SYSTEM V/68
USER'S GUIDE

PRODUCT CODE 72903
41966-00

MOTOROLA
Computer Systems

Product Code 72912
41975G01

SYSTEM V/68 DOCUMENTATION SET

VOLUME I

SYSTEM V/68 USER'S MANUAL, 72905 (41968-00)
- Introduction
- Permuted Index
- Section 1 - Commands

VOLUME II

SYSTEM V/68 USER'S MANUAL, 72905 (41968-00)
- Section 2 - System Calls
- Section 3 - Subroutines
- Section 4 - File Formats
- Section 5 - Miscellaneous Facilities
- Section 6 - Games

VOLUME III

SYSTEM V/68 ADMINISTRATOR'S MANUAL, 72900 (41963-00)
- Introduction
- Permuted Index
- Section 1M - Commands
- Section 7 - Special Files
- Section 8 - Procedures

SYSTEM V/68 ADMINISTRATOR'S GUIDE, 72901 (41964-00)
- Introduction
- Administrative Guidelines
- Using the System
- Accounting
- File System Checking
- LP Spooling System
- System Activity Package

SYSTEM V/68 OPERATOR'S GUIDE, 72904 (41967-00)
- Chapter 1 - Getting Started
- Chapter 2 - System Overview
- Chapter 3 - Using the System
- Appendix A - System Specifications
- Appendix B - Debugging Commands
- Appendix C - Error Messages

SYSTEM V/68 USER'S GUIDE, 72903 (41966-00)
- Introduction
- Primer
- Basics for Beginners
- Text Editors
- An Introduction to Shell
- Source Code Control System (SCCS)
- UNIX-to-UNIX CoPy: A Tutorial

VOLUME IV

SYSTEM V/68 PROGRAMMING GUIDE, 72908 (41971-00)

- Introduction
- An Introduction to Shell
- C Programming Language
- FORTRAN
- Curses and Terminfo Package
- Programming Language EFL

SYSTEM V/68 SUPPORT TOOLS GUIDE, 72909 (41972-000)

- Introduction
- Maintaining Computer Programs (MAKE)
- Augmented Version of MAKE
- The M4 Macro Processor
- The AWK Programming Language
- Desk Calculator Language (BC)
- Desk Calculator Program (DC)
- Lexical Analyzer Generator (LEX)
- Yet Another Compiler-Compiler (YACC)
- Common Object File Format

SYSTEM V/68 ASSEMBLER USER'S GUIDE, 72910 (41973-00)

- Introduction
- Warnings
- General Syntax Rules
- Segments, Location Counters, and Labels
- Types
- Expressions
- Pseudo-Operations
- Span-Dependent Optimization
- Address Mode Syntax
- Machine Instructions

SYSTEM V/68 COMMON LINK EDITOR REFERENCE MANUAL, 72911 (41974-00)

- Introduction
- Using the Link Editor
- Link Editor Command Language
- Notes and Special Procedures
- Error Messages
- Syntax Diagram for Input Directives

VOLUME V

SYSTEM V/68 DOCUMENT PROCESSING GUIDE, 72906 (41969-00)

- Introduction
- Advanced Editing
- Stream Editor
- Nroff and Troff User's Manual
- Table Formatting Program
- Mathematics Typesetting Program
- Memorandum Macros
- Viewgraphs and Slides Macros

SYSTEM V/68 ERROR MESSAGE MANUAL, 72902 (41965-00)

- Introduction
- Error Messages
- Index

SYSTEM V/68
USER'S GUIDE

Product Code 72903

Part Number 41966-00

Version 1

PREFACE

The *SYSTEM V/68 User's Guide,* (Part Number 41966-00, Product Code 72903) describes the capabilities of the SYSTEM V/68 operating system for a new user.

While reasonable efforts have been made to assure the accuracy of this document, Motorola assumes no liability resulting from any omissions in this document or from the use of the information obtained therein. Motorola reserves the right to revise this document and to make changes from time to time in its content without being obligated to notify any person of such revision or changes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 General

Two document types provide information about SYSTEM V/68: manuals and guides. The manuals describe commands, facilities, features, and error messages of the system. The guides provide supplemental details and instructions for system implementation, administration, and use. The manuals are organized as alphabetized entries within tabbed sections. The *SYSTEM V/68 User's Manual* contains sections 1 - 6. The *SYSTEM V/68 Administrator's Manual* contains sections 1M, 7, and 8. Throughout the documentation, references to these manuals are given as *name*(section). For example, *chroot*(1M) is a reference to the *chroot* entry in section 1M of the *SYSTEM V/68 Administrator's Manual.* The following conventions identify arguments, literals, and program names:

- **Boldface** strings are literals and are to be typed as they appear.
- *Italic* strings represent substitutable argument prototypes and program names.
- Square brackets ([]) indicate that an argument is optional.
- Ellipses (...) show that the previous argument prototype may be repeated.

1.2 Contents

The purpose of this guide is to describe and illustrate capabilities of SYSTEM V/68 for a new user. The guide supplements the information provided in the *SYSTEM V/68 User's Manual* by grouping together the commands and facilities needed to accomplish various tasks. The following paragraphs provide brief descriptions of the contents of each major section in the guide. Following these paragraphs, a glossary of terms is provided.

- **Primer.** This section provides basic instructions for accessing the operating system.
- **Basics For Beginners.** This section provides information about creating and using files and directories, preparing documents, and programming. It also includes information about the UNIX-to-UNIX Communications Package (*uucp*(1c)).
- **Text Editors.** This section provides information about the *ed*(1), *ex*(1), and *vi*(1) text editors.
- **An Introduction to Shell.** This section provides information about both basic and advanced features of the shell command programming language. Commands, pipelines and filters, shell procedures, and keyword parameters are described.
- **Remote Job Entry (RJE).** This section is a general introduction to RJE facilities. NOTE: RJE is not supported in the current release of SYSTEM V/68.
- **Source Code Control System (SCCS).** This section provides information about the SCCS facility for documenting and controlling changes in source code files. Terminology, commands, conventions, and files are described.
- **UNIX-to-UNIX CoPy Tutorial.** The UNIX-to-UNIX CoPy (uucp) programs are covered in depth in this tutorial. The first half of the tutorial addresses the novice user and describes, step-by-step, how to send or receive files, mail and commands over the international uucp network. The latter half contains detailed installation, maintenance and debugging information needed by the system administrator. Administrators should not attempt to install uucp without having read the tutorial thoroughly.

1.3 Glossary

The following list defines terms and acronyms used in this volume that may not be familiar to the user.

argument—Words following the command on a command line that provide information necessary to execute a program. Command arguments are often filenames.

ASCII—American Standard Code for Information Interchange.

background—A program execution mode in which the shell does not wait for the command to terminate before prompting for another command.

C language—A general purpose, low level programming language used to write programs (such as numerical, text-processing, and data base) and operating systems.

command—The first word of a command line. It is the name of an executable file that is a compiled program.

command line—A sequence of nonblank arguments separated by blanks or tabs typed in by a user. The first argument usually specifies the name of a command.

command list—A sequence of one or more simple commands separated or ended by a new line or a semicolon.

command procedure—A command procedure is an executable file that is not a compiled program. It is a call to the shell to read and execute commands contained in a file. A sequence of commands may be preserved for repeated use by saving it in a file called a shell procedure, a command file, or a runcom according to preference.

command substitution—When the shell reads a command line, any command or commands enclosed between grave accents (‘...‘) are executed first and the output from these commands replaces the whole expression (‘...‘).

current working directory—The current point of reference for accessing data within the file system.

delta—A set of changes made to a file that is stored by the SCCS.

directory—A type of file that is used to group and organize files and other directories.

EOF—The End-Of-File character is the same as an ASCII EOT character. See EOT.

EOT—The End-Of-Text character is generated by holding down the "CONTROL" key and pressing the lowercase "d" key once. The EOT is used to terminate the shell which usually logs a user off the system.

erase character—The character that is used to delete the previous character on the current line. To turn off the special meaning of the erase character, it must be preceded with a "\". By default, the erase character is #. See **stty**(1) to change the default character.

file—An organized collection of information containing data, programs, or both, which allows

users to store, retrieve, and modify information. A simple filename is a sequence of characters other than a slash (/).

filter—A command that reads its standard input, transforms it in some way, and prints the result as output.

foreground—A program execution mode in which the shell waits for the command to terminate before prompting for another command.

full pathname—The pathname of a specific file starting from the root directory.

group identification number (gid)—A unique number assigned to one or more logins that is used to identify groups of related users.

here documents—A command procedure that has the form **command << eofstring** and causes the shell to read subsequent lines as standard input to the command until a line is composed of only the **eofstring** is read. Any arbitrary string can be used for the **eofstring.**

HOME—Another name for the login directory.

in-line input documents—See here documents.

keyword parameters—An argument to a command procedure that has the form **name=value command arg1 arg2 . . .** here **name** is called the keyword parameter. This allows shell variables to be assigned values when a shell procedure is called. The value of **name** in the invoking shell is not affected, but the value is assigned to **name** before execution of the procedure. The arguments (arg1 arg2 . . .) are available as positional parameters ($1 $2 . . .).

kill character—The character that is used to delete all the characters typed before it on the current line. To turn off the special meaning of the kill character, it must be preceded with a "\". By default, the kill character is @. The default character can be changed via **stty(1)**.

login—A procedure that provides a user access to SYSTEM V/68.

login name—A unique string of letters and numbers used to identify a login.

log off—A procedure that disconnects the user from SYSTEM V/68.

memorandum macros—The general purpose package of text formatting macros used with nroff and troff to produce many common types of documents.

metacharacters—Characters that have a special meaning to the shell, for example: <, >, *, ?, |, &, $, ;, (,), \, ", ', [,].

mode—An absolute mode is an octal number used with **chmod(1)** to change permissions of files.

MRs—Modification Request numbers are recorded within each delta to an SCCS file to identify the reason that prompted the file revision, e.g., a trouble report, change request, trouble ticket etc.

nroff—A text formatting program for driving typewriter-like terminals and printers to produce a screen copy or a hardcopy.

parent directory—The directory immediately above another directory. A ".." is the shorthand name for the parent directory. To make the parent directory of your current working directory your new current directory enter the command: **cd ..**

partial pathname—The pathname between the current working directory and a specific file.

password—A string of up to 13 characters chosen from a 64 character alphabet (., \, 0-9, A-Z, a-z).

pathname—A sequence of directory names separated by the / character and ending with the name of a file. The pathname defines the connection path between some directory and a file.

pipe—A simple way to connect the output of one program to the input of another program, so that each program will run as a sequence of processes.

pipeline—A series of filters separated by the character |. The output of each filter becomes the input of the next filter in the line. The last filter in the line will write to its standard output.

positional parameters—Arguments supplied with a command procedure that are placed into variable names $1, $2, . . . when the command procedure is invoked by the shell. The name of the file being executed is positional parameter $0.

primary prompt—A notification (by default "$ ") to the user that SYSTEM V/68 shell is ready to accept another request.

process—A program that is in some state of execution. The execution of a computer environment including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items.

program—Software that can be executed by a user.

SCCS—The Source Code Control System is a collection of SYSTEM V/68 commands that monitors changes to text files and creates an audit trail for each change.

secondary prompt—A notification (by default "> ") to the user that the command typed in response to the primary prompt is incomplete.

shell—A SYSTEM V/68 user program written in C language that handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed.

shell procedure—See command procedure.

SID—The SCCS IDentification string identifies a particular version of a file and is composed of at most four components separated by periods (release.level.branch.sequence).

standard input—The standard input of a command is sent to an open file that is normally connected to the keyboard. An argument to the shell of the form "< file" opens the specified

file as the standard input thus redirecting input to come from the file named instead of the keyboard.

standard output—Output produced by most commands is sent to an open file that is normally connected to the printer or screen. This output may be redirected by an argument to the shell of the form "> file" which opens the specified file as the standard output.

text editor—An interactive program (*ed*) for creating and modifying text, using commands provided by a user at a terminal.

troff—A text formatting program for driving a phototypesetter to produce high quality printed text.

user-defined variables—A user variable can be defined using an assignment statement of the form **name=value** where **name** must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores up to 512 characters. The **name** is the variable. Positional parameters cannot be included in the name.

user identification number (uid)—A unique number assigned to each login that is used to identify users and the owner of information stored on the system.

variables—A variable is a name representing a string value. Variables that are normally set only on a command line are called parameters (positional parameters and keyword parameters). Other variables are names to which the user (user-defined variables) or the shell may assign string values.

NOTES

2. PRIMER

2.1 Introduction

This section of the *SYSTEM V/68 User's Guide* provides the information users need to access the SYSTEM V/68 operating system. It is not intended to be a detailed description. Many of the subjects described are discussed in detail in other sections of this volume or the *SYSTEM V/68 User's Manual*.

In this primer, software programs that can be executed by users are referred to as "programs". A program that is in some state of execution is referred to as a "process". The request typed by the user is referred to as a "command" or "command line".

In this section, the following graphic conventions are used in examples:

(RETURN)	Indicates that the user should press the RETURN key on the terminal keyboard.
(DEL)	Indicates that the user should press the key marked DEL, DELETE, or RUBOUT (whichever is appropriate for the terminal being used).

2.2 Human Interface

2.2.1 Concept of a Login. The SYSTEM V/68 operating system is accessed by the use of a "login". A login is used by the system to uniquely identify users. Before the user can access the system, a login must be assigned by the system administrator. Every login consists of the following components:

login name

user identification number (uid)

group identification number (gid)

password

A *login name* is a unique string of lowercase letters and/or numbers that identifies an individual to the system. The *login name* must begin with a letter. In many cases, a user's *login name* is his/her real first name, last name, initials, or nickname. Any string of letters and/or digits can be used as the *login name*, as long as it is unique (i.e., different from all other *login names*). Only the first eight characters of a *login name* are used by the system. *Login names* are assigned by the system administrator.

The *uid* of a login is a unique number assigned to each login by the system administrator. This number is used by the system to identify the owners of information stored on the system and the commands that users are executing.

The *gid* is a unique number assigned by the system administrator to each group. This number identifies groups of users that have something in common. For example, all logins used by people in the same department (or working on the same project) may have the same *gid*. The *gid* is important for security and accounting reasons. The impact of *gid* numbers on the user and the group that the user belongs to is described later.

The *password* is a string of 13 characters chosen from a 64-character alphabet (., \, 0-9, A-Z, a-z) that serves to control access to a login. The *password* for a login is the main security feature of the SYSTEM V/68 operating system. Usually, every login is assigned a *password*. When a user logs in to the system, the *password* (if any) assigned to the login being used is requested. Access to the system is not permitted until the correct *password* is entered. The user can change a *password* as needed to ensure that others are not accessing the user's login

and the user's data. Any string can be used as a *password* as long as it is more than five characters in length and is composed of uppercase letters, lowercase letters, numbers, or punctuation. It is recommended that obvious strings such as the user's social security number, birth date, or other data that could be well known about the user not be used as *password*s. If the *password* is something that is well known about the user, someone could gain access to the user's login with little effort. The more unusual the *password*, the more effective the security.

2.2.2 Logging In. In order to log in, the power to the terminal must be turned on and the appropriate switches set. Depending on the type of terminal and communication link, the user may need to press the RETURN or BREAK key a couple of times to synchronize the terminal with the system. When communication is established, the system will prompt with:

login:

The user should type in his/her *login name* followed by a RETURN. After the system digests the *login name*, it will prompt for a *password* with:

Password:

The user should then type his/her *password* followed by a return. The system does not echo the *password* on the terminal screen. This is an extra security measure. If you enter your *login name* and *password* correctly, the system may print one or more "messages of the day". Following the messages, the system will prompt with the primary prompt string, which is usually the **$** symbol. If a mistake is made while logging in or the system adminstrator has not set up the user's login on the system, the following error message is printed:

login incorrect

This error message is followed by the **login:** message. The user should attempt to login again.

The SYSTEM V/68 operating system has a hierarchy of directories. When the system administrator gives the user a *login name*, the administrator also creates a directory for the user. This directory ordinarily has the same name as the user *login name* and is known as the "login" or "home" directory of the user. When the user logs in, the home directory becomes the "current directory" or "working directory" of the user. Any file created under the *login name* is by default in the home directory. In addition, the user may create one or more directories under the home directory. The user may then change to subdirectories by using a "change directory" command. See *cd*(1) for details. Under a directory or a subdirectory, the user may create files as necessary. The user is the owner of the home directory and all subdirectories created under the home directory. As the owner, the user has full permission to create, alter, and remove (destroy) all files and subdirectories of the home directory. To change from one directory to another, the command **cd** is used.

2.2.3 Logging Off. After completing your work, it is best to log off the system. Before logging off, you should have received the prompt string **$** from the system. This means that all your commands have been completed and the system is ready for another command.

The preferred method for logging off is accomplished by typing an American Standard Code for Information Interchange (ASCII) End Of Text (EOT) character which is sometimes called the End-Of-File (EOF). On most terminals, the EOT character is generated by holding down the CONTROL key and pressing the lowercase **d** key once. This is also referred to as a "CONTROL-d". Regardless of the type of terminal, the power to it should be turned off when the terminal is no longer needed. For terminals connected via a phone line, you should depress the talk button and hang up the phone.

2.2.4 Entering Commands. The SYSTEM V/68 operating system "shell" (command interpreter) serves as the interface between the user and the system. The shell accepts requests from the user in the form of a command line and invokes the appropriate program to fulfill the request. The shell prompts the user when it is ready to accept another request. As noted earlier, the prompt of the SYSTEM V/68 operating system shell is the primary prompt string which is by default **$** (a dollar sign followed by a space).

2.2.4.1 Command Line Syntax. Commands or requests to the shell are usually in the form of a single line, that is, a string of one or more words followed by a RETURN. This single line request entered following the prompt is referred to as a "command line". The command line is divided into two major parts—the program name and arguments.

The first word of the command line is the name of the program to be executed. This is referred to as the *command*. All subsequent words are *arguments* to the command. Arguments are used to provide information required by the program.

Spaces and tabs serve as the delimiters for words on the command line. That is, all characters on the command line up to the first space or tab are interpreted as the command. All characters between the first space (or tab) and the second space (or tab) make up the first argument. Thus, the syntax for the command line is:

command argument argument argument ... ...(RETURN)

When spaces or tabs are needed within a single argument, that argument is enclosed by double quote marks ("). For example, to execute a program that requires two arguments such as **john 1** and **doe**, the first argument should be **john 1**. The second argument should be **doe**. The required command line in this case would be:

command **"john 1" doe**(RETURN)

2.2.4.2 Correction and Deletion. All users are likely to make mistakes, especially when typing. The SYSTEM V/68 operating system provides two features to correct command lines. These features are only effective for the current line (i.e., they must be used before the line is ended with a return).

The first correction feature is the erase character (by default, **#**), and the second correction feature is the kill character (by default, @). The erase character erases the character preceding it. For example, a command line entered as

caf#t the fik#le(RETURN)

actually is **cat the file.** The first **#** erases the first **f** and the second **#** erases the **k.** The erase character can be used to erase a series of characters such as in

this####the cat had kittens(RETURN)

which results in **the cat had kittens.** The entire word **this** is erased by the series of **#** characters following it. The first **#** erases the **s,** the second **#** erases the **i,** the third **#** erases the **h,** and the fourth **#** erases the **t.** If you had miscounted the number of erase characters you needed, as in

this ###the cat had kittens(RETURN)

the result would have been **ththe cat had kittens.** The three erase characters would have erased the space, the **s,** and the **i** preceding them.

If you need to enter a # in the command line for some reason, preceding the # with the backslash character (\) will turn off the "erase last character" meaning of the #. For example, a command line entered as

thsi##is is the \#7#7 cat(RETURN)

is actually **this is the #7 cat.**

The second correction feature is the kill character. The kill character deletes the entire current line. For example, the user enters the command line

command#####omma#####mmad argm##gmu##ment

when the user was trying to enter **command argument.** This command line is so full of mistakes and corrections it is hard to determine if it is right. It would be best to delete the entire line and start over. The user can delete the line by ending it with an @ instead of a return. For example in this sequence

kat###catteh##he file######## the flie##e@
cat the file(RETURN)

the first line is deleted by the @ character. It is much easier to delete it and reenter it (as in the second line of the example).

If the @ character is needed in a line, the backslash character (\) should precede it. For example, entering the line

The kill character is a \@.(RETURN)

results in **The kill character is a @.**

2.2.4.3 Erratic Terminal Behavior. Sometimes your terminal may appear to be acting strangely. For example, each letter may be typed twice (terminal may be in the half-duplex mode) or the RETURN may not cause a line feed or a return to the left margin. You can often change this by logging out and logging back in. If logging back in fails to correct the problem, check the following areas:

keyboard	Keys such as caps lock, local, block, etc. should not be in depressed position.
dataphone	For terminals connected via phone lines, the baud rate could be incorrect.
switches	The rear panel of your terminal normally has several switches used to control terminal operations. These switches should be set to be compatible with the SYSTEM V/68 operating system.

If all else fails, the description of the *stty*(1) command can be read to determine the appropriate action to take. To get intelligent treatment of tab characters (which are much used in the SYSTEM V/68 operating system) if your terminal does not have tabs, type the command

stty −tabs

and the system will convert each tab into sufficient blanks to space to the next 8-character field. If your terminal does have hardware tabs, the command **tabs** will set the stops correctly for you (see *tabs*(1)).

2.2.4.4 Read-ahead. The SYSTEM V/68 operating system has full read-ahead capability, which means that the user can type whenever necessary and as fast as desired, even when another command is already outputting on the terminal. If typing is done during output, the input characters appear intermixed with the output characters, but they are stored away and interpreted in the correct order. The user can type several commands one after another without waiting for the first to finish or even begin.

2.2.5 Stopping a Program. Most programs can be stopped by pressing the DEL key (perhaps called DELETE or RUBOUT on your terminal). The INTERRUPT or BREAK key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone with the talk button depressed will also stop most programs.

2.2.6 Mail. After logging in, the user may sometimes get the following message:

You have mail.

The SYSTEM V/68 operating system provides a postal system so you can communicate with other users of the system. To read your mail, type the following command:

mail

Your mail will be printed, one message at a time, most recent message first. After each message, *mail*(1) waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (it will still be there the next time you read your mailbox). If you want to save a mail message in a file, type:

s *filename*

Other responses are described in *mail*(1) in the *SYSTEM V/68 User's Manual.*

How is mail sent to someone else? Assume that **jones** is someone's login name which is recognized by *login*(1). The easiest way to send mail to **jones** is as follows:

mail jones
the text of the letter
on as many lines as you like...
CONTROL-d

As shown previously, the character CONTROL-d is produced by holding down CONTROL and typing a letter **d.**

The CONTROL-d sequence, often called End-Of-File (EOF), is used throughout the system to mark the end of input from a terminal.

For practice, send mail to yourself. (This is not as strange as it might sound—mail to oneself is a handy reminder mechanism.) There are other ways to send mail—you can send a previously prepared letter and you can mail a message to a number of people all at once. For more details, see *mail*(1).

