherwise. The default string is evaluated using the usual quoting conventions so that

```
$ echo ${d-'*'}
```

will echo * if the variable d is not set. Similarly

```
$ echo ${d-$1}
```

will echo the value of d if it is set and the value (if any) of $1 otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if d was not previously set then it is now set to the string '.'. The notation ${...=...} is not available for arguments.

```
echo ${d?message}
```

echoes the value of the variable d if it has one; otherwise, the Bourne shell prints *message*, if the shell is interactive, and stops executing the script. If *message* is absent, then a standard message is printed. A Bourne shell script that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
. . .
```

Colon (:) is a command that is built in to the Bourne shell and does nothing once its arguments have been evaluated. If any of the variables user, acct or bin are not set and the shell is not interactive, the shell stops executing the script.

Command Substitution in the Bourne Shell

In a similar way, you can substitute the standard output from a command as the value of a parameter. The command `pwd` displays on its standard output the name of the current directory. For example, if the current directory is `/home/fred/bin` then the command

```
d=`pwd`
```

is equivalent to

```
d=/home/fred/bin
```

The entire string between backquotes. (`` `...` ``) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a `` ` `` must be escaped using a `\` . For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including here documents) and the treatment of the resulting text is the same in both cases. This mechanism allows use of string processing commands within Bourne shell scripts. An example of such a command is `basename`, which removes a specified suffix and the pathname's prefix from a string. For example,

```
basename /home/fred/main.c .c
```

displays the string `main`. The following fragment from a `cc` command illustrates its use:

```
case $A in
    ...
    *.c)    B=`basename $A .c`
    ...
esac
```

that sets `B` to the part of `$A` with the pathname and suffix `.c` stripped.

Here are some composite examples.

- `` for i in `ls -t`; do ... ``
  The variable `i` is set to the names of files in time order, most recent first.
- `` set `date`; echo $6 $2 $3, $4 ``
  will print, for instance, `1977 Nov 1, 23:59:59`

Evaluation and Quoting in the Bourne Shell

The Bourne shell is a macro processor that provides parameter substitution, command substitution, and filename generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in the "Grammar" section. Before a command is executed, the following substitutions occur.

- Parameter substitution, such as `$user`
- Command substitution, such as `` `pwd` ``

  Only one evaluation of a variable occurs. For example, if the value of the variable `y` is *hello*, so that

  ```
  echo $y
  ```

  yields `hello`, and we set the variable `X` to *$y*, then

  ```
  echo $X
  ```

  yields `$y` and not `hello`.
- Blank interpretation

  Following the above substitutions, the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose "blanks" are the characters of the string `$IFS`. By default, this string consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ''
  ```

  will pass on the null string as the first argument to `echo`, whereas

  ```
  echo $null
  ```

  will call `echo` with no arguments if the variable `null` is not set or set to the null string with `null=''`.
- Filename generation

  Each word is then scanned for the file pattern characters `*`, `?`, and `[...]`, and an alphabetical list of filenames is generated to replace the word. Each such filename is a separate argument.

The evaluations just described also occur in the list of words associated with a `for` loop. Only parameter and command substitution occurs in the *word* used for a `case` branch.

As well as the quoting mechanisms described earlier using and `'...'`, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occur, but filename generation and the interpretation of blanks does not. The following characters have special meanings within double quotes and may be quoted using `\`.

Table C-2 *Characters With Special Meaning Between Double Quotes*

Character	*Meaning*
$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

passes the value of the variable x as a single argument to `echo`. Similarly,

```
echo "$*"
```

passes the argument as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation `$@` is the same as `$*` except when it is quoted.

```
echo "$@"
```

passes the arguments, unevaluated, to `echo` and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the Bourne shell metacharacters that are evaluated.

Table C-3 *Quoting Mechanisms*

Quoting Character	*Metacharacter*					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

Where *t*=terminator, *y*=interpreted, and *n*=not interpreted

In cases where more than one evaluation of a string is required, use the built-in command `eval`. For example, if the variable `X` has the value *$y* and `y` has the value *pqr*, then

```
eval echo $X
```

echoes the string *pqr*.

In general, the `eval` command evaluates its arguments (as do all commands) and treats the result as input to the Bourne shell. The input is read and the resulting command(s) are executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, `eval` is required since there is no interpretation of metacharacters, such as |, following substitution.

Error Handling in the Bourne Shell

The treatment of errors detected by the Bourne shell depends on the type of error and on whether the Bourne shell is being used interactively. A Bourne shell invoked with the `-i` flag is deemed to be interactive.

Execution of a command (see also "Command Execution") may fail for any of the following reasons.

- Input/output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault." See Table C-4 for a complete list of SunOS signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the Bourne shell goes on to execute the next command. Except for the last case, the Bourne shell displays an error message. All remaining errors cause the Bourne shell to exit from a command script. An interactive Bourne shell will return to read another command from the terminal. Such errors include the following:

- Syntax errors, such as `if ... then ... done`
- A signal such as an interrupt. The Bourne shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as `cd`.