2.2.7 Writing to Other Users. Occasionally, your terminal may display a message similar to

Message from jones tty07...

that is accompanied by a startling beep on terminals that have the capability to beep. It means that Jones (**jones**) wants to talk to you; unless you take explicit action, however, you will not

be able to talk back. To respond, type the following command:

write jones

This establishes a 2-way communication path. Now whatever Jones types on his terminal will appear on yours and vice versa. However, if you are in the middle of some program, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor—read the "Text Editor" section of this document. If you want to prevent other users from writing to your terminal, enter the following:

mesg n

The "n" or "no" tells the system that other users do not have permission to write to your terminal.

A protocol is needed to keep what you type from becoming garbled with what Jones types. Typically, a sequence like the following is used:

Jones types **write smith** and waits.

Smith types **write jones** and waits.

Jones now types a message (as many lines as necessary).
When ready for a reply, Jones signals it by typing
(o)
which stands for "over".

Now Smith types a reply, also terminated by
(o).

This cycle repeats until Smith or Jones wants to end
the conversation. The intent to quit is signalled with
(oo)
for "over and out".

To terminate the conversation, each side must
type a CONTROL-d character alone at the beginning
of a line. (DELETE also works.)
When Jones types CONTROL-d,
the message
EOF
will appear on Smith's terminal.

If you write to someone who is not logged in or who does not want to be disturbed, a message stating this will appear on your terminal. If the target is logged in but does not answer after a reasonable interval, type CONTROL-d.

2.2.8 Online Manual. The *SYSTEM V/68 User's Manual* is kept online. If you are confused and cannot find an expert to assist you, you can print on your terminal a manual section that might help. This is also useful for getting the most up-to-date information on a command. To read a manual section, type **man** *command-name*. Thus, to read up on the *who*(1) command, type:

man who

To read about the *man*(1) command, type

man man

NOTES

3. BASICS FOR BEGINNERS

3.1 Day-to-Day Use

3.1.1 Creating Files—The Editor. The SYSTEM V/68 text editors organize and save typed information. This information could be intended for a 1-page letter or a 1500-line program. (Refer to the "Text Editors" section of this guide for a detailed description.)

All SYSTEM V/68 text editors operate on a "file," a collection of information stored by the operating system. Within SYSTEM V/68 are three different text editors (*ed, ex* and *vi*), each of which provides a user with a different range of operations and capabilities. The following paragraphs are intended as an overview to describe how files are created and edited. The concepts of creating and saving information are true for all three editors; the specific examples that follow correspond to the SYSTEM V/68 text editor *ed*, the lowest level editor of the three.

To create a file called **junk** using *ed*(1), type

ed junk (invokes the text editor)
a (command to add text)
text ...
text ...
. (command to leave append mode)

The dot (.) that signals the end of adding text must be at the beginning of a line by itself. No other *ed* commands will be recognized until the dot is entered; everything typed will be treated as text to be added. No system prompt appears while appending, inserting, or changing text in the text editor.

After a file has been created, various editing operations may be performed on the text that was typed in, such as correcting spelling mistakes, rearranging paragraphs, etc. When editing is completed, the information is written into a file and permanently saved with the editor command:

w

Ed will respond with the number of characters written into the file **junk**.

Until the **w** command is used, there is no permanent record of the information. If, while editing a file, a user is logged off before writing the information into a file, all the editing changes made since the last **w** command are lost. (For each text editor, special back-up recovery procedures are available. In *ed*, the data may be saved in a file called **ed.hup** that can be retrieved at the next editing session.) Once text is written to a file, it can be retrieved any time by typing

ed junk

Type a **q** command to quit the editor. (If you try to quit before performing a **w** command to write the file, the text editor will print a "?" as a reminder. A second **q** will quit the text editor regardless.) Now create a second file called **temp** in the same manner. There should now be two files, **junk** and **temp**.

Refer to the "Text Editors" section in this guide for instructions on creating and writing (saving) files using the *ex*(1) and *vi*(1) editors.

3.1.2 Filenames. Filenames **junk** and **temp** have been used without defining a legal filename. The following rules are valid for all SYSTEM V/68 text editors.

Filenames are limited to 14 characters, which is enough to be descriptive. Although any character can be used in a filename, avoid characters that could have other meanings. In SYSTEM V/68, several characters are assigned a special meaning when they are read by the program that interprets commands. These characters include \ **$** - **!** and * (backslash, dollar sign, minus sign, exclamation mark and star). To avoid problems, use only letters, numbers and dot until you are familiar with the workings of the system.

Naming conventions should follow an internal logic. Suppose you are typing a large document, a book, for example. Logically, the book divides into many small pieces, chapters and perhaps sections. Physically, it must be divided because *ed* will not handle files over 90,000 characters. Thus, the document should be typed as a number of files. One possible method is to create a separate file for each chapter as follows:

chap1
chap2
...

Another method is breaking each chapter into several files as follows:

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

Users can see quickly where a particular file fits into the whole.

There are other advantages to a systematic naming convention. To print the whole book, a user could enter the following:

pr chap1.1 chap1.2 chap1.3 ...

Using the *pr*(1) command (print) in this way is tiring and will often lead to a typing mistake. If files are named logically, there is a shortcut. The user can enter:

pr chap*

The * here has the special meaning of "anything at all", so this translates into "print all files whose names begin with **chap** listed in alphabetical order". This shorthand notation is not a property of the *pr* command. It is system-wide, part of the capability of the program that interprets commands. The program, called the "shell" *sh*(1), is described in detail in the section "Introduction to the Shell" later in this guide.

The * is not the only pattern-matching feature available. To print only chapters 1 through 4 and 9, use the following command:

pr chap[12349]*

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated as follows:

pr chap[1-49]*

Letters can also be used within brackets. The **[a-z]** pattern-matching feature matches any character in the range **a** through **z**.

The **?** pattern matches any single character, so

pr ?

prints all files which have single-character names, and

pr chap?.1

prints the first file of each chapter *chap1.1*, *chap2.1*, etc.

So far, the rules of naming a file have not addressed the problem of uniqueness. Different users may create files with the same 14-character (or less) filename because SYSTEM V/68 identifies files by their location within the file system as well as by name. This is explained in detail in the next section, "Directories."

3.1.3 Directories. SYSTEM V/68 contains ordinary files, described above, and directories. An ordinary file has a filename by which it can be retrieved or referred. The information stored in a file can be displayed, or edited; the file itself can be copied or moved to a new location within the file system. (Copying and moving files are explained in the sections that follow.)

A directory is a file that includes information about other files and possibly, other directories. In an organizational sense, files are located within directories; a directory "contains" files. Directories can also "contain" other directories. A directory within another directory is a subdirectory. There is no limit to the number of subdirectories or files that can be built into the SYSTEM V/68 file system.

When a user is assigned a login name to log in to SYSTEM V/68, that user is also assigned a personal directory, usually with the same name. When you log in, you are "located in" your personal directory, also called your "home" directory. Enter the command

pwd

which is a request to print the working directory. Although the details will vary according to the system you are on, the *pwd* command will print something similar to

/usr/*yourname*

This message indicates that you are currently in the directory *yourname*, which is itself located in the directory **usr**. The **usr** directory is located in the root directory at the base of the file system. By convention, the root directory is written and called / (slash).

Your home directory is your storage area. Within your home directory, you can create files, make subdirectories, copy files, and remove or rename files or directories.

To illustrate the advantages of this organizational system, suppose that you are rewriting a cookbook using one of the text editors. You could divide each file into a chapter section (**chap1.1**, **chap1.2**, and so on) because of the advantages this provides when you are ready to print the files. However, when you first start rewriting, you may not know what information will appear in chapter 1, and what will be put in later chapters. In SYSTEM V/68, the file structure can be used as an organizational aid in itself.

Start in your home directory and create a subdirectory called **cookbook**. When creating a new file (or directory), the new file will be located within the user's current working directory unless special action is taken. That is, if you are located in your home directory, then any new file or directory you create will also be located in your home directory. If you are located in another user's home directory, then the file you create will also be located in the other user's directory; it doesn't matter that you are the person who created the file. Therefore, to create a subdirectory called **cookbook** in your home directory, log in to your directory and enter

mkdir cookbook

using the *mkdir*(1) command (make directory). To start, you may want to create a file that will contain an introduction for your book, for example **intro.** To keep all the files of the book together, the file **intro** should be located within the subdirectory **cookbook**. Currently, you are in your home directory. To check this, perform the command **pwd**. To reposition yourself in your subdirectory, use the command

cd cookbook

The *cd*(1) program (change directory) will move you from one directory into another. (The *cd* program is described in more detail in the paragraphs that follow.) Now you are located in the subdirectory **cookbook**. You can now create a file named **intro** for your cookbook introduction.

Next, consider a second level of subdirectories. For example, you can create five subdirectories within **cookbook**:

poultry
meats
vegetables
desserts
salads

Each of these subdirectories can be entered to add files or to add more subdirectories. At any time, you can perform the command **pwd** to remind yourself where you are located. At some point, you could perform a **pwd** and receive a message as follows:

/usr/*yourname*/cookbook/poultry/chicken/baked

This message says you are located in the subdirectory **baked** within the subdirectory **chicken** within the subdirectory **poultry** within the subdirectory **cookbook**. You can create files in all these directories by making sure you are located in the desired directory when you create the file. When you are finished writing your book, you can rename your files to take advantage of SYSTEM V/68 printing shortcuts.

3.1.4 File System Structure. In general, all files in the system are organized into a tree-like structure with each user's files located several branches into the tree. Imagine three users: Decker, Waitz and Benoit. Each user has a home directory and each home directory is a subdirectory of the directory **usr**, which in turn, is a subdirectory of root **/**. Imagine now that all three users are logged in SYSTEM V/68 and each creates a file named **temp** in their home directories. How does the system tell these files apart?

The filename that a user gives to a file is only part of that file's identity. Every file has a complete *pathname* that locates the file's position within the complete file system tree. A file's pathname represents the full name of the path taken from the root directory to arrive at the location of that file. In the example above, the full pathnames for each of the three files named **temp** would be

/usr/decker/temp
/usr/waitz/temp
/usr/benoit/temp

respectively. It is a universal rule in SYSTEM V/68 that anywhere an ordinary filename can be used the pathname can also be used.

It is possible for a user to move around the file system tree and find any file in the system by starting at the root of the tree and following the path of directories named. Conversely, a

user can start at any location and by performing the commmand **pwd**, retrace the steps back toward the root.

The list program, *ls*(1), will produce a list of all files and subdirectories located within a particular directory. For example, if you now type

ls /usr/*yourname*

the results should be a list of the filenames located within your home directory. With no arguments,

ls

lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

Next, try using the following command:

ls /usr

This should print a long series of names, among which is your own login name *yourname*. On many systems, **usr** is a directory that contains the directories of all the normal users of the system.

The next step is to try the following:

ls /

The response should be something like this (although again the details may be different):

bin
dev
etc
lib
tmp
usr

It may be that you are working in your home directory but you want to relocate to someone else's directory. For example, you may want to make a change in a file owned by someone else. To move around the file system from one directory to another, use the *cd*(1) command (change directory).

The *cd* command is used with either a full pathname or a relative pathname. The full pathname of a directory (or file) is the complete path you would follow to arrive at that directory starting from the root directory. The relative pathname of a directory (or file) is the path you would follow to arrive at that directory, starting from your current working directory. To illustrate, recall the three users Decker, Waitz and Benoit. Decker is working in her home directory but needs to read one of Waitz's files named **june84** located in a subdirectory named **intervals**. Decker has a choice; she can change directories by using the full pathname or the relative pathname. To move to the subdirectory using the full pathname from the root directory, Decker would enter

cd /usr/waitz/intervals

Once inside the directory, Decker can retrieve the file **june84**. Alternatively, to move to Waitz's subdirectory from her own home directory, Decker can move up the tree from **decker** to the directory **usr**, and then down the tree into the directory **waitz** and into **intervals**. SYSTEM V/68 abbreviates the move "up one level" to the symbol .. (dot dot) which is an abbreviated way of writing "the parent of the current directory." Similarly, an abbreviated way of writing "the current directory" is the symbol . (dot). Therefore, to move to Waitz's

subdirectory using a relative pathname, Decker would enter

cd ../waitz/intervals

The relative pathname is a much faster method of moving around the file system when many subdirectories are involved. Compare the two methods when changing from subdirectory **baked** to subdirectory **fried** in the cookbook example described in the previous section. To move from **baked** to **fried** using the full pathname, a user would type

cd /usr/*yourname*/cookbook/poultry/chicken/fried

To move to the same directory using a relative pathway is simply

cd ../fried

If a user enters

cd

by itself, the user is always relocated to the home directory.

If a file owner does not want someone else to have access to the owner's files, privacy can be arranged. Each file and directory has read, write and execute permissions for the owner, a group, and everyone else, which can be set to control access. The *ls*(1) command (list) provides information about each file, including the status of these permissions. For example, the **ls** command and its long list option

ls -l intervals

entered from within the directory **waitz** might produce

-rw-rw-rw- 1 gw bsk 41 Jul 22 02:56 june84
-rw-rw-rw- 1 gw bsk 78 Jul 22 12:57 may84

In the example above, the date and time are the date and time of the last change to the file. The **41** and **78** are the number of characters in each file (which should agree with the numbers received from *ed*). The **gw** identifies the owner of the file, i.e., the person who created it. The **bsk** identifies the group associated with **gw**. The **-rw—rw—rw—** states who has permission to read, write, or execute the file. The first character in **-rw—rw—rw—** is a **-** which indicates this is a file of data. A **d** as the first character would indicate a directory. The remaining nine characters are divided into three sets of permissions. Each set consists of three characters. The three sets correspond to the permissions of the owner, group, and all other users. In this case the owner, group, and others all have permission to read (**r**) and write (**w**) the files. Note that there is no permission for anyone to execute (**x**) the files. (Refer to *ls*(1) and *chmod*(1) for details.)

At some point, a user may want to remove directories that are no longer needed. As a precaution, a directory cannot be removed until it is completely empty. For example, if Waitz was ready to remove her subdirectory

/usr/waitz/intervals

the command would be either

rm intervals/*
rmdir intervals or
rm -r intervals

The **rm intervals/*** command removes all files in the **intervals** directory (because * is interpreted as "match any string"). The **rmdir intervals** command is then used to remove the empty directory. The **intervals** directory must be empty before the command will work.

The **rm -r intervals** command recursively deletes the entire contents of the directory and then removes the directory itself.

3.1.5 Printing Files. There are several ways to print a file. The *ed* editor can be used to print a file as follows:

```
ed junk
1,$p
```

Ed will reply with the count of the characters in the **junk** file and then print all the lines in the file. The **1,$** is an address followed by a command, **p**, to print. The entire line translates to "print all lines from 1 through $," where $ has the special meaning "the last line."

The user can also be selective about the parts of a file to be printed as follows:

```
ed junk
20,35p
```

which prints only lines 20 through 35. There are times when it is not feasible to use the editor for printing. For example, there is a limit on how big a file *ed* can handle (several thousand lines). Secondly, *ed* will only print one file at a time; often, several files must be printed, one after the other.

The simplest printing program is *cat*(1). The **cat** command simply prints on the terminal the contents of all the files named in the order listed. For example:

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two files. The files are concatenated and printed on the terminal.

A second printing program is *pr*(1). The **pr** command produces formatted printouts of files. As with *cat*, *pr* prints all the files named in a list. In addition, *pr* produces headings with date, time, page number, and filename at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print the **junk** file neatly, then skip to the top of a new page and print the **temp** file neatly.

The **pr** command can also produce multicolumn output. Entering

```
pr −3 junk
```

prints **junk** in 3-column format. Any reasonable number can be used in place of "3".

The *pr* program is not a formatting program that can change type fonts or and justifying margins. The true formatters are *nroff*(1) and *troff*(1), and are discussed in the paragraphs on document preparation later in this guide. (Refer to *pr*(1) for more information about other *pr* capabilities.)

Finally, there are also programs that print files on a hard copy printer. See *lp*(1) for more information.

3.1.6 Moving and Copying Files. Commands for moving and manipulating files are helpful for organizing information or sharing information when more than one user is working on a project. For example, a user can use the *move*(1) program to copy a file from one user's directory to another. Perhaps you want to copy the file **junk** now located in your home

directory. Use the *cp*(1) command (copy) as follows:

cp junk morejunk

The contents of **junk** are now duplicated in another file in your directory, **morejunk**. To check this, type

ls

for a list of the filenames in your directory.

It may be that you want to copy a file from another user's directory into your own. To do this, move to the directory where you want the file to be located. Then type the copy command, following the pattern "copy from there to here." For example, to get a copy of the file **june84** from Waitz's directory, you would type

cd

to get to your home directory. Then type

cp /usr/waitz/intervals/june84 waitz_june84

The new file in your directory will be named **waitz_june84**. The copy function usually requires the full pathname when you are copying from another directory.

To move a file from one location to another is the same as giving the file a new filename. This is because SYSTEM V/68 identifies a file by its location within the file system. The major difference between the *cp* program and the *mv*(1) program (move) is that *mv* will destroy the contents of the original file. To see how the **mv** command works, type

mv junk precious

This will rename the **junk** file in your home directory so that it is now named **precious**. Typing **ls** would now produce

morejunk
precious
temp
cookbook

and any other files you might have created. The file **junk** is gone. Move commands should be used with caution. If a file is moved into another file that already exists, the original contents of the existing file are written over and destroyed. In the example above, had **precious** contained any information, the contents of precious would have been lost. If you want to move a file from one directory to another, the procedure is the same as for *cp*.

When you are finished creating and moving files, the files can be removed from the file system by the *rm*(1) program. The **rm** command is used as follows:

rm precious temp

This will remove both files **precious** and **temp**.

The user will get a warning message if one of the named files is not located in the directory, but otherwise *rm* will not send an acknowledgement.

3.1.7 Using Files for I/O Instead of the Terminal. Most of the commands used so far produce output on the terminal. Other commands, may take input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

ls

makes a list of files on your terminal. But if you enter

ls >filelist

a list of your files will be placed in the file **filelist** (which will be created if it does not already exist or overwritten if it does). The symbol > means "put the output on the following file rather than on the terminal". Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

cat f1 f2 f3 >temp

Another symbol, which operates very much like >, is ≫. The ≫ means "add to the end of". That is,

cat f1 f2 f3 ≫temp

means to concatenate **f1**, **f2**, and **f3** to the end of whatever is already in **temp** instead of overwriting the existing contents. As with >, if **temp** does not exist, it will be created.

In a similar way, the symbol < means to take the input for a program from the following file instead of from the terminal. Thus, a script of commonly used editing commands can be put into a file called **script**. **Script** could then be run on a file by entering:

ed *file* **<script**

Another example is using **ed** to prepare a letter in file **let**. The letter (file **let**) could be sent to several people as follows:

mail waitz decker benoit <let

3.1.8 Pipes. A major innovation in the UNIX operating system is the idea of a "pipe". A pipe is a way to connect the output of one program to the input of another program, so the two run as a sequence of processes—a pipeline.

For example,

pr f g h

will print the files **f**, **g**, and **h**, beginning each on a new page. Instead of printing the files separately, the files can be printed together as follows:

```
cat f g h >temp
pr <temp
rm temp
```

This method is more work than necessary. To take the output of **cat** and connect it to the input of **pr**, use the following pipe:

cat f g h | pr

The vertical bar | means to take the output from **cat** which would normally have gone to the terminal and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

ls | pr −3

prints a list of your files in three columns. The program *wc*(1) counts the number of lines, words, and characters in its input; the *who*(1) command prints a list of users currently logged

on the system, one per access port. Thus

who | wc −l

tells how many people are logged on. The command

ls | wc −l

counts the files in the current working directory.

Most programs that read from the terminal can read from a pipe as well. Most programs that write on the terminal can write on a pipe as well. There can be an unlimited number of commands in a pipeline.

Many operating system programs are written to take input from one or more files if file arguments are given. If no arguments are given, the programs will read from the terminal, and thus can be used in pipelines. One example using the *pr*(1) command to print files **a**, **b**, and **c** in three columns and in the order specified is as follows:

pr −3 a b c

But in

cat a b c | pr −3

the **pr** prints the information coming down the pipeline, still in three columns.

3.1.9 The Shell. The "shell" mentioned previously is actually the *sh*(1) program. The shell is the program that interprets what is typed as commands and arguments. The shell also translates the special meanings of characters such as * into lists of filenames, and translates <, >, and | into changes of input and output streams.

The user can run two programs with one command line by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into two commands. Thus

date; who

does both commands before returning with a prompt character.

More than one program can run simultaneously; this is called running programs in the background. The background mode enables the shell to prompt for another command without waiting for the previous command to finish. An example of processing in the background is:

ed file <script&

The ampersand (&) at the end of a command line means "start this command running, then take further commands from the terminal immediately", that is, don't wait for it to complete. Thus the script will begin, but the user can do something else at the same time. To keep the output from interfering with the terminal, enter

ed file <script >script.out&

which saves the output lines in a file called **script.out.**

When a command is initiated with **&**, the system replies with a number called the process number. Programs running simultaneously can be terminated as follows:

kill *process_number*

The process number is used to identify the command to be stopped. If you forget the process number, the *ps*(1) command will list the process number for all programs you are running. (Entering **kill 0** will kill all your processes.) Entering **ps −a** will provide information about

all *active* programs that other users are currently running.

To start three commands that will execute in the order specified and in the background, enter the following:

command_1; command_2; command_3&

A background pipeline can be started as follows:

command_1 | command_2 &

The shell can read a file to get commands. For example, suppose a user wants to perform a sequence of actions after every login such as:

- Set the tabs on the terminal
- Find out the date
- Find out who's on the system.

The three necessary commands to perform these actions, *tabs*(1), *date*(1), and *who*(1), could be put in a file called **startup**. The **startup** file would then be run as follows:

sh startup

This instruction commands the machine to run the shell with the file **startup** as input. The effect is the same as typing the contents of **startup** on the terminal.

If this is to be a regular activity, the need to type **sh** every time can be eliminated by typing the following command only once:

chmod +x startup

To run the sequence of commands thereafter, you only need to enter:

startup

The *chmod*(1) command marks the file as being executable. The shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile** and place in it the line "startup". Upon logging in, the shell gains control and executes the commands found in the **.profile** file. (For further information about the **.profile** file, refer to the "Introduction to Shell" section of this guide.)

3.2 Document Preparation

UNIX operating systems are used extensively for document preparation. There are two major formatting programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, etc. The *nroff*(1) (pronounced "en-roff") program is designed to produce output on terminals and line-printers. The *troff*(1) (pronounced "tee-roff") program is designed to drive a phototypesetter, which produces very high quality output on photographic paper. This document was formatted with *troff*.

3.2.1 Formatting Packages. The basic idea of *nroff*(1) and *troff*(1) is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there may be commands that specify how long lines are, whether to use single or double spacing, and the running titles to use on each page.

Because the detailed commands of *nroff* and *troff* are cumbersome to use effectively, several "packages" of canned formatting requests are available to let you specify elements such as paragraphs, running titles, footnotes, and multicolumn output, with little effort and without having to learn all of *nroff* and *troff*. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

This section provides a brief description of the "memorandum macros" package known as *mm*(1). Formatting requests typically consist of a period and two uppercase letters, such as

.TL

which is used to introduce a title or

.P

to begin a new paragraph.

The text of a typical document is entered so it looks something like this:

.TL
title
.AU *author information*
.MT *memorandum type*
.P
text ...
text ...
.P
More text ...
text ...
.SG *signature*

The lines that begin with a period are the formatting macro requests. For example, **.P** calls for starting a new paragraph. The precise meaning of **.P** depends on the output device being used (typesetter or terminal, for instance) and the publication in which the document will appear. For example, the *mm*(1) macros normally assume that a paragraph is preceded by a space (one line in *nroff* and one-half line in *troff*) and the first word is indented. These rules can be changed if desired, but they are changed by changing the interpretation of **.P** not by retyping the document.

To produce a document in standard format using *mm*(1), a user would type the command

troff −mm *files ...*

for the typesetter and

nroff −mm *files ...*

for a terminal. The **−mm** argument tells *troff* and *nroff* to use the memorandum macro package of formatting requests. There are several similar packages; check with a local expert to determine which ones are in common use on your machine. *Nroff*, *troff*, and the macro packages are documented in detail in the *SYSTEM V/68 User's Manual* and the *SYSTEM V/68 Document Processing Guide.*

3.2.2 Supporting Tools. In addition to the basic formatters, there are many supporting programs that help with document preparation. The list in the next few paragraphs is far from complete. Refer to the *SYSTEM V/68 User's Manual* and *SYSTEM V/68 Document Processing Guide* for a full listing of available support programs.

The *eqn*(1) and *neqn* programs let you integrate mathematics into the text of a document in an easy-to-learn language that closely resembles your speaking style. For example, the *eqn* input

```
lim from {n-> inf} x sub n =0
```

produces the output

$$\lim_{n \to \infty} x_n = 0$$

The program *tbl(1)* provides an analogous service for preparing tables. The *tbl* program does all the computations necessary to align complicated columns with elements of varying widths.

The *spell*(1) program detects possible spelling mistakes in a document. The *spell* program compares the words in your document to a dictionary (stored in memory) and prints those words that are not in the dictionary. It knows enough about English spelling to detect plurals and the like.

The *grep*(1) program looks through a set of files for lines that contain a particular text pattern (similar to the editor's context search on a single file). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters **ing** in the files **chap***. The "**$**" signifies that the pattern searched for must appear at the end of the line to produce a match. A "^" could have been used to indicate that the pattern to search for must occur at the beginning of a line. The *grep* program is often used to locate the misspelled words detected by the *spell* program.

The *diff*(1) program prints a list of the differences between two files, so that two versions can be compared automatically. This is a vast improvement over proofreading by hand.

The *wc*(1) program counts the words, lines, and characters in a set of files. The *tr*(1) program translates characters into other characters. For example, *tr* will convert uppercase characters to lowercase characters and vice versa. The following command translates uppercase letters into lowercase letters:

```
tr [A-Z] [a-z] <input >output
```

The *sort*(1) program sorts files in a variety of ways while *cxref*(1) makes cross-references. The *ptx*(1) program makes a permuted index (keyword-in-context listing). The *sed*(1) program provides many of the editing facilities of the text editors but can apply them to arbitrarily long inputs. The *awk*(1) program provides the ability to do both pattern matching and numeric computations and to conveniently process fields within lines. These programs are for advanced users and are not limited to document preparation.

Most of these programs are independently documented in both the *SYSTEM V/68 Document Processing Guide*, and the *UNIX System User's Manual*.

3.2.3 Hints for Preparing Documents. Most documents go through several versions before they are finished. Following a few guidelines will make the job of revising documents much easier.

Start each sentence on a new line. Make lines short and break lines at natural places, such as after commas and semicolons. Because most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions simplify any editing needed later.

Keep document files relatively small, perhaps 10,000 to 15,000 characters. Larger files edit more slowly. If a mistake is made, it is better to clobber a small file than a big one. Split the files at natural boundaries in the document for the same reasons that you start each sentence on a new line.

Another suggestion is not to commit to the formatting details too early. One of the advantages of using formatting packages is that the package permits many format decisions to be delayed until the last possible moment.

As a rule of thumb, almost all documents should be produced using a set of requests or commands (macros). The macros used should be defined either by using one of the existing macro packages (the easiest way) or by defining your own *nroff* and/or *troff* macros. As long as the text is entered systematically, it can easily be cleaned up and formatted through a combination of editing commands and macro definitions.

3.2.4 Programming. No attempt will be made here to teach any of the programming languages available, but a few words of advice are in order. One of the reasons why the UNIX operating system is a productive programming environment is that there is already a rich set of tools available. Facilities like pipes, input/output redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing a program completely from scratch.

3.2.5 Shell Programming. The pipe mechanism lets you build complicated operations with spare parts that already exist. For example, the first draft of the *spell* program was (roughly)

cat ...	collect the files
| tr ...	put each word on a new line
| tr ...	delete punctuation, etc.
| sort	into dictionary order
| uniq	discard duplicates
| comm	print words in text but not in dictionary

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be used to do things that would normally require special programs on other systems. For example, to list the first and last lines of each file in a set of files, such as a book, you could laboriously type:

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

The same job can be performed much more easily. One procedure is to type:

```
ls chap* >temp
```

to get the list of filenames into a file called **temp**. The **temp** file can then be edited using global commands combined with special characters as follows:

```
1,$ s/^.*$/e &\
1p\
$p/
```

The results are written into the **script** file (using the command **1,$ w script**) and then the following command is entered:

```
ed < script
```

Beginners need not understand how the shell reads the special character meanings, but instead appreciate that this shortcut eliminates the need for repetitive typing.

Users can also build shell loops to repeat a set of commands over and over again for a set of arguments, as illustrated below:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable **i** to each filename in turn, then does the command. The shell loop and command can be entered at the terminal or put into a file for later execution.

An option often overlooked by new users is that the shell is itself a programming language, with variables, control flow **if-else, while, for, case** subroutines, and interrupt handling. Since there are many building-block programs, writing new programs can sometimes be avoided by piecing together some of the building blocks with shell command files. Examples and rules can be found in the "Introduction To Shell" section of this guide.

3.2.6 Programming in C. The C language is a reasonable choice for programming a substantial task. Everything in the SYSTEM V/68 operating system is based on the C language. The system itself is written in C, as are most of the programs that run on the system. The C language is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how to do input/output and similar functions.

Most input and output in C is best handled with the standard input/output library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, limit the system interactions in a program to the facilities provided by this library. (Refer to Section 3 of the *SYSTEM V/68 User's Manual*.)

The C programs that do not depend too much on the special features of the UNIX operating system (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the PDP-11, it currently includes Data General NOVA and ECLIPSE, Harris /7, Honeywell 6000, HP 2100, IBM 370, Intel 8086, Interdata 8/32, Motorola 68000, VAX-11/780, Western Electric 3B20 and 3B5, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of programs that support C. The *lint*(1) program checks C programs for potential portability problems and detects errors such as mismatched argument types and uninitialized variables.

For larger programs whose source is on more than one file, the *make*(1) program allows users to specify the dependencies among the source files and the processing steps needed to make a

new version. The program then checks the times that the files were last changed and does the minimal amount of recompiling to create a consistent updated version.

The *sdb*(1) program is useful for debugging C programs. Yet, the most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited statistical service, so a user can find where programs spend their time executing. Compile the programs with the **−p** option; after the test run, use the *prof*(1) command to print a program execution profile. The command *time*(1) will give the gross run-time statistics of a program, but the times are not very accurate or reproducible.

3.2.7 Other Languages. If FORTRAN must be used, there are two possibilities—FORTRAN 77 (*f77*(1)) and RATFOR (*ratfor*(1)). *Ratfor* provides the control structures and free-form input that characterize C, yet permits the writing of code that is also portable to other environments. SYSTEM V/68 FORTRAN tends to produce large and relatively slow-running programs. Furthermore, support software like *prof*(1), is virtually useless with FORTRAN programs. If there is a FORTRAN 77 compiler on your system, it may be a viable alternative to *ratfor* and it has the advantage that it is compatible with the C language and related programs. (The *ratfor* processor and C tools can be used with FORTRAN 77 too.)

If your application requires translating a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, the *yacc*(1) compiler-compiler is recommended for developing a compiler quickly. The *lex*(1) lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself or as a front end processor to recognize inputs for a *yacc*-based program. Both *yacc* and *lex* require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later.

Ratfor, *yacc*, and *lex* are documented in the *SYSTEM V/68 User's Manual* and *SYSTEM V/68 Support Tools Guide*.

4. TEXT EDITORS

4.1 SYSTEM V/68 Editors

SYSTEM V/68 provides three interactive programs for creating and modifying text files: *ed*(1), *ex*(1), and *vi*(1). When any of the editors are invoked to edit a file, the file is copied into a buffer. All changes directed by the user at a terminal are made to the buffer copy and later are written to the original file. *Ed*(1) is a line editor; i.e., the user must specify the text on which an operation is to be performed. *Ex*(1) has many additional features which make it easier to use and more efficient than *ed*(1). These improvements include additional operations, more error messages, recovery from a system crash, and the ability to make use of advanced terminal types. *Vi*(1) is a visual editor; text is displayed on the terminal screen and the user moves the cursor to the place where a change is to be made. In addition to display editing, *vi*(1) has access to all the *ex*(1) commands. Similarly, *ex*(1) has a visual mode which is the same as using the *vi*(1) editor. Both *vi*(1) and *ex*(1) have an open mode. This is the same as visual mode, except that only one line of text is displayed. Dumb terminals or hard copy terminals use open mode. Deciding which editor or mode to use depends on several factors. *Vi*(1) and the visual mode of *ex*(1) require definitions of the terminal being used. As stated previously, *ex*(1) has many features that are not present in *ed*(1). Because of the use of the edit buffer, there is a limit on the size of the file that can be edited with *ed*(1), *ex*(1), or *vi*(1). SYSTEM V/68 has a stream editor, *sed*(1) that can be used for large files. This is a non-interactive text editor that applies a command or set of commands to an entire file. A description of *sed* is provided in the *SYSTEM V/68 Support Tools Guide*.

4.2 The ed Text Editor

4.2.1 General. *Ed* is a basic text editor which is available on all UNIX systems. This section is a tutorial introduction and guide for new users of *ed*. Only the most useful and frequently used facilities are discussed. These include: printing, appending, changing, deleting, moving, and inserting entire lines of text; reading and writing files; context searching and line addressing; substituting; global changing; and using some special characters for easier editing.

This tutorial is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using the editor to follow the examples. Read the description in Section 1 of the *SYSTEM V/68 User's Manual* while experimenting with *ed*. The exercises illustrate techniques not completely discussed in the actual text. A summary at the end of section 4.2 lists the *ed* commands and their functions.

It is assumed that the user knows how to log on to the operating system and has a basic understanding of what an operating system file is. For more information about the SYSTEM V/68 operating system facilities, refer to the section, "Basics For Beginners". The user must know what character to type as the end-of-line character on the user's particular terminal. This character is the RETURN or newline character (key) on most terminals. Hereafter, the end-of-line character, whatever it is, will be referred to as RETURN.

4.2.2 Editing Commands.

4.2.2.1 Getting Started. Assume that the user has logged in to a SYSTEM V/68 operating system and it has just printed the *prompt character*, usually a

```
$
```

The easiest way to invoke *ed* is to type:

ed (followed by a RETURN)

Now the *ed* program has been invoked and is waiting to be told what to do.

4.2.2.2 Creating Text. Suppose some text is to be created starting from scratch. Perhaps the very first draft of a document or paper is to be entered. Normally, it will be entered in rough form and undergo modifications (editing) later. This part will describe how to enter some text to get a file of text started. How to make changes and corrections to the text is described later.

When *ed* is first invoked, it is like working with a blank piece of paper (the file)—there is no text or information present on the paper (in the file). The text must be supplied by the person using *ed*. This is usually done by typing in the text or by reading it into *ed* from a file. We will start by typing in some text and return shortly to how to read files.

First we will discuss a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if desired, or simply as the information that is to be edited. In effect, the buffer is like a piece of paper on which we will write things, change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to the text by typing instructions called "commands." Most commands consist of a single lowercase letter. Each command is typed on a separate line. (Sometimes the command is preceded by information about the line or lines of text to be affected—these will be described below.) The *ed* text editor makes no response to most commands—there is no prompting or response message like "ready".

The first command is "append," written as the letter

a

on a command line all by itself. It means "append (or add) text lines to the buffer as I type them in." To enter lines of text into the buffer, type an

a

followed by a RETURN and the lines of text desired:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

To stop appending, type a line that contains only a period. The . is used to tell *ed* that the appending is finished. (If *ed* seems to be ignoring your entries, type an extra line with just the . on it. You may find you have added some garbage lines to your text which will have to be deleted later.)

After the append command has been used, the buffer will contain the following three lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The **a** and the . are not there because they are not text.

To add more text to what already exits, just issue another **a** command and continue typing.

To practice this, enter **ed** and create some text using the append command **a**

a
...text...
.

Note that no system prompt appears while in the text editor. Do not forget to write the text into memory with the write command **w**. Then leave *ed* with the **q** command and print the file to see that everything worked. To print a file, enter

pr *filename*
or
cat *filename*

in response to the prompt character (**$**). Try both.

4.2.2.3 Error Messages (?). If at any time the user makes an error in the commands typed into *ed*, the text editor will tell the user by typing the following:

?

This is about as cryptic as it can be, but with practice the user can usually figure out the goof. The user can get a brief explanation of the error by typing

h

The **help** command gives a short error message that explains the reason for the most recent **?** diagnostic.

4.2.2.4 Writing Text Files-The Write Command. Usually, you will want to save your text for later use. To write out the contents of the buffer onto a file, use the write command

w

followed by the filename to write on. This will copy the buffer's contents onto the specified file, destroying any previous information on the file. To save (write) the text in a file named **junk**, for example, type:

w junk

Leave a space between **w** and the filename. The *ed* program will respond by printing the number of characters it wrote out. In this case, *ed* would respond with:

68

Remember that blanks and the return character at the end of each line are included in the character count. Writing a file just makes a copy of the text—the buffer's contents are not disturbed, so the user can go on adding lines to it. This is an important point. At all times the *ed* program works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. (Writing out the text onto a file from time to time as it is being created is a good idea. If the system crashes or if the user makes some horrible mistake, all the text in the buffer will be lost but any text that was written onto a file is relatively safe.)

4.2.2.5 Leaving ed-The Quit Command. To terminate a session with *ed*, first save your text by writing it onto a file using the **w** (write) command, and then type the quit command:

q

The system will respond with the prompt character:

```
$
```

At this point your buffer vanishes, with all its text, which is why the user would want to write before quitting. Actually *ed* will print the character

```
?
```

if the user tries to quit without writing. At this point, the user writes if desired; if not, another **q** will get you out regardless and will not save the text in the buffer.

4.2.2.6 Reading And Editing Text Files. A common way to get text into the buffer is to read it from another file in the file system. This is what you do to edit text that you saved with the **w** command in a previous session. The edit command

```
e
```

retrieves the entire contents of a file into the buffer. So if the user had saved the three lines "Now is the time", etc., with a **w** command in an earlier session, the edit command

```
e junk
```

would place the entire contents of the file **junk** into the buffer and respond with a number

```
68
```

which is the number of characters in the **junk** file. **If anything was already in the buffer, it is deleted first.**

If the **e** command is used to read a file into the buffer, then the user does not need to use a file name after a subsequent **w** command; *ed* remembers the last filename used in an **e** command, and **w** will write on this file. Thus a good practice to follow is:

```
ed
e filename
[editing session]
.
w
q
```

This way, the user can simply enter **w** from time to time and be secure in the knowledge that if the user got the filename right at the beginning, the user is writing into the proper file each time. Note that after each edit command (**e**) or each write command (**w**) the number of characters is returned by *ed*.

The user can find out at any time what filename *ed* is remembering by typing the file command **f**. In this example, if you typed

```
f
```

ed would reply

```
junk
```

Sometimes you want to read a file into the buffer without destroying information that is already in the buffer. This is done using the read command **r**. The command

```
r junk
```

will read the file **junk** into the buffer. The command appends the file specified to the end of whatever file is already in the buffer. So if you do a read after an edit command such as

e junk
r junk

the buffer will contain *two* copies of the orginal text as follows:

Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.

Like the **w** and **e** commands, **r** prints the number of characters read in after the reading operation is complete. Generally speaking, **r** is much less used than **e.**

The read command **r** may also be used to read a file external to the buffer into the file in the buffer. While in *ed* and at the current line, enter the command

.r *filename*

and *filename* will be read into the file (already in the buffer) immediately after the current line. None of the file in the buffer is destroyed, rather the external file *filename* has been read into and been combined with the file already in the buffer. The file that was read remains in *filename* also. You only copied it. The significant difference between **r** and **.r** is the final destination of the file. The **r** command appends the file to whatever is already in the buffer; the **.r** command reads the file into the buffer immediately after the current line.

Experiment with the **e** command—try reading and printing various files. You may get an error *?name* where *name* is the name of a file. This means that the file does not exist. Some typical causes of getting an empty file are spelling the filename wrong or perhaps trying to read or write a particular file which your permissions will not allow. Try alternately reading and appending to see that they work similarly. Verify that

ed filename

is exactly equivalent to

ed
e filename

What does

f filename

do?

4.2.2.7 Printing Buffer Contents. To print or list the contents of the buffer (or parts of it) on the terminal, use the print command **p.** This is done as follows. Specify the line numbers where printing is to begin and end. These numbers have a comma between the beginning number and the ending number:

beginning line number, ending line number **p**

Thus to print the first ten lines of the contents of any buffer (i.e., lines 1 through 10), type:

1,10p (prints lines 1 through 10)

The *ed* will respond by printing the specified starting line (1) through the specified ending line (10).

Suppose it is desirable to print *all* the lines in the buffer. You could use **"1,30p"** as above if it is known there are exactly 30 lines in the buffer. But in general, the *ed* program provides a shorthand symbol for "line number of the last line in the buffer," the dollar sign **$**. To print all the lines in the buffer, use it this way:

1,$p (Prints all lines in buffer)
or
,p (Prints all lines in buffer also)

This will print all the lines in the buffer (line 1 through the last line). The **1,$p** can be abbreviated **$,p** . To stop the printing before the last line is printed, push the DEL key or the DELETE (or equivalent) key on the terminal. The *ed* program will respond

?

and wait for the next input command.

To print the *last* line of the buffer, you could use

$,$p

but *ed* lets you abbreviate this to

$p

Any *single* line can be printed by typing the line number followed by a **p**. Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further. You can print any single line by typing just the line number—no need to type the letter **p**. If you enter

$

ed will print the last line of the buffer. Entering a single line number will print that line only.

It is also possible to use **$** in combinations like

$-5,$p

which prints the last five lines of the buffer. This helps to determine the end of the contents of the buffer when more is to be entered.

Create some text using the **a** command and experiment with the **p** command. You will find, for example, that line 0 or a line beyond the end (last line) of the buffer can not be printed. Attempts to print a buffer in reverse order by entering

3,1p

will not work.

Suppose the buffer contains the six lines of text

Now is the time
for all good men
to come to the aid of their party
Now is the time
for all good men
to come to the aid of their party

and the following was entered

1,3p

and *ed* has printed the three lines. Try typing just

p (no line numbers)

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact, it is the last (most recent) line that was processed. (It was the line just printed.) The **p** command can be repeated without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line processed so that it can be used instead of an explicit line number. The most recent line is referred to by the shorthand symbol

. (Pronounced "dot")

Dot is a line number in the same way that **$** is. Dot means exactly "the current line", or loosely, "the line that was processed most recently." The dot can be used in several ways—one possibility is to enter:

.,$p

This will print all the lines, including the current line, to the last line of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, while others do not. The print command **p** sets dot to the number of the last line printed; the last command entered (**.,$p**) will set both **.** and **$** to the last line in the buffer (line 6).

Dot is most useful when used in combinations, for example

.+1 (or equivalently, **.+1p**)

This means "print the next line" and is a handy way to step slowly through a buffer. You can also enter

.−1 (or **.−1p**)

which means "print the line before the current line". This enables stepping through the buffer backwards if desired. Another useful combination is

.−3,.−1p

which prints the previous three lines.

All of these combinations change the value of dot. The user can learn the current value of dot by typing

.= (dot line number is ?)

The *ed* program will respond by printing the value (line number) of dot.

Let us summarize some things about the **p** command and dot. Essentially, **p** can be preceded by 0, 1, or 2 line numbers (for our example). If no line number is given, it prints the "current line", the line to which the dot refers. If one line number is given with or without the letter **p**, it prints that line and sets dot there. If two line numbers are separated by a comma, it prints all the lines in that range from the first number to the last number, and sets dot to the last line printed. If two line numbers are specified, the first can not be bigger than the second.

Typing a single RETURN will cause printing of the next line; RETURN is equivalent to

.+1p

Try it. Typing a ^ is equivalent to typing the minus −. It can be used in multiples, as ^^^, which will move the current line or dot line backwards three lines from the current line. The − or the ^ can be considered equivalent to −1p since either moves the dot back one line.

4.2.2.8 Deleting Lines. Suppose three extra lines in the buffer are not needed. They may be removed by use of the delete command:

d

Except that **d** deletes lines instead of printing them, its action is similar to that of the print command **p**. The lines to be deleted are specified for **d** exactly as they are for **p** as follows:

starting line, ending line **d**

Thus the command

4,$d

deletes lines 4 through the end. There are now three lines left, that can be checked by using:

1,$p

And notice that **$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**. The delete command **d** and the print command **p** may be used together thus

dp

which deletes the current line, prints the following line, and sets dot to the line printed.

Experiment with **a**, **e**, **r**, **w**, **p**, and **d** until you become familiar with their use. While experimenting, also use **.**, **$**, and line numbers to understand their use.

When you start to feel adventurous, try using line numbers with **a**, **r**, and **w** as well. You will find that **a** will append lines *after* the line number that you specify (rather than after dot); **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and **w** will write out exactly the lines specified, not necessarily the whole buffer. These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering:

0r *filename*

Lines can be entered at the beginning of the buffer by using:

0a
...text...
.

Notice that "**.w**" is *very* different from

.
w

4.2.2.9 The Substitute Command. One of the most used of all commands is the substitute command:

s

This is the command that is used to change individual words or letters within a line or group of lines. The substitute command is used for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

Now is th time

notice the *e* has been left off. The **s** command can be used to fix this as follows:

1s/th/the/

This says: in line 1, substitute the characters **the** for the characters **th.** Since *ed* will not print the result automatically, enter

p

to verify that the substitution worked, and you should get

Now is the time

which is what is desired. Notice that dot must have been set to the line where the substitution took place since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line **s/***change this/to this/*

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between *starting—line* and *ending—line*. Only the first occurrence on each line is changed however. If every occurrence is to be changed, see "Exercise 5". The rules for line numbers are the same as those for the print command **p** except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is not changed. This causes an error response **?** as a warning.)

Thus the following can be entered

1,$s/speling/spelling/

to correct the first spelling mistake (**speling** in this case) on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/*something /something else /***p**

which makes some correction on the current line and then prints it (current line) to make sure it worked out right. If it did not, you can try again. Notice that there is a **p** on the same

line as the **s** command. With few exceptions, **p** can follow any substitute command.

It is also legal to say

s/....//

which means change the first string of characters (....) to *nothing*, i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if the buffer contained

Nowxx is the time

this can be corrected by entering

s/xx//p

to get

Now is the time

Notice that // (two adjacent slashes) means "no characters" *not* a blank.