The Bourne shell flag `-e` terminates the Bourne shell if any error is detected.

Table C-4 *SunOS Signals*

Signal Name	*Signal Number*	*Notes*	*Description*
SIGHUP	1		hangup
SIGINT	2		interrupt
SIGQUIT	3	*	quit
SIGILL	4	*	illegal instruction
SIGTRAP	5	*	trace trap
SIGABRT	6	*	used by abort
SIGEMT	7	*	EMT instruction
SIGFPE	8	*	floating point exception
SIGKILL	9		kill — cannot be caught, blocked, or ignored
SIGBUS	10	*	bus error
SIGSEGV	11	*	segmentation violation
SIGSYS	12	*	bad argument to system call
SIGPIPE	13		write on a pipe with no one to read it
SIGALRM	14		alarm clock
SIGTERM	15		software termination signal from kill
SIGURG	16		urgent condition on IO channel
SIGSTOP	17	†	stop — cannot be caught, blocked, or ignored
SIGTSTP	18	†	stop signal from tty
SIGCONT	19	•	continue after a stop — cannot be blocked
SIGCHLD	20	•	to parent on child stop or exit
SIGTTIN	21	†	background read attempted from control terminal
SIGTTOU	22	†	background write attempted from control terminal
SIGIO	23		input/output possible signal *
SIGXCPU	24		exceeded CPU time limit
SIGXFSZ	25		exceeded file size limit
SIGVTALRM	26		virtual time alarm
SIGPROF	27		profiling time alarm
SIGWINCH	28	•	window changed
SIGLOST	29		resource lost

Notes on the Signals

* These signals normally create a memory image of the terminated process ("core dumped").

• These signals are discarded if the signal action is SIG_DFL.

† These signals normally stop the process.

The Bourne shell itself ignores quit, which is the only external signal that can cause a dump. The signals in this list of potential interest to Bourne shell programs are 1, 2, 3, 14, and 15.

Fault Handling in the Bourne Shell

Bourne shell scripts normally terminate when an interrupt is received from the terminal. The `trap` command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received it executes the commands

```
rm /tmp/ps$$; exit
```

`exit` is another built-in command that terminates execution of a Bourne shell script. If `exit` is not specified, the Bourne shell will resume executing the script at the place where it was interrupted.

SunOS signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without its having to take any further action. If a signal is being ignored, on entry to the Bourne shell script, for example, by invoking it in the background (see "Command Execution"), then `trap` commands (and the signal) are ignored.

The use of `trap` is illustrated by this modified version of the `name` command. You'll recall that the version of the `name` command shown using a here document would only look for one name at a time and that if we modified it to look for multiple names, the here document would be read every time around the `for` loop. Here is a version that copies the here document into a temporary file. The name of the temporary file is derived from the name and process ID of this command. When the script terminates, the `trap` is called to remove the temporary file. Let's take a look at this version of the `name` command (note that script creates a temporary file using `$0` for the command name and `$$` for its PID):

```
#! /bin/sh -u
if [ $# -lt 1 ]; then
    echo Usage: name person ...
    exit 1
fi
junk=/tmp/$0.$$
trap "rm -f $junk; exit" 0 1 2 15
cat > $junk <<woof
Ted Applehead           teda@seeds              7534
Bernice Barns           boat@carib              7441
    . . .
    more names
    . . .
David Smiter            acme@nadir              7435
Ben Tortcake            tort@icky               7258
Dave von Noknock        dave@dove               7296
woof
for person
    do grep -i $person $junk
done
```

The `trap` command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in SunOS, the Bourne shell uses it to indicate the commands to be executed on exit from the Bourne shell script.

A script may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the `nohup` command:

```
trap '' 1 2 3 15
```

which causes both the script and the invoked commands to ignore the *hangup*, *interrupt*, and *kill* signals.

Traps may be reset by saying:

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing:

```
trap
```

The `scan` Script

The `scan` script shown below is an example of the use of `trap` where there is no exit in the `trap` command. `scan` takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end-of-file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when `scan` is waiting for input.

```
d=`pwd`
for i in *
do if test -d $d/$i
    then cd $d/$i
        while echo "$i:"
            trap exit 2
            read x
        do trap : 2; eval $x; done
    fi
done
```

`read` is a built-in command that reads one line from the standard input and places the result in the variable which is its argument. `read` returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

Here is an example of the `scan` command in action:

```
$ scan
bin:
ls
diffmark     henry.pct    lifescreen   scan.sh
bin:
^D
experiments:
ls
Makefile        doctools        macro.packages  test.bs
Old.Stuff       ellipse.ps      macros          test.pages
diffs           junk            new.macros      tmac.ex
experiments:
rm junk
experiments:
^D
misc:
ls -CF
addresses/        memos/            squash/
henry.raving/     quotes/           status.reports/
howto/            ski.cabins/       stoneman/
jokes/            solar/            sugfest/
letters/          sources/          sun.board
misc:
^D
system.v.book:
ls
Makefile           intro.mexp          shell.mexp
book.mss           login.mexp          shex1.mss
docprep.mexp       mail.mexp           shex2.mss
ed.and.sed.mexp    manpage.mss         softtool.mexp
ex.mexp            misc                stdio.mexp
filesystem.mexp    preface.mexp        system.admin.mexp
headex.mss         roman.mss           tablex.mss
system.v.book:
^D
$
```