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, enter

a
the other side of the coin
.
s/the/on the/p

which results in the following:

on the other side of the coin

A substitute command changes only the first occurrence of the first string. All occurrences can be changed by adding a **g** (for "global") command to the **s** command, like this:

s/.../.../gp

Try other characters instead of slashes to delimit the two sets of characters in the **s** command—anything should work except blanks or tabs.

The characters

^ . $ [* \ &

have special meanings in a substitute command that are discussed in detail later in this section.

4.2.2.10 The Search Command. When the substitute command is mastered, you may move on to another highly important feature of *ed*—context searching.

Suppose the original three lines of text in the buffer are as follows:

Now is the time
for all good men
to come to the aid of their party.

Suppose the word **their** is to be changed to **the.** How is the line that contains **their** located? With only three lines in the buffer, it is easy to keep track of what line the word **their** is on. But when the buffer contains several hundred lines, users need a method of specifying the desired line, regardless of what its number is, by specifying some context (unique text) on it.

The way to say "search for a line that contains this particular string of characters" or "unique text" is to type:

/string of characters to find/

For example, the *ed* expression

/their/

is a context search which is sufficient to find the desired line—it will locate the next occurrence of the characters between slashes (**"their"**). It also sets dot to that line and prints that line for verification:

"Next occurrence" means that *ed* starts looking for the string at line ".+1" and searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search "wraps around" from **$** to **1**. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can not be found in any line, *ed* types the error message

?

Otherwise, it prints the line it found.

The search for the desired line and the substitution can be done together, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, and print the line.

The expression **"/their/"** is a context search expression. In the simplest form, all context search expressions are like this—a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

/Now/+1
/good/
/party/−1

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, enter

/Now/+1s/good/bad/
or
/good/s/good/bad/
or
/party/−1s/good/bad/

The choice is dictated only by convenience. All three lines could be printed by entering

/Now/,/party/p
or
/Now/,/Now/+2p

or by any number of similar combinations. The first one of these might be better if you do not know how many lines are involved. (Of course, if there were only three lines in the buffer, a convenient method of printing would be

1,$p

but not if there were several hundred.)

The basic rule is: a context search expression is the same as a line number, so it can be used wherever a line number is needed.

Experiment with context searching. Try a body of text with several occurrences of the same string of characters and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. They can also be used with **r**, **w**, and **a**.

Try context searching using **"?text?"** instead of **"/text/"**. This scans lines in the buffer in reverse order rather than normal (forward) order. This is sometimes useful if you go too far while looking for some string of characters—it is an easy way to back up.

The characters

^ . $ [* \ &

have special meanings in a context search that are discussed in detail later in this section.

The *ed* program provides a short method for repeating a context search for the same string. For example, the *ed* line number

/string/

will find the next occurrence of "*string*". It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely:

//

This short method stands for "the most recently (last) used context search expression". It can also be used as the first string of the substitute command, as in

*/string1/***s**//*string2/*

which will find the next occurrence of *string1* and replace it by *string2*. This can save a lot of typing. Similarly

??

means "scan backwards for the same expression."

4.2.2.11 Changing and Inserting Text. This section discusses the change command

c

which is used to change the current line or to replace the current line with a group of one or more lines, and the insert command

i

which is used for inserting a group of one or more lines immediately before the current line.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change the first line (.+1) through the last line ($) of a file to something else, type

.+1,$c
...type the lines of text you want here...
.

The lines typed between the **c** command and the '.' (dot) command will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors.

If only one line is specified in the **c** command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of '.' (dot) to end the input—this works just like the '.' (dot) in the **a** command and must appear by itself at the beginning of a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append. For example,

*/string/***i**
...type the lines to be inserted here...
.

will insert the given text before the next line that contains "*string*". The text between **i** and the '.' (dot) is inserted *before* the specified line. If no line number is specified, the dot line is used. Dot is set to the last line inserted.

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

starting-line,ending-line **d**
i
...text...
.

is almost the same as

starting-line,ending-line **c**
...text...
.

These are not precisely the same if the last line ($) gets deleted. Check this out. What is dot?

Experiment with the append command **a** and the insert command **i** to see that they are similar but not the same. You will observe that

line-number **a**
...text...
.

appends *after* the given line, while

line-number **i**
...text...
.

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, **a** appends after line dot, and **c** changes line dot.

4.2.2.12 Moving Text-The Move Command. The move command **m** is used for cutting and pasting—it allows a group of lines to be moved from one place to another in the buffer. Suppose the first three lines of the buffer are to be placed at the end of the buffer instead of at the beginning. This could be performed by entering:

1,3w temp
$r temp
1,3d

This method will work, but it is a lot easier using the **m** command as follows:

1,3m$

The general case is:

starting-line,ending-line **m** *after this line*

Notice that there is a third line to be specified—the line after which the other lines are to be moved. Of course, the lines to be moved can be specified by context searches; if you had

First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.

the two paragraphs could be reversed like this:

/Second/,/end of second/m/First/−1

Notice the "−1" which means that the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

4.2.3 The Global Commands. The two global commands are **g** and **v**. The global command **g** is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

g/peling/p

prints all lines that contain "**peling**". More usefully,

g/peling/s//pelling/gp

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

1,$s/peling/pelling/gp

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** if "**peling**" is not found, whereas the **s** command will.

There may be several commands used in conjunction with the **g** command, but every line except the last must end with a backslash "\". For example:

```
g/xxx/-1s/abc/def/\
.+2s/ghi/jkl/\
.-2,.p
```

makes changes in the lines before and after each line that contains "**xxx**", then prints all three lines.

The **v** command is the same as **g** except that the commands are executed on every line that does *not* match the string following **v**. The following input

```
v/ /d
```

deletes every line that does not contain a blank.

4.2.4 Special Characters. You may have noticed that command work differently when some characters like **.**, *****, **$**, are used in context searches and in the **s** command. The reason is that *ed* treats these characters as special, with special meanings. For instance, in a context search or the first string of the substitute command only,

```
/x.y/
```

means "a line with an **x**, *any character*, and a **y**", not just "a line with an **x**, a period, and a **y**."

The following is a complete list of the special characters that have special meanings.

```
^  .  $  [  *  \  &
```

Warning: **The backslash character "\" is special to "ed". For safety's sake, avoid it where possible.**

If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash.

Here is a brief synopsis of the other special characters. First, the circumflex "^" signifies the beginning of a line. Thus

```
/^string/
```

finds "*string*" only if it is at the beginning of a line. It will find

```
string
```

but not

```
the string...
```

The dollar sign "**$**" is just the opposite of the circumflex; it means the end of a line. The input

```
/string$/
```

will only find an occurrence of "*string*" at the end of some line. This implies, of course, that

```
/^string$/
```

will find a line that contains just "*string*" and

```
/^.$/
```

finds a line containing exactly one character.

The character ".", as mentioned above, matches anything. For example, the input

```
/x.y/
```

matches any of the following:

```
x+y
x-y
xy
x.y
```

This is useful in conjunction with "*" which is a repetition character. The "a*" is a shorthand input for "any number of a's" therefore ".*" matches any number of anythings. For example, input

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since ".*" finds the longest possible match, this goes up to the last comma.)

The "[" is used with the "]" to form *character classes*; for example,

```
/[0123456789]/
```

matches any single digit, i.e., any one of the characters inside the braces will cause a match. This can be abbreviated to

```
[0-9]
```

Finally, the "&" is another shorthand character — it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. One tedious method is just to retype the line. Another method is to enter

```
s/^/(/
s/$/)/
```

using your knowledge of "^" and "$". But the easiest way uses the "&" as follows:

```
s/.*/(&)/
```

This says "match the whole line and replace it by itself surrounded by parentheses." The "&" can be used several times in a line; consider using

```
s/.*/&?&!!/
```

to produce

```
Now is the time? Now is the time!!
```

You do not have to match the whole line, of course. If the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

the end of the world is at hand

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string "**/world/**" found the desired line; the shorthand "//" found the same word in the line; and the "**&**" saved you from typing it again.

The "**&**" is a special character only within the replacement text of a substitute command and has no special meaning elsewhere. You can turn off the special meaning of "**&**" by preceding it with a backslash "\". Inputting

s/ampersand/\&/

will convert the word "**ampersand**" into the literal symbol "**&**" in the current (dot) line.

4.2.5 Summary of Commands and Line Numbers. The general form of the *ed* text editor commands is the *command name*, perhaps preceded by one or two line numbers. In the case of the edit command **e**, the read command **r**, and the write command **w**, the *command name* is also followed by a *filename*. Normally, only one command is allowed to be entered per line, but a print command **p** may follow any other command (except for the edit command **e**, the read command **r**, the write command **w**, and the quit command **q**).

a *Append*, adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a dot "." is typed at the beginning (first character) of a new line. Dot is set to the last line appended.

c *Change* the specified lines to the new text which follows. Entering new lines is terminated by a dot "." as with **a**. If no lines are specified, the current line (dot) is replaced. Dot is set to the last line changed.

d *Delete* the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless **$** is specified in which case dot is set to the last line, **$**.

e *Edit* new file. Any previous contents of the buffer are thrown away, so issue a write command **w** beforehand.

f Print the remembered *filename*. If a name follows **f**, the remembered name will be set to it.

g The *global* command **g/—/***commands* will execute the commands on those lines that contain "—".

i *Insert* lines before the specified line or the current line (dot line) until a "." is typed at the beginning of a new line. Dot is set to last line inserted.

m *Move* lines specified to the line named after **m**. Dot is set to the last line moved.

n Print the *number* of the addressed line(s) followed by a tab and the line itself.

p *Print* specified lines. If none are specified, print line dot. A single line number is equivalent to "line number". A single RETURN prints the next line, i.e., the dot-plus-one line, ".+1".

q The *quit* command exits from *ed*. It wipes out all text in the buffer if you give it twice in a row without first giving a write command **w**.

r	*Read* a file into the buffer (at the end unless specified elsewhere). Dot is set to the last line read. If **.r** *filename* is used, the *filename* is read into the buffer immediately after the dot line.
s	The **s**/*string1*/*string2*/ command is used to *substitute* the characters "*string1*" into "*string2*" in the specified lines. If no lines are specified, the substitution is made in line dot. Dot is set to the last line in which a substitution took place; if no substitution took place, dot is not changed. The command **s** changes only the first occurrence of "*string1*" on a line; to change all occurrences on a line, type a **g** after the final slash.
v	The *exclude* command **v**/—/*commands* executes commands only on those lines that do *not* contain "—".
w	The *write* command writes out the buffer contents onto a file. Dot is not changed.
.=	The ".=" causes the printout of the current line number. The *dot value* prints the line number of the current line (dot line). The "=" by itself prints the value of the last line in the file.
!	The "!" is a *temporary escape* command. The line "!*command-line*" causes "*command-line*" to be executed as an operating system command.
/——/	The *context search* command searches for the next line which contains the string of characters "——" and prints it. Dot is set to the line where string was found. Search starts at line ".=1", wraps around from the last line "$" to line "1", and continues to dot (the current line) if necessary.
?——?	Performs *context search* in reverse direction. Starts search at the previous line ".—1", scans to line 1, wraps around to the last line "$", and scans back to the current line (dot line) if necessary.

4.3 The ex Text Editor

4.3.1 Starting the ex Editor. When invoked, *ex* determines the terminal type from the TERM variable in the environment. If there is a TERMCAP variable in the environment and the type of the terminal described matches the TERM variable, then that description is used. If the TERMCAP variable contains a pathname (beginning with a /), the editor will seek the description of the terminal in that file (rather than the default **/etc/termcap**). If there is a variable EXINIT in the environment, the editor will execute the commands in that variable; otherwise, if there is a file **.exrc** in your HOME directory, *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or **.exrc** will be executed before each editor session.

A command to enter *ex* has the following prototype. (Brackets ([]) surround optional parameters.)

ex [**-**][**-v**][**-t** *tag*][**-r**][**-l**][**-w***n*][**-R**][**+***command*] *name...*

a. The most common case edits a single file with no options, i.e.:

ex *filename*

b. The - command line option suppresses all interactive-user feedback and is useful in processing editor scripts in command files.

c. The **-v** option is equivalent to using *vi* rather than *ex*.

d. The **-t** option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition.

e. The **-r** option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files.

f. The **-l** option sets up for editing LISP, setting the *showmatch* and *lisp* options.

g. The **-w** option sets the default window size to *n*, and is useful on dial-ups to start in small windows.

h. The **-R** option sets the *readonly* option at the start. (Not available in all Version 2 editors due to memory constraints.)

i. The *name* arguments indicate files to be edited.

j. An argument of the form *+command* indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form /pat/ or line numbers, e.g., +100 starting at line 100.

4.3.2 File Manipulation.

4.3.2.1 Current File. The *ex* editor is normally used to edit the contents of a single file. The file being edited is considered the current file; its name is recorded as the current filename. The *ex* editor performs all editing actions in a buffer (a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible.

The current file is almost always considered to be edited. This means that the contents of the buffer are logically connected with the current filename, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not edited then *ex* will not normally write on it if it already exists. The *file* command will say "**[Not edited]**" if the current file is not considered edited.

4.3.2.2 Alternate File. Each time a new value is given to the current filename, the previous current filename is saved as the alternate filename. Similarly, if a file is mentioned but does not become the current file, it is saved as the alternate filename.

4.3.2.3 Filename Expansion. Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character **%** in filenames is replaced by the current filename and the character **#** by the alternate filename. This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a **"No write since last change"** diagnostic message is received.

4.3.2.4 Multiple Files and Named Buffers. If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the argument list. The current argument list may be displayed with the **args** command. The next file in the argument list may be edited with the **next** command. The argument list may also be respecified by specifying a list of names to the **next** command. These names are expanded with the resulting list of names becoming the new argument list, and *ex* edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names **a** through **z**. It is also

possible to refer to A through Z; the uppercase buffers are the same as the lowercase buffers but commands **append** text to named buffers rather than replacing the buffer contents if uppercase names are used.

4.3.2.5 Read-Only Mode. It is possible to use *ex* in the read-only mode to look at files that you have no intention of modifying. This mode protects you from accidentally overwriting the file. Read-only mode is on when the *readonly* option is set. It can be turned on with the **-R** command line option, with the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting the *noreadonly* option. It is possible to write, even while in read-only mode, by writing to a different file, or by using the **!** form of write.

4.3.3 Exceptional Conditions.

4.3.3.1 Errors and Interrupts. When errors occur *ex* (optionally) rings the terminal bell and prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints **"Interrupt"** and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

4.3.3.2 Recovering From Hang-ups and Crashes. If a hang-up signal is received and the buffer has been changed since it was last written, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second case) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hang-up or editor crash. To recover a file you can use the **-r** option. If you were editing the file *resume*, then you should change to the directory where you were when the crash occurred, giving the command

ex -r *resume*

After checking that the retrieved file is good, you can write it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

ex -r

will print a list of the files that have been saved for you. In the case of a hang-up, the file will not appear in the list, although it can be recovered.

4.3.4 Editing Modes. The *ex* editor has five distinct modes.

a. The primary mode is the *command* mode. Commands are entered in *command* mode when a **:** prompt is present and are executed each time a complete line is sent.

b. In *text input* mode *ex* gathers input lines and places them in the file. The append, insert, and change commands use *text input* mode. No prompt is printed. This mode is left by typing a "." alone at the beginning of a line and *command* mode resumes.

c. The last three modes are *open* mode, *visual* mode (entered by the commands of the same name), and *text insertion* mode (within *open and visual* modes).

   — The *open* mode allows local editing operations to be performed on the text in the file. The **open** command displays one line at a time and can be used on any terminal.

   — The *visual* mode allows local editing operations to be performed on the text in the file. The **visual** command works on CRT terminals with random positioning cursors, using the screen as a single window for file editing changes.

4.3.5 Command Structure. Most command names are English words (initial prefixes of the words are acceptable abbreviations). The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the **substitute** command can be abbreviated "s". The shortest available abbreviation for the **set** command is "se".

4.3.5.1 Command Parameters. Most commands accept prefix addresses specifying the lines in the file that they are to affect. The forms of these addresses will be discussed below. A number of commands also may take a trailing count specifying the number of lines to be involved in the command. (Counts are rounded down if necessary.) Thus, the **10p** command will print the tenth line in the buffer. The **delete 5** command will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, the information always being given after the command name. Examples are: option names in a **set** command (**set** *number*), a filename in an **edit** command, a regular expression in a **substitute** command, or a target address for a **copy** command (**1,5 copy 25**).

4.3.5.2 Command Variants. A number of commands have two distinct variants. The variant form of the command is invoked by placing an **!** immediately after the command name. Some of the default variants may be controlled by options; in this case the **!** serves to toggle the default.

4.3.5.3 Flags After Commands. The characters **#**, **p** and **l** may be placed after many commands. (A **p** or **l** must be preceded by a blank or tab except in the single special case **dp**.) In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, **p** is rarely necessary. Any number of + or - characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

4.3.5.4 Comments. It is possible to give editor commands that are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of the text (shell escapes and the substitute and map commands).

4.3.5.5 Multiple Commands Per Line. More than one command may be placed on a line by separating each pair of commands by a | character. However, the global commands, comments, and the shell escape **!** must be the last command on a line, as they are not terminated by a |.

4.3.5.6 Reporting Large Changes. Most commands that change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the **report** option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an **undo** command. After commands with more global effect (such as **global** or **visual**), you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

4.3.6 Command Addressing.

4.3.6.1 Addressing Primitives.

The current line. Most commands leave the current line as the last line that they affect. The default address for most commands is the current line; thus, **.** is rarely used alone as an address.

n	The *n*th line in the editor buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+n -n	An offset relative to the current buffer line. The forms **.+3**, **+3**, and **+++** are all equivalent; if the current line is line 100 they all address line 103.
/pat/ ?pat?	Scan forward and backward, respectively, for a line containing *pat*, a regular expression (as defined below). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing *pat*, then the trailing **/** or **?** may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located. (The forms **\/** and **\?** scan using the last regular expression used in a scan; after a substitute, **//** and **??** would scan using the substitute's regular expression.)
" 'x	Before each non-relative motion of the current line **.**, the previous current line is marked with a tag, subsequently referred to as **"**. This makes it easy to refer or return to this previous context. Marks may also be established by the **mark** command, using single lowercase letters **x** and the marked lines referred to as *'x*.

4.3.6.2 Combining Addressing Primitives. Addresses to commands consist of a series of addressing primitives, separated by **,** or **;**. Such address lists are evaluated left-to-right. When addresses are separated by **,** the current line **.** is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses, the default in this case is the current line (**.**); thus, **,100** is equivalent to **.,100**. It is an error to give a prefix address to a command that expects none.

4.3.7 Command Descriptions. The following form is a prototype for all *ex* commands:

address command **!** *parameters count flags*

All command parts are optional. Within *visual* mode, the *ex* editor ignores a **.** preceding any command. In the following command descriptions, the default addresses are shown in parentheses. These parentheses are not part of the command syntax. Confusion may arise from some of the command descriptions that include the ↑ character. We use the ↑ character to represent the key labeled ^ on your terminal. This convention has been adopted to avoid confusing the ^ terminal key and the notation commonly used to mean "control".

abbreviate *word rhs* abbr: **ab**

Add the named abbreviation to the current list. When in *visual* mode, if *word* is input as a complete word, it will be changed to *rhs*.

(.) **append** abbr: **a**
text
.

Reads the input text and places it after the specified line. After the command, **.** addresses the last line input or the specified line if no lines were input. If address **0** is given, text is placed at the beginning of the buffer.

a!
text
.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text*.

args

The members of the argument list are printed, with the current argument delimited by [and].

(.,.) **change** *count* abbr: **c**
text
.

Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a **delete**.

c!
text
.

The variant toggles *autoindent* during the change.

(.,.) **copy** *addr flags* abbr: **co**

A copy of the specified lines is placed after *addr*, which may be **0**. The current line **.** addresses the last line of the copy. The command **t** is a synonym for **copy**.

(.,.) **delete** *buffer count flags* abbr: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, the specified lines are saved in that buffer or appended to it if an uppercase letter is used.

ex *file* abbr: **e**

Used to begin an editing session on a new file. The editor first checks to see if the current buffer has been modified since the last **write** command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file, and prints the new filename. After insuring that this file is not a binary file (such as a directory), a block or character special file (other than **/dev/tty**), a terminal, or a binary or executable file (as indicated by the first word), the editor reads the file into its buffer. If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered edited. If the trailing newline character is missing from the last line of the input file, it will be supplied and a complaint will be issued. This command leaves the current line (.) at the last line read. If executed from within *open* or *visual* mode, the current line is initially the first line of the file.

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +n *file*
Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g., *+/pat*.

file abbr: **f**
Prints:

the current filename

whether it has been "modified" since the last **write** command

whether it is *read-only* mode

the current line

the number of lines in the buffer

the percentage of the way through the buffer of the current line.

In the rare case that the current file is "not edited" this is noted also. In this case you have to use the form **w!** to write to the file, since the editor is not sure that a **write** command will not destroy a file unrelated to the current contents of the buffer.

file *file*
The current filename is changed to *file* which is considered "not edited".

(1,$) **global** */pat/cmds* abbr: **g**
First marks each line among those specified that matches the given regular expression. Then the given command list is executed with . initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a \. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. The **append**, **insert**, and **change** commands and associated input are permitted; the . terminating input may be omitted if it would be on the last line of the command list. The **open** and **visual** commands are permitted in the command list and take input from the terminal.

The **global** command itself may not appear in *cmds*. The **undo** command is also not permitted there, since **undo** instead can be used to reverse the entire **global** command. The options *autoprint* and *autoindent* are inhibited during a **global** command, (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire **global** command. Finally, the context mark (') is set to the value of . before the **global** commands begin and is not changed during a **global** command, except perhaps by an *open* **or** *visual* mode within the **global** command.

g! */pat/cmds* abbr: **v**
The variant form of a **global** command runs *cmds* at each line not matching *pat*.

(.) **insert** abbr: **i**

text

.

Places the given text before the specified line. The current line is left at the last line input. If there were no lines input it is left at the line before the addressed line. This command differs from **append** only in the placement of text.

i!
text

.

The variant toggles *autoindent* during the insert.

(.,.+1) **join** *count flags* abbr: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there is a . at the end of the line, or none if the first following character is a). If there is already white space at the end of the line, the white space at the start of the next line will be discarded.

j!

The variant causes a simpler *join* with no white space processing. Characters in the lines are simply concatenated.

(.) **k** *x*

The **k** command is a synonym for **mark**. It does not require a blank or tab before the following letter.

(.,.) **list** *count flags*

Prints the specified lines in a more unambiguous way. Tabs are printed as ^I and the end of each line is marked with a trailing **$**. The current line is left at the last line printed.

map *lhs rhs*

The **map** command is used to define macros for use in *visual* mode. The *lhs* should be a single character, or the sequence *#n* (for a digit), referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* has been typed. On terminals without function keys, you can type *#n*.

(.) **mark** *x*

Gives the specified line mark *x*, a single lowercase letter. The *x* must be preceded by a blank or a tab. The addressing form **'x** then addresses this line. The current line is not affected by this command.

(.,.) **move** *addr* abbr: **m**

The **move** command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes that may have been made.

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list. The first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(.,.) **number** *count flags* abbr: # or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* abbr: **o**
(.) **open** */pat/flags*

Enters intra-line editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use **Q**. (Not available in all Version 2 editors due to memory constraints.)

preserve

The current editor buffer is saved as though the system has just crashed. This command is for use only in emergencies when a **write** command has resulted in an error and you do not know how to save your work. After a **preserve** you should seek help.

(.,.)**print** *count* abbr: **p** or **P**

Prints the specified lines with non-printing characters printed as control characters ^*x*; delete (octal 177) is represented as ^?. The current line is left at the last line printed.

(.)**put** *buffer* abbr: **pu**

Puts back previously deleted or yanked lines. Normally used with **delete** to effect movement of lines, or with **yank** to effect duplication of lines. If no buffer is specified, then the last deleted or yanked text is restored. (No modifying commands may intervene between the **delete** or **yank** and the **put**, nor may lines be moved between files without using a named buffer.) By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: **q**

Causes the *ex* editor to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last **write** command was issued, and does not **quit** (the *ex* editor will also issue a diagnostic if there are more files in the argument list). Normally, you will wish to save your changes and you should give a **write** command. If you wish to discard them, use the **q!** command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

(.)**read** *file* abbr: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *filename* is given, the current filename is used. The current filename is not changed unless there is none, in which case *file* becomes the current name. The sensibility restrictions of the **edit** command apply here also. If the file buffer is empty and there is no current name, then **ex** treats this as an **edit** command.

Address **0** is legal for this command and causes the file to be read at the beginning of the

buffer. Statistics are given as for the **edit** command when the **read** successfully terminates. After a **read** the current line is the last line read. (Within *open* and *visual* the current line is set to the first line read rather than the last.)

(.)**read** !*command*
Reads the output of the *command* into the buffer after the specified line. Typically, the read specifies a *filename*; here, the read specifies a *command* instead. A blank or tab before the **!** is mandatory.

recover *file*
Recovers *file* from the system save area. Used after an accidental hang-up of the phone or a system crash (The system saves a copy of the file you are editing only if you have made changes to the file.) or **preserve** command, except that when you use **preserve** you will be notified by mail when a file is saved.

rewind abbr: **rew**
The argument list is rewound, and the first file in the list is edited.

rew!
Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*
With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all option values.

Giving an option name followed by a **?** causes the current value of that option to be printed. The **?** is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form "**set** *option*" to turn them on or "**set** *nooption*" to turn them off. String and numeric options are assigned via the form "**set** *option=value*".

More than one parameter may be given to **set**; they are interpreted left-to-right.

shell abbr: **sh**
A new shell is created. When it terminates, editing resumes.

source *file* abbr: **so**
Reads and executes commands from the specified file. The **source** commands may be nested.

(.,.)**substitute** */pat/repl/options count flags* abbr: **s**
On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl*. If the *global* indicator option character **g** appears, then all instances are substituted. If the *confirm* indication character **c** appears, then before each substitution, the line to be substituted is typed with the string to be substituted marked with ↑ characters. By typing a **y**, one can cause the substitution to be performed; any other input causes no change to take place. After a **substitute** command the current line is the last line substituted.

Lines may be split by substituting newline characters into them. The newline in *repl* must be escaped by preceding it with a \. Other metacharacters available in *pat* and *repl* are described below.

(.,.) **substitute** *options count flags* abbr: **s**

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the & command.

(.,.) **t** *addr flags*

The **t** command is a synonym for *copy*.

ta *tag*

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file. If you have modified the current file before giving a *tag* command, you must write it out, giving another *tag* command; specifying no *tag* will reuse the previous tag.

The *tag* file is normally created by a program such as *ctags* and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form that can be used by the editor to find the tag. This field is usually a contextual scan using */pat/* to be immune to minor changes in the file. Such scans are always performed as if *nomagic* was set.

Names in the *tag* file must be sorted alphabetically. (Not available in all Version 2 editors due to memory constraints.)

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

undo abbr: **u**

Reverses the changes made in the buffer by the last buffer editing command.

Note: **global** commands are considered a single command for the purpose of **undo** (as are **open** and **visual** commands). Also, the commands **write** and **edit** which interact with the file system cannot be undone.

Undo is its own inverse. The **undo** command always marks the previous value of the current line (.) as ". After an **undo** command, the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect, such as **global** and **visual**, the current line regains its pre-command value after an **undo**.

unmap *lhs*

The macro expansion associated by **map** for *lhs* is removed.

(1,$) *v/pat/cmds*

A synonym for the **global** command variant **g!**, running the specified *cmds* on each line that does not match *pat*.

version abbr: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

(.) **visual** *type count flags* abbr: **vi**

Enters visual mode at the specified line. The *type* argument is optional and may be -, ↑,

or . as in the **z** command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See sections 4.4.1 through 4.4.10 for more details. To exit *vi* and return to *ex*, type **Q**.

visual file
visual +*n* file
From visual mode, this command is the same as edit

(1,$) **write** *file* abbr: **w**
Write changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file. The current line remains the same. (The editor writes to a file if it is the current file and is edited, or if the file is actually a teletype (**/dev/tty, /dev/null**). If the file does not exist, it is created, then written to. Otherwise, you must give the variant form **w!** to force the write.)

If an error occurs while writing the current and edited file, the editor considers that there has been no write since last change even if the buffer had not previously been modified.

(1,$) **write**>> *file* abbr: **w**>>
Writes the buffer contents at the end of an existing file.

w! *name*
Overrides the checking of the normal **write** command and will write to any file that the system permits.

(1,$) **w** !*command*
Writes the specified lines into *command*.

Note: There is a difference between **w!** which overrides checks and **w !** which writes to a command. An example of the **w !** command is **w !cat**, which enables you to quickly scan a section of text.

wq *name*
A **write** command followed with a **quit** command.

wq! *name*
The variant overrides checking on the sensibility of the **write** command, as **w!** does.

xit *name*
If any changes have been made and not written, writes the buffer out, then quits.

(.,.)**yank** *buffer count* abbr: **ya**
Places the specified lines in the named buffer for later retrieval via **put**. If no buffer name is specified, the lines will be saved only until you execute a **put** or press ESCAPE. If you perform any other edit function before the **put**, the lines will be lost (see the **put** command description).

(.+1) **z** *count*
Print the next *count* lines (default *window*).

(.) z *type count*

Prints a window of text with the specified line at the top. If *type* is - the line is placed at the bottom; a . causes the line to be placed in the center.

Note: Forms z= and z↑ also exist; z= places the current line in the center, surrounds it with lines of - characters, and leaves the current line at this line. The form z↑ prints the window before z- would. The characters +, ↑ and - may be repeated for cumulative effect. On some Version 2 editors, no *type* may be given.

A *count* gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a *count* that is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the ! character is sent to a shell to be executed. Within the text of *command* the characters % and # are expanded as in filenames, and the ! character is replaced with the text of the previous command. Thus, in particular, !! repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "no write" of the buffer contents since the last change to the editing buffer, then as a warning, a diagnostic message will be printed before the command is executed. A single ! is printed when the command completes.

(*addr, addr*) ! *command*

Takes the specified address range and supplies it as standard input to *command*. The resulting output then replaces the input lines.

(.) =

Prints the line number of the addressed line. The current line is unchanged.

(.,.) > *count flags*
(.,.) < *count flags*

Performs intelligent shifting on the specified lines: < shifts left and > shifts right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space characters, blanks and tabs, are shifted. No non-white space characters are discarded in a left-shift. The current line becomes the last line that was changed in the shift.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1,.+1)
(.+1,.+1)|

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

(.,.) & *options count flags*

Repeats the previous substitute command.

(.,.) *options count flags*
Replaces the previous regular expression with the previous replacement pattern from a substitution.

4.3.8 Regular Expressions and Substitute Replacement Patterns.

4.3.8.1 Regular Expressions. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. The *ex* editor remembers two previous regular expressions: the previous regular expression used in a substitute command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression). The previous regular expression can always be referred to by a null RE (// or ??).

4.3.8.2 Magic and Nomagic. The regular expressions allowed by the *ex* editor are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* editor default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character \ to use them as "ordinary" characters. With *nomagic* option set, there are only three metacharacters: \ , in all cases; **$** , at the end of a regular expression; and ↑ at the beginning of a regular expression. The characters and **&** also lose their special meanings to the replacement pattern of a substitute when *nomagic* option is set. However, the power of metacharacters is still available by preceding the (now) ordinary character with a \ .

The remainder of the discussion of regular expressions assumes that the setting of this option is *magic*.

4.3.8.3 Basic Regular Expression Summary. The following basic constructs are used to build regular expressions in *magic* mode.

char — An ordinary character matches itself. The following characters are not ordinary characters and must be escaped (preceded) by a \ to be recognized: ↑ at the beginning of a line; **$** at the end of line; * as any character other than the first; and . (dot); \ (backslash); [(bracket); and ~ (tilde).

↑ — At the beginning of a pattern, forces the match to succeed only at the beginning of a line.

$ — At the end of a regular expression, forces the match to succeed only at the end of the line.

. — Matches any single character except the newline character.

* — Matches any number (including 0) of adjacent occurrences of the regular expression it follows.

\< — Forces a match to occur only at the beginning of a variable or word, including a variable or word at the beginning of a line.

\ > — Forces a match to occur only at the end of a variable or word.

[*string*] — Matches any single character in the string and no others. A pair of characters separated by - in *string* defines the set of characters between the specified lower and upper bounds, thus **[a-z]** as a regular expression matches any single lowercase letter. If the first character of the string is an ↑, then the construct matches those characters that it otherwise would not; thus **[↑a-z]** matches anything except a lowercase letter (and a newline character). To place any of the characters ↑, **[**, or -

in *string* you must escape them with a preceding \ .

4.3.8.4 Combining Regular Expression Primitives. A concatenation of regular expressions results in a single regular expression. This combined regular expression matches a concatenation of strings when each string matches the corresponding component of the expression, read from the left. For example, the combined regular expression

/an.*an/

matches the following standard input text

Bevan, Litterman, Packer, Smith

The expression is defined as the occurrence of "an" followed by any character repeated any number of times followed by a second occurrence of "an". This expression could also be written as

/\(an\).*\1/

where the expression \n means the same string of characters matched by an expression enclosed in \(and \) earlier in the same expression. The *n* is a single digit; the sequence \n is replaced by the text matched by the *n*th regular subexpression enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left.

The character ~ may be used in a regular expression to match the text that defined the replacement part of the last substitute command.

4.3.8.5 Substitute Replacement Patterns. The basic metacharacters for the replacement pattern are **&** and ~; these are given as \ **&** and \~ when *nomagic* is set. Each instance of **&** is replaced by the characters that the regular expression matched. The metacharacter ~ stands (in the replacement pattern) for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escape character \ . The sequences \ **u** and \ **l** cause the immediately following character in the replacement to be converted to uppercase or lowercase, respectively, if this character is a letter. The sequences \ **U** and \ **L** turn such conversion on, either until \ **E** or \ **e** is encountered or until the end of the replacement pattern.

4.3.9 Option Descriptions.

autoindent, ai (default: **noai**)

Can be used to ease the preparation of structured program text. Autoindent operates within *open* or *visual* mode with the **append, change, insert, substitute** and **O** (open new line) commands. With **append**, *ex* looks at the beginning of the line that is being appended and duplicates its indentation for any new lines created during the edit. With **change, insert, substitute** and **O** commands, *ex* calculates the amount of white space at the start of a new line in the edit and then aligns the cursor with that indentation for each succeeding line of the edit.

If additional white space is typed at the beginning of a line, all lines that follow will be aligned with the first non-white character of the previous line. To back the cursor to the preceding tabstop, hit ^**D**. The tabstops (going backwards) are defined as multiples of the *shiftwidth* option. You cannot backspace over the indent, except by sending an end-of-file with a ^**D**.

A line with no character in it turns into a completely blank line (the white space provided for the *autoindent* is discarded). Lines beginning with an ↑ and immediately

followed by a ^D will reposition the input text at the beginning of the line while retaining the previous indent for the next line. Similarly, a **0** followed by a ^D will reposition the input text at the beginning of the line without retaining the previous indent.

The *autoindent* option does not operate in **global** commands or when the input is not a terminal.

autoprint, ap (default: **ap**)
Causes the current line to be printed after each **delete, copy, join, move, substitute, t, undo** or **shift** command. This has the same effect as supplying a trailing **p** to each such command. The *autoprint* is suppressed in globals and only applies to the last of many commands on a line.

autowrite, aw (default: **noaw**)
Causes the contents of the buffer to be written to the current file if you have modified it and give a **next, rewind, tab**, or **!** command, or a ^↑ (switch files) or ^] (tag goto) command in *visual* mode.

Note: The command does not autowrite. In each case, there is an equivalent way of switching when the *autowrite* option is set to avoid the autowrite (**ex** for **next, rewind!** for **rewind, tag!** for **tag, shell** for **!**, and **:e #** and a **:ta!** command from within *visual* mode).

beautify, bf (default: **nobeautify**)
Causes all control characters except tab, newline, and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. The *beautify* option does not apply to command input.

directory, dir (default: **dir=/tmp**)
Specifies the directory in which *ex* places its buffer file. If this directory in not writeable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible (default: **noedcompatible**)
Causes the presence or absence of **g** and **c** suffixes on substitute commands to be remembered and to be toggled by repeating the suffices. The suffix **r** makes the substitution similar to the ~ command, instead of the **&**. (Version 3 only.)

errorbells, eb (default: **noeb**)
Error messages are preceded by a bell. (Bell ringing in *open* and *visual* mode on errors is not suppressed by setting *noeb*.) If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht (default: **ht=8**)
Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic (default: **noic**)
All uppercase characters in the text are mapped to lowercase in regular expression matching. In addition, all uppercase characters in regular expressions are mapped to lowercase except in character class specifications.

lisp (default: **nolisp**)
The *autoindent* option indents appropriately for *lisp* code, and the (), {}, [[, and]] commands in *open* and *visual* modes are modified to have meaning for *lisp*.

list (default: **nolist**)
All printed lines will be display tabs and end-of-lines markers as in the **list** command.

magic (default: **magic** for **ex** and *vi*)
If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only \, ↑ and **$** having special effects. In addition, the metacharacters ~ and **&** of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a \.

mesg (default: **mesg**)
Causes write permission to the terminal to be turned off while you are in *visual* mode, if *nomesg* is set. (Version 3 only.)

number, nu (default: **nonumber**)
Causes all output lines to be printed with line numbers. In addition, each input line will be prompted by supplying the next line number.

open (default: **open**)
If *noopen*, the commands **open** and **visual** are not permitted.

optimize, opt (default: **optimize**)
Directs the terminal to skip automatic carriage returns when printing more than one (logical) line of output. This will speed output on terminals without addressable cursors whenever text with leading white space is printed.

paragraphs, para (default: **para**=IPLPPPQPP LIbp)
Specifies paragraphs for the { and } operations in *open* and *visual* modes. The character pairs in the option's value are the names of the macros that start paragraphs.

prompt (default: **prompt**)
Command mode input is prompted for with a colon (:).

readonly (default: **noreadonly**)
Can be used to set the permission mode to read-only from within the editor. It is possible to write to a file using the **!** form of write, even while in read-only mode.

redraw (default: **noredraw**)
On a dumb terminal, the editor uses great amounts of output to simulate an intelligent terminal. For example, during insertions in *visual* mode the characters to the right of the cursor position are refreshed as each input character is typed. This option is useful only at very high speed.

remap (default: **remap**)
If on, macros are repeatedly tried until they are unchanged. (Version 3 only.) Assume, for example, **o** is mapped to **O**, and **O** is mapped to **I**. If *remap* is set, **o** will map to **I**; if *noremap* is set, it will map to **O**.

report (default: **report**=5)

Specifies a threshold for feedback from commands. Any command that changes more than the specified number of lines will provide feedback about the scope of its changes. For commands such as **global**, **open**, **undo**, and **visual**, which have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus, notification is suppressed during a **global** command on the individual commands performed.

scroll (default: **scroll**=½ window)
Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in *command* mode and the number of lines printed by a *command* mode **z** command (double the value of *scroll*).

sections (default: **sections**=SHNHH HU)
Specifies the section macros for the [[and]] operations in *open* and *visual* modes. The pairs of characters in the option's value are the names of the macros that start paragraphs.

shell, sh (default: **sh**=/bin/sh)
Gives the pathname of the shell forked for the shell escape command **!**, and by the **shell** command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw (default: **sw**=8)
Gives the width for a software tabstop. The tabstop is used with *autoindent* to reverse tab (with **^D**) and by the shift commands.

showmatch, sm (default: **nosm**)
In *open* and *visual* modes, when a) or } is typed, *showmatch* moves the cursor to the matching (or { for one second, if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow (terminal dependent)
Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent.

tabstop, ts (default: **ts**=8)
The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl (default: **tl**=0)
Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags (default: **tags**=tags/usr/lib/tags)
A path of files to be used as tag files for the *tag* command. (Version 3 only.) A requested tag is searched for in the specified files sequentially. By default, files called **tags** are searched for in the current directory and in **/usr/lib** (a master file for the entire system).

term (from environment TERM)
The terminal type of the output device.

terse (default: **noterse**)
Shorter error diagnostics are produced for the experienced user.

warn (default: **warn**)
Warn if there has been "[No write since last change]" before a **!** command escape.

window (default: **window**=speed dependent)
The number of lines in a text window in the **visual** command. The default is 8 lines at slow speeds (600 baud or less), 16 lines at medium speed (1200 baud), and the full screen (minus 1 line) at higher speeds.

w300, w1200, w9600
These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws (default: **ws**)
Searches that use regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm (default: **wm**=0)
Defines a margin for automatic wrapover of text during input in *open* and *visual* modes.

writeany, wa (default: **nowa**)
Inhibits checks normally made before **write** commands, allowing a write to any file that the system protection mechanism will allow.

The options described are of three kinds: numeric, string and toggle. You can set numeric and string options by a statement of the form

set *opt=val*

and toggle options can be set or canceled by statements of one of the forms

set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment or given while you are running *ex* by preceding them with a **:** and following them with a CR.

You can get a list of all options that you have changed with the command

:setCR

or the value of a single option by the command

:set *opt*?CR

A list of all possible options and their values is generated by

:set allCR

Set can be abbreviated **se**. Multiple options can be placed on one line, for instance:

:se ai aw nuCR

Options set by the **set** command last only while you stay in the editor. It is common to want certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands that are run every time you start up *ex*, *edit*, or *vi*. (All commands that start

with : are *ex* commands.) A typical list includes a **set** command and possibly a few **map** commands (on Version 3 editors). Try to get these commands on one line separated with the | character; for example,

set ai aw terse|map @ dd|map # x

which establishes the **set** command options *autoindent*, *autowrite*, *terse*, makes @ delete a line (the first **map**), and makes # delete a character (the second **map**) (see subpart 4.4.7.9 for a description of the Version 3 **map** command). This string could be placed in the variable EXINIT in your environment. Using the shell, you could also put these lines in the file **.profile** in your home or working directory:

EXINIT=set ai aw terse|map @ dd|map # x
export EXINIT

Or the following line could be put in the file **exrc** in your home directory:

set ai aw terse|map @ dd|map # x

The options you select, if any, will depend on your working environment.

4.3.10 Limitations. The user is likely to encounter the following editor limits:

1024 characters per line
256 characters per global command list
128 characters per filename
128 characters in the previous inserted and deleted text in *open* or *visual* modes,
100 characters in a shell escape command
63 characters in a string valued option
30 characters in a tag name
250,000 lines if the file is silently enforced.

The *visual* implementation limits to 32 the number of macros defined with map, and the total number of characters in macros must be less than 512.

4.4 The vi Text Editor

4.4.1 General. This section provides an introduction to the *vi* (visual) editor, versions 2 and 3. Version 2 is the version of *vi* that runs on the PDP11; Version 3 runs on 32-bit machines.

You should be running *vi* on a file you are familiar with while reading this. Sections 4.4.2 through 4.4.6 describe the basics for using *vi* and include the display editing features of the *ex* editor. Some topics of special interest are presented in sections 4.4.7 and 4.4.8; additional information about the editor is given in section 4.4.9 to avoid cluttering the initial presentation.

A summary of commands, control characters, and key functions is provided at the end of sections 4.4.3 and 4.4.4. The summary gives the name of command, paragraph of reference, and a short description. Section 4.4.10 provides a complete list of characters and their special meanings to the *vi* editor.

The following discussions refer to commands that are generated by pressing the control key at the same time you hit another key. We use the notation ^ to indicate the control key; for example, **^D** is a command generated by pressing the control key while hitting the D key. You may have a key labeled ^ on your terminal. The ^ key will be represented as ↑ in this document; the ^ is used exclusively as part of the notation for control characters.

In the command examples shown in this part:

- Input that must be typed "as is" will be presented in **boldface** type.
- Text that should be replaced wich appropriate input will be given in *italics*.

4.4.2 Getting Started.

4.4.2.1 Specifying Terminal Type. Before you can start *vi*, you must tell the system what kind of terminal you are using. An incomplete list of terminal type codes follows. If your terminal does not appear here, consult with the staff members on your system to learn the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal created.

CODE	FULL NAME	TYPE
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3A	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart ACT-IV	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is 2621. The command sequence

```
TERM=2621
export TERM
```

would tell the system you have a Hewlett-Packard 2621A/P.

If you want to have your terminal type established automatically when you log in, place the above commands in your **.profile**.

4.4.2.2 Editing a File. After telling the system which kind of terminal you have, make a copy of a file you are familiar with and run *vi* on this file with the command

$ vi *filename*

Replace *filename* with the name of the copy file just created. The screen should clear and the text of your file appear on the screen. If something else happens, you may have given the system an incorrect terminal type code. Another possibility is that you may have typed the wrong filename and the editor printed an error diagnostic. If something unexpected appears on your screen, hit the keys **:q** (colon and the q key) and then the RETURN key. This should take you out of the editor and get you back to the command level interpreter. Try to figure out what happened, then attempt the procedure again.

If the editor doesn't respond to the commands, try sending it an interrupt by hitting the DEL or RUB key, and then giving the **:q** command, again followed by a carriage return.

4.4.2.3 Editor Copy in Buffer. The *vi* editor does not directly change the file that you are editing. Instead, it makes a copy of this file in the buffer and remembers the filename. You do not affect the contents of the file until you write the changes into the original file.

4.4.2.4 Arrow Keys. The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labeled with arrows on an *adm3a*).

- *h* moves cursor to the left (control-h does the same)
- *j* moves cursor down (in the same column)
- *k* moves cursor up (in the same column)
- *l* moves cursor to the right.

Note: On the HP2621 terminal, the function keys must be used with the shift key, otherwise they only act locally. Unshifted use will leave the cursor positioned incorrectly.

4.4.2.5 Special Characters. Several special characters are very important, so be sure to find them right away. Look on your keyboard for a key labeled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in an inactive state. On smart terminals, the editor may quietly flash the screen rather than ring the bell. Partially formed commands are canceled with the ESC key. When you insert text in the file, text insertion is ended with the ESC key. If you become confused during an edit, you may hit the ESC key to cancel any operation and start again.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard and is the same command used at the end of each shell command.