Command Execution in the Bourne Shell

To run a command (other than a built-in), the Bourne shell first creates a new process using the *fork* system call. The execution environment for the command includes input, output, and the states of signals, and is established in the child process before the command is executed. The built-in command `exec` is used in the rare cases when no fork is required and simply replaces the Bourne shell with a new command. For example, a simple version of the `nohup` command looks like:

```
trap '' 1 2 3 15
exec $*
```

The `trap` turns off the specified signals so that they are ignored by subsequently created commands, and `exec` replaces the shell by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No filename generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

writes its output into a file whose name is `*.c`. Input/output specifications are evaluated left to right as they appear in the command.

A *file descriptor* is a number assigned to a file when the file is opened for reading and/or writing. File descriptors 0, 1, and 2 refer to the *standard input*, *standard output*, and *standard error* (error messages) respectively.

> *word*	The standard output (file descriptor 1) is sent to the file *word*, which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of Bourne shell input that follow, up to but not including a line consisting only of *word*. If *word* is quoted, then no interpretation of the document occurs. If *word* is not quoted, then parameter and command substitution occur, and `\` is used to quote the characters `\` `$` `` ` `` and the first character of *word*. In the latter case, newlines quoted with backslashes are ignored (cf quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.
<&-	The standard input is closed.
>&-	The standard output is closed.

Any of the above may be preceded by a digit, in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background, such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file `/dev/null`. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
$ ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that the command ignores them. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the SunOS convention for a signal is that if it is set to 1 (ignored), then it is never changed, even for a short time. Note that the Bourne shell command `trap` has no effect for an ignored signal.

Calling the Bourne Shell

The Bourne shell interprets the following flags when it is called. If the first character of argument zero is a minus—that is, the command itself starts with a minus—then commands are read from the file `.profile`.

`-c` *string*
If the `-c` flag is present, commands are read from *string*.

`-s` If the `-s` flag is present or if no arguments remain, commands are read from the standard input. Bourne shell output is written to file descriptor 2.

`-i` If the `-i` flag is present or if the Bourne shell input and output are attached to a terminal (as determined by `gtty`), then this Bourne shell is *interactive*. In this case TERMINATE is ignored (so that `kill 0` does not kill an interactive Bourne shell), and INTERRUPT is caught and ignored (so that `wait` is interruptable). In all cases, the shell ignores QUIT.

Bourne Shell Grammar

Commands are parsed initially according to the following grammar.

```
item:       word
      input-output
      name = value

simple-command: item
      simple-command item

command: simple-command
      ( command-list )
      { command-list }
      for name do command-list done
      for name in word... do command-list done
      while command-list do command-list done
      until command-list do command-list done
      case word in case-part... esac
      if command-list then command-list else-part fi

pipeline:       command
      pipeline | command

andor:          pipeline
      andor && pipeline
      andor || pipeline
```

command-list: *andor*
  command-list ;
  command-list &
  command-list ; *andor*
  command-list & *andor*

input-output: > *file*
  < *file*
  >> *word*
  << *word*

file: *word*
  & *digit*
  & -

case-part: *pattern*) *command-list* ;;

pattern: *word*
  pattern | *word*

else-part: `elif` *command-list* `then` *command-list else-part*
  `else` *command-list*
  empty

empty:

word: *a sequence of non-blank characters*

name: *a sequence of letters, digits or underscores starting with a letter*

digit: 0 1 2 3 4 5 6 7 8 9

Bourne Shell Metacharacters and Reserved Words

Syntactic

|	pipe symbol
&&	"andf" symbol
||	"orf" symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

Patterns	`*`	match any character(s) including none
	`?`	match any single character
	`[...]`	match any of the enclosed characters
Substitution	`${...}`	substitute shell variable
	`` `...` ``	substitute command output
Quoting	`\`	quote the next character
	`'...'`	quote the enclosed characters except for '
	`"..."`	quote the enclosed characters except for `$` `` ` `` `\` `"`

Reserved Words

```
if  then  else  elif  fi
case  in  esac
for  while  until  do  done
{  }
read
```

Index

Systems for Open Computing™

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 37-29639

For U.S. Sales Office locations call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
TLX 859017

Australia: (02) 413 2666
Canada: 416 477-6745
France: (1) 40 94 80 00

Germany: (089) 95094-0
Hong Kong: 852 5-8651688
Italy: (39) 6056337
Japan: (03) 221-7021
Korea: 2-7802255
New Zealand: (04) 499 2344
Nordic Countries: +46 (0)8 7647810
PRC: 1-8315568
Singapore: 224 3388
Spain: (1) 2532003
Switzerland: (1) 8289555
The Netherlands: 033 501234

Taiwan: 2-7213257
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 51440

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales

sun
microsystems

Part Number: 800-3833-10
Revision A, of March 27, 1990