Another useful key is the DEL (or RUB key). It generates an interrupt, which tells the editor to stop what it is doing. This is a forceful way of making the editor return to an inactive state. Try hitting the / key on your terminal. It is used when you want to specify a search string. The cursor should now be positioned at the bottom line of the terminal after a / printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key. Backspacing over the / will also cancel the search. From now on we will refer to hitting the DEL or RUB key as "sending an interrupt".

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt. (On some systems, you cannot type ahead while the editor is computing with the cursor on the bottom line.)

4.4.2.6 Leaving the Editor. After you have practiced working with *vi* and you wish to do something else, you can give the **ZZ** command to leave the editor. This writes the contents of the editor buffer (including any changes made) back into the file you are editing and then quits the editor. You can also end an editor session by giving the command **:q!**CR. All commands that read from the last display line can also be terminated with an ESC as well as a CR character. The **:q!**CR command ends the editor session and discards all your changes. This command is useful if you change the editor copy of a file you wish only to look at. Be

very careful **not** to give this command when you really want to save the changes you have made.

4.4.3 Moving Around in the File.

4.4.3.1 Scrolling and Paging. The editor has many commands for moving around in the file. The most used command is generated by hitting the control and D keys at the same time, a control-D or **^D**. This command scrolls down in the file; the **D** stands for down. Many editor commands are mnemonic which makes them much easier to remember. For instance, the command to scroll up is **^U**. Many dumb terminals can't scroll up at all, in which case hitting **^U** clears the screen and refreshes it with a line that is farther back in the file at the top.

If you want to see more of the file below where you are, you can hit **^E** to expose one more line at the bottom of the screen, leaving the cursor where it is. (Version 3 only.) The command **^Y** (which is non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen.

There are other ways to move around in the file; the keys **^F** and **^B** move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these commands as well.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **^F** to move forward a page repeats only a couple of lines of text. Scrolling on the other hand gives more context and functions more smoothly. You can continue to read the text while scrolling is taking place.

4.4.3.2 Searching, goto, and Previous Context. Another way to position yourself in the file is by giving the editor a string to search for. Type the character **/** followed by a string of characters terminated by a RETURN. The editor will position the cursor at the next occurrence of this string. Now try hitting **n** to go on to the next occurrence of this string. The character **?** will search backward from where you are but is otherwise like **/**.

Searches will normally wrap around the end of the file and find the string even if it is not on a line in the direction you searched, provided it is anywhere else in the file. You can disable this wrap-around in scans by giving the command:

:se nowrapscanCR
or
:se nowsCR

If the search string you give the editor is not present in the file, the editor will print a diagnostic on the last line of the screen and the cursor will return to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an **↑**. To match only at the end of a line, end the search string with a **$**. Thus

/↑firstCR

will search for the word 'first' at the beginning of a line, and

/last$CR

searches for the word 'last' at the end of a line.

Actually, the string you give to search for can be a *regular expression* in the sense of the editors *ex*(1) and *ed*(1). (Refer to section **4.3.8.3** for more information on *regular expression.*) You can disable the special meanings of these characters by putting the

:se nomagicCR

command in EXINIT in your environment.

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, then the cursor moves to the end of the file.

If you are near the end of the file and the last line is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are beyond the end of the file.

You can determine the state of the file you are editing by typing a ^**G** command. The editor will show you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer the cursor is located. Try doing this now and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you were.

You can also get back to a previous position by using the command **``** (two back quotes). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with **/** or **?** and then a **``** to get back where you started. If you accidentally hit **n** or any command that moves you far away from a context of interest, you can quickly get back by hitting **``**.

4.4.3.3 Moving Around on the Screen. Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction), try them. If you don't have working arrow keys, use **h**, **j**, **k**, and **l**. Experienced users of *vi* prefer these keys to arrow keys, because they are right underneath their fingers.

Hit the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The - key is like + but goes the other way. These are very common keys for moving the cursor up and down lines in the file. Notice that if the cursor goes off the bottom or top of the screen when using these keys, then the text will scroll down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the + key.

The *vi* editor also has commands to take you to the top, middle and bottom of the screen. The **H** command will take you to the top line (home) on the screen. Try preceding it with a number as in **3H**. This will take you to the third line on the screen. Many *vi* commands take preceding numbers. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last line on the screen. The **L** command also takes counts; the **5L** command will take you to the fifth line from the bottom.

There are two control characters that move the cursor up or down a line, but keep it in the same column. The ^**N** causes the cursor to move to the same column of the next line. To move to the same column of the previous line use the ^**P** command.

4.4.3.4 Moving Within a Line. Now try picking a word on some line on the screen, not the first word on the line. Move the cursor (using RETURN and -) to be on the line where the word is. Try hitting the **w** key. This will advance the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Also try the **e** key which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves the cursor right one character and the BS key (backspace or^**H**) which moves left one character. The **h** key works as ^**H** does and is useful if you don't have a BS key. Also, as noted above, the **l** key will move the cursor to the right.

If the line has punctuation in it you may have noticed that the **w** and **b** keys stop at each group of punctuation. You can also go backward and forward without stopping at punctuation by using **W** and **B** rather than the lowercase equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lowercase **w** and **b.**

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w.**

4.4.3.5 Summary of Cursor Commands. A paragraph reference is provided for each *vi* command in the following list.

BS	4.4.3.4	Move cursor one position to the left
SPACE	4.4.3.4	Move cursor one position to the right
^B	4.4.3.1	Move backward to previous page
^D	4.4.3.1	Scroll down in the file
^E	4.4.3.1	Expose another line at bottom of screen (Version 3)
^F	4.4.3.1	Move forward to next page
^G	4.4.3.2	Determine state of file (filename, current line number, number of lines in the buffer, and per-cent way through the buffer)
^H	4.4.3.4	Move cursor one space to the left
^N	4.4.3.3	Move cursor to next line, same column
^P	4.4.3.3	Move cursor to previous line, same column
^U	4.4.3.1	Scroll up in the file
^Y	4.4.3.1	Expose another line at the top of screen (Version 3)
+	4.4.3.3	Advance cursor to beginning of next line
-	4.4.3.3	Move cursor to beginning of previous line
/	4.4.3.2	Search forward for a character string
?	4.4.3.2	Search backward for a character string
B	4.4.3.4	Move cursor backward a word, ignoring punctuation
G	4.4.3.2	Move cursor to specified line or to last line if default
H	4.4.3.3	Move cursor to top line (home) on screen
M	4.4.3.3	Move cursor to middle line on screen
L	4.4.3.3	Move cursor to last line on screen
W	4.4.3.4	Move cursor forward a word, ignoring punctuation
b	4.4.3.4	Move cursor backward a word.
e	4.4.3.4	Move cursor to end of current word
h	4.4.2.4	Moves cursor to the left
j	4.4.2.4	Moves cursor down (in same column)

k	4.4.2.4	Moves cursor up (in same column)
l	4.4.2.4	Moves cursor to the right
n	4.4.3.2	Repeat scan for next instance of / or ? pattern
w	4.4.3.4	Move cursor forward a word

4.4.3.6 The view Editor. If you want to use the editor to look at a file, rather than to make changes, invoke it as **view** instead of **vi**. This will set the *readonly* option which will prevent you from accidentally overwriting the file.

4.4.4 Making Simple Changes.

4.4.4.1 Inserting. One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself on some word in the file and try inserting text before this word. If you are on a dumb terminal it will seem that some of the characters in your line have been overwritten, but they will reappear when you hit ESC.

Now try finding a word that can, but does not, end in an 's'. Position yourself at that word and type **e** (move to end of word), **a** (append), and 'sESC' (terminate the textual insert). This sequence of commands can be used to make a word plural.

Try inserting and appending a few times to make sure you understand how this works:

- **i** places text to the left of the cursor
- **a** places text to the right of the cursor.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lowercase key and the other is given by an uppercase key. In these cases, the uppercase key often differs from the lowercase key in its sense of direction, with the uppercase key working backward and/or up, while the lowercase key moves forward and/or down.

Often you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the **o** command to create a new line after the line you are on, or the **O** command to create a new line before the line you are on. After you create a new line in this way, the text you type up to an ESC is inserted on the new line.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed and the missing lines will reappear when you hit ESC.

While inserting new text, you can use the characters normally used at the system command level (usually **^H** or **#**) to backspace over the last character typed, and the character used to kill input lines (usually **@**, **^X**, or **^U**) can be used to erase the input you have typed on the current line. In fact, the **^H** (backspace) always works to erase the last input character, regardless of what your erase character is. The **^W** will erase a whole word and leave you after the space following the previous word. It is useful for quickly backing up in an insert. The following conditions should be noted:

- When you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backward, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when you hit ESC. If you want to get rid of them immediately, hit an ESC and then **a** again.
- You cannot erase characters that you did not insert, and you cannot backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

4.4.4.2 Making Small Corrections. You can make small corrections in existing text quite easily. Find a single character that is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key, **^H**, or just **h**) or SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you **x** out characters when you make mistakes on a typewriter.

If the character is incorrect, you can replace it with the correct character by giving the **r***c* command, where *c* is the correct character. If the character that is incorrect should be replaced by more than one character, give the command

s*string*ESC

which substitutes a string of characters, and ends with ESC. If there are a small number of characters that are wrong you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

4.4.4.3 Making Corrections With Operators. You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator and the **c** key acts as a change operator.

- The **dw** command deletes a following word.
- The **db** command deletes a preceding word.
- The **d**SPACE command deletes a single character, and is equivalent to the **x** command.
- The **cw** command changes the text of a single word. It is followed with replacement text ending with an ESC. Find a word that you can change to another and try this now. Notice that the end of the text to be changed is marked with the character '$' so that you can see this mark as you are typing in the new text material.

The **.** command repeats the last command that made a change.

4.4.4.4 Operating on Lines. It is often the case that you want to operate on lines. Find a line you want to delete and type **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability. **D** deletes the rest of the text on the current line.

Try repeating the **c** operator twice (**cc**); this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC. The command **S** is a convenient synonym for **cc**, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters. **C** changes the rest of the text on the current line.

You can delete or change more than one line by preceding the **dd** or **cc** with a count (**5dd** deletes 5 lines). You can also give a command like **dL** to delete all the lines up to and including the last line on the screen or **d3L** to delete through the third from the bottom line. One subtle point here involves using the / (search) after a **d**. This will normally delete characters from the current position to the point of the match. If you desire to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

Note: The *ex* editor lets you know when you change a large number of lines so that you can see the extent of the change. It will also always tell you when a change you make affects text that you cannot see.

4.4.4.5 Undoing. Suppose the last change you made was incorrect; you could use the insert, delete, and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change you made. Try this a few times, and give it twice in a row to notice that a **u** also undoes a **u**.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text that you deleted, even if undo will not bring it back; see subpart 4.4.7.3 on recovering lost text.

4.4.4.6 Summary of Basic vi Commands. A paragraph reference is provided for each command in the following list.

SPACE	4.4.4.2	Advance cursor one position to the right
^H	4.4.4.1	Move cursor one space to the left
^W	4.4.4.1	Erase a word during an insert
erase	4.4.4.1	Erase a character during an insert (usually **^H** or **#**)
kill	4.4.4.1	Kill the insert on this line (usually **@**, **^X**, or **^U**)
.	4.4.4.3	Repeat the last change command
O	4.4.4.1	Open and input new lines above the current line
S	4.4.4.4	Substitute on lines
U	4.4.4.5	Undo the changes made to the current line
a	4.4.4.1	Append text after the cursor
c	4.4.4.3	Change the specified object (word) to the following text
d	4.4.4.3	Delete the specified object (word, space, etc.)
i	4.4.4.1	Insert text before the cursor
o	4.4.4.1	Open and input new lines below the current line
r	4.4.4.2	Replace a character
s	4.4.4.2	Replace a character with a string
u	4.4.4.5	Undo the last change

x	4.4.4.2	Delete a character
cc	4.4.4.4	Change a whole line
dd	4.4.4.4	Delete a line

4.4.5 Moving About, Rearranging, and Duplicating Text.

4.4.5.1 Low Level Character Motions. Move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is the character sought. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a **;**, which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s, you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also an **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

When operating on the text in a line, it is often desirable to delete characters including the first instance of a character. Try **df***x* for some *x* and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x* (the **t** stands for "to") to delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, an **↑** moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus, **$a** will append new text at the end of the current line.

Your file may have tab characters (**^I**) in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions by default. Tab stops are set by a command of the form

:se ts= *x*CR

where *x* is 4 to set tab stops every four columns. The tab stop setting has an effect on screen representation within the editor. When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have non-printing characters in it. These characters are displayed in the same way they are represented in this manual; i.e., with a 2-character code, the first character of which is the '^' character. On the screen non-printing characters resemble a '^' character adjacent to another character. Spacing or backspacing over the character reveals that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a **^V** before the control character. The **^V** quotes the following character causing it to be inserted directly into the file.

4.4.5.2 Higher Level Text Objects. In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections.

- The **(** and **)** operations move to the beginning of the previous and next sentence, respectively. Thus the **d)** command will delete the rest of the current sentence. The **d(** command will:

delete the previous sentence if you are at the beginning of the current sentence
or
delete the current sentence up to where you are if you are not at the beginning of the current sentence.

- A sentence is defined to end at a ., !, or ? that is followed by either the end of a line or two spaces. Any number of),], ", and ' closing characters may appear after the ., !, or ?, and before the spaces or end of line.
- The { and } operations move over paragraphs.
- The [[and]] operations move over sections. They require the operation character to be doubled because they can move the cursor from where it currently is. While it is easy to get back with the " command, these commands would still be frustrating if they were easy to hit accidentally.
- A paragraph begins after each empty line and also at each of a set of paragraph macros specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the -ms and -mm macro packages, i.e., the .IP, .LP, .PP and .QP, .P and .LI macros. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See subpart 4.4.7.2 for details. The **.bp** request is also considered to start a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.
- Sections in the editor begin after each macro in the *sections* option, normally, .NH, .SH, .H and .HU, and each line with a form feed ^L in the first column. Section boundaries are always line and paragraph boundaries.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking at it using the section commands. Section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command and a small count to see each successive section heading in a small window.

4.4.5.3 Rearranging and Duplicating Text. The editor has a single unnamed buffer where the last deleted or changed text is saved, and a set of named buffers **a-z** that you can use to save copies of text and to move text around in your file and between files.

The **y** operator yanks a copy of the object that follows into the unnamed buffer. If preceded by a buffer name, "***x*****y**, where ***x*** is replaced by a letter **a-z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; the **p** command puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text that you yank forms a part of a line or is an object such as a sentence, which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines without changing the current line. In this case, the put acts much like an **o** or **O** command (subpart 4.4.4.1).

Try the **YP** command. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in **"a5dd** deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of these lines and do an **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back by giving a command of the form

:e *name*CR

where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files; so to move text from one file to another, you should use a named buffer.

4.4.5.4 Summary of Advanced vi Commands. A paragraph reference is provided for each command in the following list.

^I	4.4.5.1	Tab, add spaces up to next tab stop
^L	4.4.5.2	Form feed
^V	4.4.5.1	Quote the following character
↑	4.4.5.1	Move cursor to first non-white position on line
$	4.4.5.1	Move cursor to end of line
)	4.4.5.2	Balance of sentence forward
}	4.4.5.2	Move cursor forward over paragraph operator
]]	4.4.5.2	Move cursor forward over section operator
(	4.4.5.2	Balance of sentence backward
{	4.4.5.2	Move cursor backward over paragraph operator
[[	4.4.5.2	Move cursor backward over section operator
;	4.4.5.1	Find next instance of same character found with the **f***x* or **F***x*
d	4.4.5.1	Delete the specified object (character, word, space)
f*x*	4.4.5.1	Find the first *x* character to right of the cursor
p	4.4.5.3	Put text back, after cursor or below current line
u	4.4.5.1	Undo the last change
y	4.4.5.3	Yank operator, for copies and moves
t*x*	4.4.5.1	To *x* forward, for operators
F*x*	4.4.5.1	Find the first *x* character to left of the cursor
P	4.4.5.3	Put text back, before cursor or above current line
T*x*	4.4.5.1	Up to *x* backward in line

4.4.6 High-Level Commands.

4.4.6.1 Writing, Quitting, and Editing New Files. So far we have seen how to enter the *vi* editor and to write out our file using either **ZZ** or **:w**CR commands. The first command exits from the editor (writing if changes were made), the second command writes and stays in the editor.

If you have changed the editor copy of the file but do not wish to save your changes, then you can give the command

:q!CR

to quit from the editor without writing the changes. You can also re-edit the same file (start over) by giving the command

:e!CR

These commands should be used only rarely and with caution, since it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command

:e *name*CR.

If you have not written out your file before you try to do this, then the editor will tell you this and delay editing the other file. You can then give the command

:wCR

(to save your changes) and then the

:e *name*CR

command again or carefully give the command

:e! *name*CR

which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT and use **:n** instead of **:e**.

4.4.6.2 Escaping to a Shell. You can get to a shell to execute a single command by giving a *vi* command of the form

:!*cmd*CR

The system will run the single command (*cmd*), and when finished, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN; the editor will clear the screen and redraw it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case, the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command

:shCR

This will give you a new shell. When finished with the shell, end it by typing a **^D**. The editor will clear the screen and continue.

4.4.6.3 Marking and Returning. The command **''** returns you to the previous place after a motion of the cursor by a command such as **/**, **?**, or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, such as an **a**.

Then move the cursor to a different line (any way you like) and hit **'a**. The cursor will return to the place that you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case you can use the form *'x* rather than *`x*. Used without an operator, *'x* will move to the first non-white character of the marked line; similarly, '' moves to the first non-white character of the line containing the previous context mark ''.

4.4.6.4 Adjusting the Screen. If the screen image contains random or unwanted characters because of a transmission error to your terminal or because some program other than the editor wrote output to your terminal, you can hit a **^L**, the ASCII form-feed character, which will cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to cause the editor to retype the screen, closing up these holes.

If you wish to place a certain line on the screen at the top, middle, or bottom of the screen, you can position the cursor to that line and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a **.** if you want it at the center, or a **-** if you want it at the bottom.

4.4.7 Special Topics.

4.4.7.1 Editing on Slow Terminals. When you are on a slow terminal, it is important to limit the amount of output that is generated to your screen so that you will not suffer long delays waiting for the screen to be refreshed.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command **:se slow**CR. If your system is sluggish, this helps lessen the amount of output coming to your terminal. You can disable this option by **:se noslow**CR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command **:se redraw**CR. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this with the **:se noredraw**CR command.

The editor also makes editing at low speed more pleasant by starting the edit in a small window and letting the window expand as you work. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window size as an argument to the commands that cause large screen motions:

: / ? [[ ]] ' '

Thus if you are searching for a particular instance of common string in a file you can precede the first search command by a small number, such as 3, and the editor will draw 3-line windows around each instance of the string that it locates.

You can easily expand or contract the window and place the current line as you choose by giving a number with the **z** command (after the **z** and before the following RETURN, **.** or **-**). Thus the command **z5.** redraws the screen with the current line in the center of a 5-line

window.

Note: The command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a DEL or RUB as usual. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a **^L**; or move or search again, ignoring the current state of the display.

See subpart 4.4.9.8 on *open* mode for another way to use the *vi* command set on slow terminals.

4.4.7.2 Options, Set, and Editor Startup Files. The editor has a set of options, some of which have been mentioned above. The most commonly used options are defined in the following table.

OPTION	DEFAULT	DESCRIPTION
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before **:n**, **:ta**, **^↑**, and **!**
ignorecase	noic	Ignore case in searching
lisp	nolisp	**(** { **)** } commands deal with S-expressions
list	nolist	Tabs print as **^I**; end of lines marked with **$**
magic	nomagic	The characters [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names that start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names that start new sections
shiftwidth	sw=8	Shift distance for **<**, **>** and input **^D** and **^T**
showmatch	nosm	Show matching **(** or { as **)** or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using

The options are of three kinds: numeric, string, and toggle. You can set numeric and string options by a statement of the form

set *opt=val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment or given while you are running *vi* by preceding them with a **:** and following them with a CR.

You can get a list of all options that you have changed with the command

:setCR

or the value of a single option by the command

:set *opt*?CR

A list of all possible options and their values is generated by

:set allCR

Set can be abbreviated **se**. Multiple options can be placed on one line, for instance:

:se ai aw nuCR

Options set by the **set** command last only while you stay in the editor. It is common to want certain options set whenever you use the editor. Refer to section 4.3.9. for a full explanation of how to create a list of *ex* commands that are to be run every time you start up *ex*, *edit*, or *vi*.

4.4.7.3 Recovering Lost Lines. Occasionally, you may delete a number of lines and then regret that they were deleted. As backup, the editor saves the last nine deleted blocks of text in a set of numbered registers, 1 through 9. You can retrieve the *n*th previously deleted text block by the command **"*n*p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command that puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this, then **.** (period) to repeat the put command. In general, the **.** command will repeat the last change. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

"1pu.u.u

if repeated long enough, will display all the deleted text that has been saved. You can omit the **u** commands here to gather up all this text in the buffer or stop after any **.** command to keep just the then-recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

4.4.7.4 Recovering Lost Files. If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next log in, giving you the name of the file that has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

:vi -r *name*

replacing *name* with the name of the file that you were editing. This will recover your work to a point near where you left off. (In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string LOST. These lines will almost always be among the last few that you changed.)

You can get a listing of the files that are saved for you by giving the command:

:vi -r

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. You can get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a superuser on your system, and the *mail* program must exist to receive mail. The invocation **:vi -r** will not always list all saved files, but they can be recovered even if they are not listed.

4.4.7.5 Continuous Text Input. When you are typing in large amounts of text, it is convenient to have lines broken near the right margin automatically. You can cause this to happen by setting the *wrapmargin* option

:se wm=10CRCR

This causes all lines to be broken at a space at least ten columns from the righthand edge of the screen. (This feature is not available on some Version 2 editors. In the editors that have *wrapmargin*, the break can only occur to the right of the specified boundary instead of to the left.)

If the editor breaks an input line and you wish to put it back together, you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined (as in **3J** to join 3 lines). The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

4.4.7.6 Features for Editing Programs. Several commands in the editor are designed to help you edit programs.

First, the editor has an *autoindent* facility to help generate correctly indented programs. To enable this facility, you can give the command **:se ai**CR. Open a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D** you back up one position, normally to an 8-column boundary. This amount is variable; the editor has an option called *shiftwidth* which you can set to change this value. Try giving the command

:see sw=4CR

and then experimenting with *autoindent* again.

For shifting lines in the program left and right, there are operators **<** and **>**. These shift the lines you specify right or left by one *shiftwidth*. Try **<<** and **>>** which shift one line left or right, and **<L** and **>L** shifting the rest of the display left and right.

A second aid to editing programs is the **%** command. If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit **%**. This will show you the matching parenthesis. This works also for braces ({}), and brackets ([]).

If you are editing C programs, you can use the [[and]] keys to advance or retreat to a line starting with a {, i.e., a function declaration. When]] is used with an operator, it stops after a line that starts with }; this is sometimes useful with **y]]**.

4.4.7.7 Filtering Portions of the Buffer. You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer or to reformat portions of the buffer when you are using a printer with a *beautify* option. Try entering a list of random words, one per line and ending it with a blank line. Backspace to the beginning of the list and give the command **!}sort**CR. This will sort the next paragraph of

material (your list). The blank line ends the paragraph.

4.4.7.8 Commands for Editing LISP. (The **LISP** features are not available on some Version 2 editors due to memory constraints.)

If you are editing a **LISP** program, you should set the option *lisp* by doing

:se lispCR

This changes the (and) commands to move backward and forward over s-expressions. The { and } commands are similar to (and) but don't stop at atoms. These can be used to skip to the next list or through a comment quickly.

The *autoindent* option works differently for **LISP** supplying indent to align at the first argument to the last open list. If there is no such argument, then the indent is two spaces more than the last level.

There is another option that is useful for typing in **LISP**, the *showmatch* option. Try setting it with **:se sm**CR and then try typing a (, some words, and a). Notice that the cursor shows the position of the (that matches the) briefly. This happens only if the matching (is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the = operator. Try the command **=%** at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing **LISP**, the **[[** and **]]** advance and retreat to lines beginning with a (and are useful for dealing with entire function definitions.

4.4.7.9 Macros.

NOTE: The macro feature is available only in Version 3 editors.

The *vi* editor has a macro facility so that when you enter a single keystroke, the editor will act as though you had entered some longer sequence of keystrokes. You can set up a macro if you find yourself typing the same sequence of commands (keystrokes) repeatedly.

Briefly, there are two methods for assigning and calling up macros:

a) One method is to put the macro body in a buffer register, such as *x*. You can then type **@x** to invoke the macro. The @ may be followed by another @ to repeat the last macro.

b) The second method is to use the map command from *vi* (typically in your EXINIT) with a command of the form:

:map *lhs rhs*CR

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke, either one character or one function key, because *lhs* must be entered within one second. However, if *notimeout* is set, you can type *lhs* as slowly as you wish and *vi* will wait for you to finish it before it echoes anything. The *lhs* can be no longer than ten characters; the *rhs* no longer than 100. To put a space, tab, or newline into *lhs*, you should escape the characters with a **^V** (it may be necessary to double the **^V** if the map command is given inside *vi* rather than in *ex*). Only newline characters inside the *rhs* need to be escaped.

To make the **q** key write and exit the editor, you can give the command

:map q :wq^V^VCR CR

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A **^V** is needed because without it the CR would end the **:** command rather than becoming part of the **map** definition. There are two **^V**'s because you are working within *vi*. The first CR is part of the *rhs*, the second terminates the **:** command.

Macros can be deleted with

unmap *lhs*

If the *lhs* of a macro is #0 through #9, it maps the particular function key instead of the 2-character # sequence (and need not be typed within one second). The form #x will mean function key *x* on all terminals so that terminals without function keys can access these definitions. The character # can be changed by using a macro in the usual way; to use tab, for example:

:map ^V^V^I #

This will not affect the *map* command, which still uses #, but just the invocation from *visual* mode.

The undo command reverses an entire macro call as a unit.

Placing a **!** after the word **map** causes the mapping to apply to *input* mode rather than *command* mode. Thus, to arrange for **^T** to be the same as four spaces in *input* mode, you can type:

:map! ^T ^V␣␣␣␣

where **␣** is a blank. The **^V** is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

4.4.8 Word Abbreviations. (Version 3 only.) A feature similar to macros in *input* mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are

:abbreviate
and
:unabbreviate

or

:ab
and
:una

and have the same syntax as **:map**. For example:

:ab eecs Electrical Engineering and Computer Sciences

causes the word **eecs** to be changed into the phrase **Electrical Engineering and Computer Sciences**. Word abbreviation is different from macros in that only whole words are affected. If **eecs** were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke as it should be with a macro.

4.4.9 Additional Information.

4.4.9.1 Line Representation in the Display. The editor folds long logical lines onto many physical lines in the display. Commands that advance lines will advance by logical lines and will skip over all the segments of a line in one motion. The | command moves the cursor to a specific column and is useful for getting near the middle of a long line to split it in half. Try **80|** on a line that is more than 80 columns long. (You can make long lines very easily by using **J** to join together short lines.)

The editor puts only full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, the editor will often clear just the lines to @ to save time (rather than rewriting the rest of the screen). You can always re-type the information on the screen by giving the **^R** command.

The editor can place line numbers before each line on the display. Give the command

:se nuCR

to enable this, and the command

:se nonuCR

to turn it off. Tabs will be represented as **^I** and the ends of lines indicated with '$' by giving the command

:se listCR

The following command removes the display of tabs and ends of lines:

:se nolistCR

Lines consisting of only the ~ character are displayed when the last line in the file is in the middle of the screen. These represent physical lines that are past the logical end of file.

4.4.9.2 Counts. Most *vi* commands can use a preceding count to affect their behavior in some way.

Commands that take a new window size as count often cause the screen to be redrawn; for example **: / ? [[ ]] ' and '**. If you anticipate this, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a - or similar command, or off the bottom with a command such as RETURN or **^D**. The window will revert to the last specified size the next time it is cleared and refilled. (However, **^L** only redraws the screen as it is.)

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified, using half the basic window size initially.

The simple insert commands use a count to specify a repetition of the inserted text. Thus,

10a +—ESC

will append the string 10 times, creating a grid-like string of text. Try it.

A few commands also use a preceding count as a line or column number, such as **z**, **G**, and **|**.

Except for the commands that ignore any counts (such as **^R**), most of the editor commands use a count to indicate a simple repetition of their effect. Thus, **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete

two more words with the **2.** command.

4.4.9.3 More File Manipulation Commands. The following table lists the file manipulation commands that you can use when you are in *vi*.

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as **ZZ**).
:e *name*	edit file*name*
:e!	re-edit, discarding changes
:e + *name*	edit, starting at end
:e +*n*	edit, starting at line *n*
:e #	edit alternate file
:w *name*	write *filename*
:w! *name*	overwrite *filename*
:*x*,*y***w** *name*	write lines *x* through *y* to *name*
:r *name*	read *filename* into buffer
:r *!cmd*	read output of *cmd* into buffer
:n	edit next file in argument list
:n!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with one with a **:w** and start editing a new file by giving a **:e** command, or set *autowrite* and use **:n** <file>.

If you make changes to the editor copy of a file, but do not wish to write them back, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. **Use this carefully.**

The **:e** command can be given a + argument to start at the end of the file, or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usually a scan like +/*pat* or +?*pat*. In forming new names to the **e** command, you can use the character %, which is replaced by the current filename, or the character **#**, which is replaced by the alternate filename. The alternate filename is generally the last name you typed other than the current file. Thus, if you try to do a **:e** and get a diagnostic that you have not written into the file, you can give a **:w** command and then a **:e #** command to redo the previous **:e**.

You can write part of the buffer to a file by finding the lines that bound the range to be written using **^G**, to obtain the line numbers, and giving these numbers after the **:** and before the **w**, separated by ,'s. You can also mark these lines with **m** and then use an address of the form '*x*,'*y* on the **w** command.

You can read another file into the buffer after the current line by using the **:r** command. You can similarly read in the output from a command, just use the **:r !***cmd* instead of a filename.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to respecify the list of files to be edited by giving the **:n** command a list of filenames, or a pattern to be expanded as you would have given it on the initial *vi* command.

If you are editing large programs, you will find the **:ta** command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as

ctags, to find quickly a function by name. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again. (The tag feature is not available in some Version 2 editors.)

4.4.9.4 More About Searching for Strings. When you are searching for strings in the file with **/** or **?**, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c**, or **y**, you may want to affect lines up to the line before the line containing the pattern. You can give a search of the form

/pat/ -n

to refer to the *n*th line before the next line containing *pat*, or you can use + instead of - to refer to the lines after the one containing *pat*. If you do not give a line offset, the editor will affect characters up to the exact matched place rather than whole lines; use +0 to affect characters up to the line that matches.

You can have the editor ignore the case of words in the searches it does by giving the command

:se icCR

The command

:se noicCR

turns this off.

Strings defined in searches may actually be regular expressions. If you do not want or need this facility, you should

set nomagic

in your EXINIT. In this case, only the characters ↑ and **$** are special in patterns. The character \ remains special (as it is most everywhere in the system), and may be used to get at the extended pattern-matching facility. It is also necessary to use a \ before a / in a forward scan or a ? in a backward scan. The following table gives the extended forms when **magic** is set.

↑	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	matches any character
\ <	matches the beginning of a word
\>	matches the end of a word
[*str*]	matches any single character in *str*
[↑*str*]	matches any single character not in *str*
[*x*-*y*]	matches any character between *x* and *y*
*	matches any number of the preceding pattern

If you use **nomagic** mode, then the **.** **[** and * primitives are given with a preceding \ .

4.4.9.5 More About Corrections In Input Mode. There are a number of characters that you can use to make corrections during input mode. These are summarized in the following table.

^H	deletes the last input character
^W	deletes the last input word
erase	erase character, same as **^H**
kill	kill character, deletes the input on this line
\	escapes a following **^H**, erase, and kill

ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*
0^D	kills all the *autoindent*
↑^D	same as **0^D**, but restores indent next line
^V	quotes the next non-printing character into the file

The most usual way of making corrections to input is by typing **^H** to correct a single character, or by typing one or more **^W**'s to back over incorrect words. If you use **#** as your erase character in the normal system, it will work like **^H**.

Your system kill character, normally **@**, **^X**, or **^U** will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters that you did not insert with this insertion command. To make corrections on the previous line after a new line has been started, you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue. The command **A** which appends at the end of the current line is often useful for continuing.

If you wish to type in your erase or kill character (such as **#** or **@**) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a **^V**. The **^V** echoes as a ↑ character on that the cursor rests. This indicates that the editor expects you to type a control character. In fact, you may type any character and it will be inserted into the file at that point.

Note: The editor does not allow the NULL (^@) character to appear in files. Also the LF (line feed or **^J**) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type **^S** or **^Q**, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

If you are using *autoindent*, you can backtab over the indent that it supplies by typing a **^D**. This backs up to a *shiftwidth* boundary. This works only immediately after the supplied *autoindent*.

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then **^D**. The editor will move the cursor to the left margin for one line and restore the previous indent on the next. You can also type a **0** followed immediately by a **^D** if you wish to kill all the indent and not have it come back on the next line.

4.4.9.6 Uppercase Only Terminals. If your terminal has only uppercase characters, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lowercase, and you can type uppercase letters by preceding them with a \. The characters

```
few { ~ } | ‘
```

are not available on such terminals, but you can escape them as

```
\(↑\)\!\’.
```

These characters are represented on the display in the same way they are typed (the \ character you give will not echo until you type another key).

4.4.9.7 Relation Between vi And ex Editors. The *vi* editor is actually one mode of editing within the editor *ex*. All of the : commands that were introduced above are available in *ex*. Likewise, most *ex* commands can be invoked from *vi* using :, and ending the command with a CR.

In rare instances, an internal error may occur in *vi*. You will get a diagnostic and be left in the command mode of *ex*. You can save your work and quit if you wish by giving a command **x** after the : that *ex* prompts you with, or you can reenter *vi* by giving *ex* a ***vi*** command.

Many operations are done more easily in *ex* than in *vi*. Systematic changes in line-oriented material are particularly easy. On occasion, you may want to escape from *vi* to *ex* to execute several line-oriented commands. You can quit *vi* completely by giving the command **Q**. To return to *vi*, give the command **:vi**. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing. (Refer to the *SYSTEM V/68 Document Processing Guide* for the editor *ed* more information about this style of editing.)

4.4.9.8 Open Mode: vi on Hard Copy Terminals and Glass tty's. (Not available in all Version 2 editors due to memory constraints.)

If you are on a hard copy terminal or a terminal that does not have a cursor that can move off the bottom line, you can still use the command set of *vi*, but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex*, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is displayed.

In *open* mode, the editor uses a single line window into the file. Moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open* mode: **z** and **^R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns to the current line.

If you are on a hard copy terminal, the **^R** command will retype the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters that are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals that can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

4.4.10 Character Functions Summary. This section shows how the *vi* editor interprets each character. Characters are presented in their order in the ASCII character set: control characters first, most special characters, digits, uppercase characters, and then lowercase characters.

Each character is defined with a meaning it has as a command and any meaning it has during an insert. If it has meaning only as a command, then only this is discussed. Numbers in parentheses indicate where the character is discussed.

^@ Not a command character.
If typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters have been inserted the mechanism is not available. A ^@

cannot be part of the file.
(4.4.7.9, 4.4.9.5)

^A Unused.

^B Backward window. A count specifies repetition. Two lines of continuity are kept if possible.
(4.4.3.1, 4.4.9.2)

^C Unused.

^D As a command, it scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ^D and ^U commands.
(4.4.3.1, 4.4.9.2)
During an insert, it backtabs over *autoindent* white space at the beginning of a line. This white space cannot be backspaced over.
(4.4.7.6, 4.4.9.5)

^E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only)
(4.4.3.1)

^F Forward window. A count specifies repetition. Two lines of continuity are kept if possible
(4.4.3.1, 4.4.9.2)

^G Equivalent to :fCR. ^G prints the current filename; if it has been modified; the current line number; the number of lines in the file; and the location of the current line as a percent of the file length.
(4.4.3.2)

^H (BS) Same as **left arrow** (see **h**). During an insert, it eliminates the last input character, backing over it but not erasing it.
(4.4.3.4, 4.4.4.1, 4.4.4.2, 4.4.9.5)

^I (TAB) Not a command character.
When inserted it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces that represent the tab. The spacing of tab stops is controlled by the *tabstop* option.
(4.4.5.1, 4.4.7.2)

^J (LF) **Down arrow**. It moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include **j** and ^N.
(4.4.2.4, 4.4.3.3, 4.4.9.5)

^K Unused.

^L The ASCII form feed character, which causes the screen to be cleared and redrawn. It is useful after a transmission error, after output is stopped by an interrupt, or if characters from a program other than the editor have scrambled the screen.
(4.4.6.4, 4.4.9.2)

^M (CR) A carriage return advances to the first non-white position of the next line. Given a count, it advances that many lines.
During an insert, a CR causes the insert to continue onto another line.

^N **Down arrow**. It moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms

include **j** and **^J**.
(4.4.2.4, 4.4.3.3)

^O Unused.

^P **Up arrow**. It moves the cursor one line up. A synonym is **k**.
(4.4.2.4, 4.4.3.3)

^Q Not a command character.
In input mode, **^Q** quotes the next character, the same as **^V**, except that some teletype drivers do not echo **^Q** to the editor.
(4.4.9.5)

^R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hard-copy terminals in *open* mode, retypes the current line.
(4.4.6.4, 4.4.9.1, 4.4.9.2, 4.4.9.8)

^S Unused.
Some teletype drivers use **^S** to suspend output until **^Q** is invoked.
(4.4.9.5)

^T Not a command character.
During an insert, with *autoindent* set and at the beginning of the line, it inserts *shiftwidth* white space.
(4.4.7.2, 4.4.7.9)

^U Scrolls the screen up, inverting **^D** which scrolls down. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands. The previous scroll amount is common to both. On a dumb terminal, **^U** will often necessitate clearing and redrawing the screen further back in the file.
(4.4.3.1, 4.4.9.2, 4.4.9.5)

^V Not a command character.
In input mode, it quotes the next character so that it is possible to insert non-printing and special characters into the file.
(4.4.7.9, 4.4.9.5)

^W Not a command character.
During an insert, it backs up as **b** would in command mode; the deleted characters remain on the display (see **^H**).
(4.4.9.5)

^X Unused.

^Y Exposes one more line above the current screen, leaving the cursor where it is if possible. There is no mnemonic value for this key; however, it is next to **^U** (which scrolls up many lines) (Version 3 only).
(4.4.3.1)

^Z Unused.

^[(ESC) Cancels a partially formed command (such as a **z** when no following character has yet been given), terminates inputs on the last line (read by commands such as **:**, **/**, and **?**), and ends insertions of new text into the buffer. If an ESC is given when the editor is inactive, the editor rings the bell or flashes the screen. If you are confused about the operation the editor is working on, hit ESC to stop the operation.
(4.4.2.4, 4.4.4.1, 4.4.8.5)

^\ Unused.

^] Searches for the word that is after the cursor as a tag. It is equivalent to typing **:ta**, this word, and then a CR.
(4.4.9.3)

^↑ Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file that you specified if you got a **"No write since last change"** diagnostic and do not want to have to type the filename again. You have to do a **:w** before ^↑ will work in this case. If you do not wish to write the file you should do **:e! #**CR instead.
(4.4.7.2, 4.4.9.3)

^_ Unused.
Reserved as the command character for the Tektronix 4025 and 4027 terminals.

SPACE Same as **right arrow** (see **I**).
(4.4.2.4, 4.4.3.5)

! An operator, which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise, the count is passed on to the object after the **!**. Thus **2!}***fmt*CR reformats the next two paragraphs by running them through the program *fmt*.
(4.4.7.7, 4.4.9.3)
To read a file or the output of a command into the buffer use **:r**.
(4.4.9.3)
To simply execute a command use **:!**.
(4.4.6.2, 4.4.9.3)

" Precedes a named buffer specification. Named buffers **1-9** are used for saving deleted text; named buffers **a-z** are available for general use.
(4.4.5.3, 4.4.7.3)

The macro character that, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.
(4.4.7.2, 4.4.7.9)

$** Moves to the end of the current line. If the **:se list**CR command is used, then the end of each line will be shown by printing a **$ after the end of the displayed text in the line. When a count is used, the cursor advances to the end of the line following the count. For example, **2$** advances the cursor to the end of the following line.
(4.4.3.2, 4.4.5.1, 4.4.7.2, 4.4.9.1)

% Moves to the parenthesis (()) or brace ({}) that balances the parenthesis or brace at the current cursor position.
(4.4.7.6, 4.4.7.8)

& A synonym for **:&**CR, analogous to the **ex &** command.

' When followed by a ', the cursor returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter (**a-z**), it returns to the line that was

marked with this letter with an **m** command, at the first non-white character in the line. When used with an operator such as **d**, the operation takes place over complete lines; if you use ', the operation takes place from the exact marked place to the current cursor position within the line.
(4.4.6.3)

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a ., !, or ? that is followed by either the end of a line or by two spaces. Any number of closing characters (),], ", and ') may appear after the ., !, or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[). A count advances that many sentences.
(4.4.5.2, 4.4.7.8)

) Advances to the beginning of a sentence. A count repeats the effect. See (for the definition of a sentence.
(4.4.5.2, 4.4.7.8)

* Unused.

+ Same as CR when used as a command.
(4.4.3.3)

, Reverse of the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. Especially useful after hitting too many ; characters. A count repeats the search.

- Retreats to the previous line at the first non-white character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center.
(4.4.3.3)

. Repeats the last command that changed the buffer. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, a **3.** deletes three words.
(4.4.4.3, 4.4.7.3, 4.4.9.2, 4.4.9.4)

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The search begins when you hit CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress. The search may be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can force whole lines to be affected. To do this, give a pattern with a closing / and then an offset *+n* or *-n*.

To include the / character in the search string, you must escape it with a preceding \. An ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A **$** at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available (see paragraph 4.4.9.4). Unless you set **nomagic** in your **.exrc** file you will have to precede the characters ., [, *, and ~ in the search pattern with a \ to

disable their special meanings.
(4.4.3.2, 4.4.7.1, 4.4.9.4)

0 Moves to the first character on the current line.

1-9 Used to form numeric arguments to commands.
(4.4.3.3, 4.4.9.2)

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with a CR, and the command is then executed. You can return to where you were by hitting DEL or RUB if you hit : accidentally.
(4.4.7.1, 4.4.9.3)

; Repeats the last single character find that used **f**, **F**, **t**, or **T**. A count iterates the basic scan.
(4.4.5.1)

< An operator that shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, it affects lines when repeated, as in **<<**. Counts are passed through to the basic object, thus **3<<** shifts three lines.
(4.4.7.6, 4.4.9.4)

= Re-indents lines for LISP, as though they were typed in with *lisp* and *autoindent* set.
(4.4.7.8)

> An operator that shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in **>>**. Counts repeat the basic object.
(4.4.7.6, 4.4.9.4)

? Scans backwards, the opposite of **/**. See the **/** description for details on scanning.
(4.4.3.2, 4.4.7.1)

@ A macro character.
(4.4.7.9)
If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.
(4.4.4.1, 4.4.9.5)

A Appends at the end of line, a synonym for **$a**.
(4.4.9.5)

B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.
(4.4.3.4)

C Changes the rest of the text on the current line; a synonym for **c$**.
(4.4.4.4)

D Deletes the rest of the text on the current line; a synonym for **d$**.
(4.4.4.4)

E Moves forward to the end of a word, defined as blanks and nonblanks, like **B** and **W**. A count repeats the effect.

F Finds a single following character, backwards in the current line. A count repeats the search that many times.
(4.4.5.1)

G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary.
(4.4.3.2)

H **Home arrow.** Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count's line on the screen. In either case, the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected.
(4.4.3.3)

I Inserts at the beginning of a line; a synonym for **↑i.**

J Joins lines together, supplying appropriate white space: one space between words; two spaces after a .; and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default value of two.
(4.4.7.5, 4.4.9.1)

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a line count number, moves the cursor to the first non-white character of the indicated line from the bottom. Operators affect whole lines when used with **L.**
(4.4.3.3, 4.4.4.4)

M Moves the cursor to the middle line on the screen, at the first non-white position on the line.
(4.4.3.3)

N Scans for the next match of the last pattern given to / or **?**, but in the reverse direction; this is the reverse of **n.**

O Opens a new line above the current line and inputs text up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better.
(4.4.4.1)

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines; otherwise, the text is inserted between the characters before and at the cursor. The **P** character may be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1-9** contain deleted material, buffers **a-z** are available for general use.
(4.4.5.3, 4.4.7.3)

Q Quits from *vi* to *ex* command mode. To return to *vi*, you must enter a **:vi** command. Once in *ex*, the editor supplies the : as a prompt.
(4.4.9.7)

R Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.

S Changes whole lines; a synonym for **cc.** A count substitutes for that many lines. The lines are saved in the numeric buffers and erased on the screen before the substitution begins.
(4.4.4.4)

T Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.
(4.4.5.1)

U Restores the current line to its state before you started changing it.
(4.4.4.5)

V Unused.

W Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/nonblank characters. A count repeats the effect.
(4.4.3.4)

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.
(4.4.5.3)

ZZ Exits the editor (same as **:x**CR). If any changes have been made, the buffer is written out to the current file. Then the editor quits.
(4.4.2.6)

[[Backs up to the previous section boundary. A section begins at each macro in the *sections* options, normally a .NH or .SH and also at lines that start with a form feed **^L**. Lines beginning with { also stop **[[**; this makes it useful for looking backward, a function at a time, in C programs. If the option *lisp* is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects.
(4.4.5.2, 4.4.7.1, 4.4.7.6, 4.4.9.2)

**** Unused.

]] Moves forward to a section boundary, see **[[** for a definition.
(4.4.5.2, 4.4.7.1, 4.4.7.6, 4.4.9.2)

↑ Moves to the first nonwhite position on the current line. Also used in search strings to match a pattern at the beginning of a line.
(4.4.3.2, 4.4.5.1)

_ Unused

‘ When followed by a ‘ returns to the previous context. The previous context is set whenever the current line is moved in a nonrelative way. When followed by a letter **a-z**, returns to the position that was marked with this letter with an **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ', the operation takes place over complete lines.
(4.4.3.2, 4.4.6.3)

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC.
(4.4.4.1, 4.4.9.2)

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.
(4.4.3.4)

c An operator that changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text that is changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences.
(4.4.4.3, 4.4.9.4)

d An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w**.
(4.4.4.3, 4.4.4.4, 4.4.5.1, 4.4.9.4)

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect.
(4.4.3.4)

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find.
(4.4.5.1)

g Unused.

h **Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**^H**) has the same effect. On Version 2 editors, arrow keys on certain kinds of terminals (those that send escape sequence, such as vt52, c100, or hp) cannot be used. A count repeats the effect.
(4.4.2.4, 4.4.3.3, 4.4.3.4)

i Inserts text before the cursor, otherwise like **a**.
(4.4.4.1)

j **Down arrow**. Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include **^J** (line feed) and **^N**.
(4.4.2.4, 4.4.3.3)

k **Up arrow**. Moves the cursor one line up. **^P** is a synonym.
(4.4.2.4, 4.4.3.3)

l **Right arrow**. Moves the cursor one character to the right. SPACE is a synonym.
(4.4.2.4, 4.4.3.3, 4.4.3.4)

m Marks the current position of the cursor in the mark register that is specified by the next character **a-z**. Return to this position or use with an operator using **‘** or **’**.
(4.4.6.3)

n Repeats the last **/** or **?** scanning commands.
(4.4.3.2)

o Opens new lines below the current line; otherwise like **O**.
(4.4.4.1)

p Puts text after/below the cursor; otherwise like **P.**
(4.4.5.3, 4.4.7.3)

q Unused.

r Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above, which is usually the more useful iteration of **r.**
(4.4.4.2)

s Changes the single character under the cursor to the text that follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c.**
(4.4.4.2)

t Advances the cursor up to the character before the next character typed. Most useful with operator such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this does not delete enough the first time.
(4.4.5.1)

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states; thus it is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers.
(4.4.4.5)

v Unused.

w Advances to the beginning of the next word, as defined by **b.**
(4.4.3.4)

x Deletes the single character under the cursor. With a count, deletes that many characters forward from the cursor position, but only on the current line.
(4.4.4.2, 4.4.7.5)

y An operator that yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. Text can be recovered by a later **p** or **P.**
(4.4.5.3, 4.4.9.4)

z Redraws the screen with the current line placed as specified by the following character:

RETURN	specifies the top of the screen
.	specifies the center of the screen
-	specifies the bottom of the screen.

A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line.
(4.4.6.3, 4.4.7.1)

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally .IP, .LP, .PP, .QP and **.bp.** A paragraph also begins after a completely empty line, and at each section boundary (see **[[**).
(4.4.5.2, 4.4.7.6, 4.4.9.6)

	Places the cursor on the character in the column specified by the count. (4.4.9.1, 4.4.9.2)
}	Advances to the beginning of the next paragraph. See { for the definition of paragraph. (4.4.5.2, 4.4.7.6, 4.4.9.6)
~	Unused.
^? (DEL)	Interrupts the editor, returning it to command-accepting state. (4.4.9.5)

5. AN INTRODUCTION TO SHELL

5.1 General

The shell is a command programming language that provides an interface to the operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while**, **if then else**, **case**, and **for** are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically filenames or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The shell can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through pipes can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The shell is both a command language and a programming language that provides an interface to the SYSTEM V/68 operating system. This section describes, with examples, the SYSTEM V/68 shell. The "Simple Commands" section of this chapter covers most of the everyday requirements of terminal users. Some familiarity with SYSTEM V/68 is an advantage when reading this section. The section entitled "Shell Procedures" describes those features of the shell primarily intended for use within shell commands or procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be helpful when reading this section. The "Keyword Parameters" section describes the more advanced features of the shell. Refer to Table 5-1 for a defined listing of grammar words used in this section.

5.2 Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument —l tells *ls*(1) to print status information, size, and the creation date for each file.

5.2.1 Background Commands. To execute a command, the shell normally creates a new process and waits for it to finish. In addition, the shell can prompt for another command without waiting for the first to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file **pgm.c** . The trailing **&** instructs the shell not to wait for the command to finish. To help keep track of such a process, the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps*(1) command.

5.2.2 Input/Output Redirection. Most commands produce output to the "standard output" that is initially connected to the terminal. This output may be directed to a file by the notation > ; for example,

ls −l > *file*

The notation > *file* is interpreted by the shell and is not passed as an argument to *ls*(1). If *file* does not exist, the shell creates it; otherwise, the original contents of *file* are replaced with the output from *ls*(1). Output may be appended to a file using the notation ≫ as follows:

ls −l ≫ *file*

In this case, *file* is also created if it does not already exist.

The "standard input" of a command may be taken from a file instead of the terminal by the notation < ; for example,

wc < *file*

The command *wc*(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

wc −l < *file*

can be used.

5.2.3 Pipelines and Filters. The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, between commands as in

ls −l | wc

Two or more commands connected in this way constitute a "pipeline", and the overall effect is the same as

ls −l > *file* **; wc <** *file*

except that no *file* is used. Instead the two processes are connected by a pipe (refer to *pipe*(2)) and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting *wc*(1) when there is nothing to read and halting *ls*(1) when the pipe is full.

A "filter" is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*(1), selects from its input those lines that contain some specified string. For example,

ls | grep old

prints those lines, if any, of the output from **ls** that contain the string **old**. Another useful filter is *sort*(1). For example,

who | sort

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands; for example,

ls | grep old | wc −l

prints only the number of filenames in the current directory containing the string **old**.

5.2.4 Filename Generation. Many commands accept arguments which are filenames. For example,

ls –l main.c

prints only information relating to the file **main.c**. The **ls —l** command alone prints the same information about all files in the current directory.

The shell provides a mechanism for generating a list of filenames that match a pattern. For example,

ls —l *.c

generates as arguments to **ls** all filenames in the current directory that end in **.c**. The character * is a pattern that matches any string including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a minus matches any character lexically between the pair.

For example,

[a–z]*

matches all names in the current directory beginning with one of the letters **a** through **z**. The input

/usr/fred/test/?

matches all names in the directory **/usr/fred/test** that consist of a single character. If no filename is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

echo /usr/fred/*/core

finds and prints the names of all **core** files in subdirectories of **/usr/fred**. (The *echo*(1) command is a standard SYSTEM V/68 command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all subdirectories of **/usr/fred**.

There is one exception to the general rules given for patterns. The character . at the start of a filename must be explicitly matched. Therefore, the input

echo *

echoes all filenames in the current directory not beginning with the . character. The input

echo .*

echoes all filenames that begin with the . character. This avoids inadvertent matching of the names . and .. which mean "the current directory" and "the parent directory", respectively. (Notice that *ls*(1) suppresses information for the files . and .. .)

5.2.5 Quoting And Escaping. Characters that have a special meaning to the shell, such as

< > * ? | &

are called "metacharacters". A complete list of metacharacters is given in Table 5-2. Often it is necessary to conceal the special meaning that the shell associates with these characters. When any character is preceded by a \ (backslash), it is "escaped" and loses its special meaning. For example, the * carries a special meaning when read by the shell: "in a pattern, match any character, including the null character". The \ escapes the special meaning of the * in the command

ls -l *

so that the command attempts to list a file named *. Try this yourself. Now repeat the command without escaping the special meaning of *:

ls -l *

Standard output will display a long list of all files in the current working directory. As another example, the sequence \ **newline** escapes the special meaning of the newline (RETURN) character: "send command". Escaping the **newline** enables long strings of commands to be continued over more than one line. The \ is convenient for escaping single characters but when more than one character must be escaped, the \ mechanism is clumsy and prone to errors. Metacharacters that are included in a string of characters may be escaped by placing the string inside single acute accent characters (′). For example

ls -l mm′$*′**

searches for a file named **mm$***. Within single acute accent characters, all characters (except ′ itself) are taken literally with all special meanings ignored. Therefore,

stuff=′echo $? $*; ls * | wc′

results in the string

echo $? $*; ls * | wc

being assigned to the variable *stuff*.

A different result occurs when a character string is enclosed in single grave accent characters. The grave accents (`), sometimes called *back quotes*, signify a command substitution. Command substitutions are discussed later in this chapter and in more detail in *SYSTEM V/68 Programming Guide*, Chapter 2. To understand the practical difference, enter the command

echo ′pwd′

The output will echo **pwd**. Now enter the command

echo `pwd`

The output will now print the working directory.

Enclosing a character string within double quotes performs an escape function on most metacharacters but preserves the special meaning of a few. The characters that retain their special meaning to the shell are **$** (dollar sign: signifies parameter substitution), ` (grave accent: signifies command substitution), " (double quotes: signifies the end of the quoted string) and \ (backslash: escapes the special meanings just mentioned for **$** ` and "). Therefore, within double quotes, it is possible for command substitution to take place. To hide the special meaning of these characters within double quotes, precede each one with a \ (backslash).

In general, single metacharacters are most easily escaped by preceding each with a \ (backslash). If several metacharacters in a string must be escaped, enclose the string in single acute accent characters. Double quotes will hide the special meaning of some but not all

metacharacters. The special meanings for the **$** ` " and \ that are preserved within double quotes are valid only in shell commands. In other SYSTEM V/68 functions, such as text editing, these meanings do not apply. Specific details concerning the use of double quotes are described under paragraph "Evaluation and Quoting" later in this chapter.

5.2.6 The Shell and Login. Following the user's *login*(1), the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** (which is assumed to contain commands), the shell reads it immediately before reading any commands from the terminal.

5.2.7 Prompting by the Shell. When the shell is used from a terminal, it issues a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is **$** . A user may provide specific instructions to change the prompt by entering a special command in the file **.profile**. To do this, start in your home directory. Use *vi* to edit the file **.profile**.

vi .profile

Change the prompt by entering

PS1=*newprompt*

setting the prompt to be the string *newprompt*. If a newline is typed and further input is needed, the shell issues the prompt > . Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the shell to read another command. The other prompt (>) may be changed by entering

PS2=more

To see that the prompt is now changed, save **.profile** and log off SYSTEM V/68. Log on again. The shell will now read the new **.profile** and display the new prompt string.

5.2.8 Summary.

ls	Prints the names of files in the current directory.
ls > *file*	Puts the output from *ls*(1) into *file*.
ls | wc −l	Prints the number of files in the current directory.
ls | grep old	Prints those filenames containing the string **old**.
ls | grep old | wc −l	Prints the number of filenames containing the string **old**.
cc pgm.c &	Runs **cc** in the background.

5.3 Shell Procedures

The shell may be used to read and execute commands contained in a file. For example, the following call

sh *file* [*args* ...]

calls the shell to read commands from *file*. Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the "positional parameters" **$1, $2, ...** . For example, if the file **wg** contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All operating system files have three independent attributes (often called "permissions"): read (**r**), write (**w**), and execute (**x**). File permissions are changed according to mode. The operating system command *chmod*(1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file **wg** has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#** . The name of the file being executed is available as **$0** .

A special shell parameter **$*** is used to substitute for all positional parameters except **$0** . A typical use of this is to provide some default arguments, as in

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

5.3.1 Control Flow: for. A frequent use of shell procedures is to loop through the arguments (**$1, $2, ...**), executing commands once for each argument. An example of such a procedure is **tel**, which searches the file **/usr/lib/telnos**, which contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of **tel** is

```
for i
do
    grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in **/usr/lib/telnos** that contain the string **fred.**

The command

```
tel fred bert
```

prints those lines containing **fred** followed by those containing **bert.**

The **for** loop notation is recognized by the 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 a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. A *name* is a shell 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 positional parameter; that is, **in $*** is assumed.

Another example of the use of the **for** loop is the *create* command (*creat*(2)), whose text is

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

The command

```
create alpha beta
```

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

5.3.2 Control Flow: case. A multiple way (choice) branch is provided by the **case** notation. For example,

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

performs an append operation. (Note the use of semicolons to delimit the cases.) When called with one argument as in

append *file*

$# is the string **1,** and the standard input is appended (copied) onto the end of *file* using the *cat*(1) command.

append *file1 file2*

appends the contents of *file1* onto *file2.* If the number of arguments supplied to **append** is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

case *word* **in**
pattern) *command-list* ;;
...
esac

The 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, it can be used for the default case.

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 below, the commands following the second * will never be executed since the first * executes everything it receives.

```
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*(1) command.

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

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
    ?)...
```

matches the character ? .

5.3.3 Here Documents. The shell procedure **tel** described under "Control Flow: **for**" in this section uses the file **/usr/lib/telnos** to supply the data for *grep*(1). An alternative is to include this data within the shell procedure as a **here** document, as in

```
for i
do
  grep $i << !
  ...
  fred mh0123
  bert mh0789
  ...
!
done
```

In this example, the shell takes the lines between << ! and ! as the standard input for *grep*(1). The string ! is arbitrary. The document is terminated by a line that consists of the string following << .

Parameters are substituted in the document before it is made available to *grep*(1), as illustrated by the following procedure called **edg**.

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

The call

edg *string1 string2 file*

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented by using \ to quote the special character **$** as in

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

(This version of **edg** is equivalent to the first except that *ed*(1) will print a **?** if there are no occurrences of the string **$1** .) Substitution within a **here** document may be prevented entirely by quoting the terminating string; for example,

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

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

5.3.4 Shell Variables. The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

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

which assigns values to the variables *user*, *box*, and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

echoes **fred.**

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```

moves the *file* from the current directory to the directory **/usr/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 by a letter or digit. For example,

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

directs the output of *ps*(1) to the file **/tmp/psa**, whereas

```
ps a >$tmpa
```

causes the value of the variable *tmpa* to be substituted.

Except for **$?**, the following are set initially by the shell.

$? 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 nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). Used, for example, in an append operation to check the number of parameters.

$$ The process number of this shell (in decimal). Because process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$– The current shell flags, such as **–x** and **–v**.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively, the shell 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 shell prints the message **you have mail** before prompting for the next command. This variable is typically set in the file **.profile** in the user's login directory. For example,

MAIL=/usr/mail/fred

$HOME The default argument for the *cd*(1) command. The current directory is used to resolve filename references that do not begin with a / and is changed using the *cd* command. For example,

cd /usr/fred/bin

makes the current directory **/usr/fred/bin** . Then

cat wn

prints on the terminal the file **wn** in this directory. The command **cd** with no argument is equivalent to

cd $HOME

This variable is also typically set in the user's login **.profile.**

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

PATH=:/usr/fred/bin:/bin:/usr/bin

specifies that the current directory (the null string before the first :), **/usr/fred/bin**, **/bin**, and **/usr/bin** are to be searched in that order. In this way, individual users can have their own private commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used; a single attempt is made to execute the command.

$PS1 The primary shell prompt string; by default, **$** .

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

$IFS The set of characters used by "blank interpretation". (Refer to paragraph "Evaluation and Quoting" in the section entitled "Keyword Parameters".)

5.3.5 The test Command. The *test* command is intended for use by shell programs. For example,

test −**f** *file*

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are as follows (refer to *test*(1) for a complete specification).

test *s*	true if the argument *s* is not the null string
test —f *file*	true if *file* exists
test —r *file*	true if *file* is readable
test —w *file*	true if *file* is writable
test —d *file*	true if *file* is a directory

5.3.6 Control Flow: while. The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided with actions that are determined by the exit status returned by commands. A **while** loop has the general form

```
while command-list1
do
        command-list2
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-list1* is executed. If a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

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

is equivalent to

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

The *shift* command is a shell command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1** .

Another way of using 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 (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

5.3.7 Control Flow: if. Also available is a general conditional branch of the form

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

which tests the value returned by the last simple command following **if**.

The **if** command may 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
```

An example of the use of **if**, **case**, and **for** constructions is given in the paragraph entitled "The *man* Command" in this section.

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 as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

The *touch* command changes the "last modified" time for a list of files. The command may be used in conjunction with *make*(1) to force recompilation of a list of files. The following example illustrates the *touch* command:

```
flag=
for i
do
    case $i in
       -c)   flag=N ;;
       *)    if test -f $i
             then
                ln $i junk$$
              rm junk$$
             elif test $flag
             then
                echo file \ '$i\ ' does not exist
             else
                >$i
             fi ;;
    esac
done
```

The **-c** flag is used in this command to force creation of subsequent files if they do not already exist. Otherwise, if the files do not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **-c** argument is encountered. The commands

ln ... ; rm ...

make a link to the file and then remove it.

The sequence

if *command1*
then
command2
fi

may be written

command1 **&&** *command2*

Conversely,

command1 || *command2*

executes *command2* only if *command1* fails. In each case, the value returned is that of the last simple command executed.

5.3.7.1 Command Grouping. Commands may be grouped in two ways,

{ *command-list* ; }

and

(*command-list*)

The first form, *command-list*, is simply executed. 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**.

5.3.8 Debugging Shell Procedures. The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

set −v

(**v** for verbose) and causes lines of the procedure to be printed as they are read. It is useful for isolating syntax errors. It may be invoked without modifying the procedure by entering

sh −v *proc ...*

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the **−n** flag, which prevents execution of subsequent commands. (Note that typing **set −n** at a terminal renders the terminal useless until an end-of-file is typed.)

The command

set −x

produces an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the preceding commands at the terminal to see the effect they have.) Both flags may be turned off by typing

set −

and the current setting of the shell flags is available as **$−** .

5.3.9 The man Command. The *man*(1) command is used to print sections of the *SYSTEM V/68 User's Manual.* It is called by entering

man sh
man −t ed
man 2 fork

In the first call, the manual section for *sh* is printed. Since no section is specified, section 1 is used. The second call typesets (**−t** option) the manual section for *ed*. The last call prints the *fork* manual page from section 2 of the manual.

A version of the *man* command follows:

```
cd /usr/man
: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1
for i
do
    case $i in
        [1-9]*)   s=$i ;;
        -t)   N=t ;;
        -n)   N=n ;;
        -*)   echo unknown flag \'$i\' ;;
        *)    if test -f man$s/$i.$s
              then
                  ${N}roff man0/${N}aa man$s/$i.$s
              else
                  : 'look through all manual sections'
              found=no
              for j in 1 2 3 4 5 6 7 8 9
              do
                  if test -f man$j/$i.$j
                  then man $j $i
                    found=yes
                  fi
              done
              case $found in
                  no) echo '$i: manual page not found'
              esac
        fi ;;
    esac
done
```

5.4 Keyword Parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure 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 positional parameters **$1**, **$2**, ...

The **set** command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

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

5.4.1 Parameter Transmission. When a shell procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to

be exported. For example,

export user box

marks the variables *user* and *box* for export. When a shell procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without an 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 with *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.

5.4.2 Parameter Substitution. If a 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}

echoes nothing. A default string may be given as in

echo ${d-.}

which echoes 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-'*'}

echoes * if the variable *d* is not set. Similarly,

echo ${d-$1}

echoes 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* has not been set previously, it is set now to the string . (dot). (The notation **${...=...}** is not available for positional parameters.)

If there is no sensible default, the notation

echo ${d?*message*}

echoes the value of the variable *d* if it has one; otherwise, *message* is printed by the shell, and execution of the shell procedure is abandoned. If *message* is absent, a standard message is printed. A shell procedure that requires some parameters to be set might start as follows:

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

Colon (:) is a command built into the shell that does nothing after its arguments have been evaluated. If any of the variables *user*, *acct*, or *bin* are not set, the shell abandons execution of

the procedure.

5.4.3 Command Substitution. The standard output from a command can be substituted in a similar way to parameters using the grave accent marks (`). The command *pwd*(1) prints on its standard output the name of the current directory. For example, if the current directory is **/usr/fred/bin**, the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (`...`) is interpreted as the command to be executed and is replaced with the output from the command. Command substitutions are written following normal quoting conventions. For example,

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

is equivalent to

```
ls $1
```

where $1 retains its special meaning.

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 string-processing commands to be used within shell procedures. An example of such a command is *basename*(1), which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string **main**. Its use is illustrated by the following fragment from a *cc*(1) command

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

which sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

```
for i in `ls -t`; do ...
```

The variable *i* is set to the names of files ordered according to the time of last modification, with the most recent first.

```
set `date`; echo $6 $2 $3, $4
```

prints, for example,

```
1984 Jun 1, 23:59:59
```

5.4.4 Evaluation and Quoting. The 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 listed in Table 5-1. Before a command is executed, the following substitutions occur:

a. Parameter substitution; e.g., **$user**

b. Command substitution; e.g., **`pwd`**

   Only one evaluation occurs so that if, for example, the value of the variable *X* is the string **$y** then

   echo $X

   echoes **$y**.

c. Blank interpretation

   Following the above substitutions, the resulting characters are broken into nonblank 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 ''

   passes on the null string as the first argument to **echo**, whereas

   echo $null

   calls **echo** with no arguments if the variable *null* is not set or is set to the null string.

d. 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 substitution occurs in the word used for a **case** branch.

Evaluations conform to the escape and quoting mechanisms described earlier i.e., \(backslash), '...'(acute accent), and " (double quotes). Within double quotes, parameter and command substitution occur, but filename generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	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 positional parameters as a single argument and is equivalent to

echo " $1 $2 ... "

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

echo " $@ "

passes the positional parameters, unevaluated, to **echo** and is equivalent to

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

The following illustration details how the shell evaluates metacharacters located in a string enclosed by acute accents, grave accents and double quotes.

metacharacter

	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t = terminator
y = interpreted
n = not interpreted

When more than one evaluation of a string is required, the built-in command *eval* may be used. For example, if the variable *X* has the value **$y** and if *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 shell. The input is read and the resulting command(s) executed. For example,

wg='eval who|grep'
$wg fred

is equivalent to

who|grep fred

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

5.4.5 Error Handling. The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gtty*(2)). A shell invoked with the **–i** flag is also interactive.

Execution of a command (refer to the "Command Execution" paragraph of this section) may fail for any of the following reasons:

a. Input/output redirection may fail; for example, if a file does not exist or cannot be created.

b. The command itself does not exist or cannot be executed.

c. The command terminates abnormally, for example, with a bus error or memory fault signal.

d. The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell goes on to execute the next command. Except for the last case, an error message is printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell returns to read another command from the terminal. Such errors include the following:

a. Syntax errors; e.g., **if ...then... done**.

b. A signal such as SIGINT. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.

c. Failure of any of the built-in commands such as *cd*(1).

The shell flag **−e** causes the shell to terminate if any error is detected. The following is a list of the SYSTEM V/68 signals (refer to *signal*(2)):

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	kill (cannot be caught 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*(1))
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child
SIGPWR	19	power fail

The operating system signals marked with an asterisk (*) in the list produce a core dump if not caught. However, the shell itself ignores SIGQUIT, which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14, and 15.

5.4.6 Fault Handling. Shell procedures 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); if this signal is received, it will execute the following commands:

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

The **exit** is another built-in command that terminates execution of a shell procedure. The **exit** is required; otherwise, after the trap has been taken, the shell resumes executing the procedure at the place where it was interrupted.

SYSTEM V/68 signals can be handled in one of three ways.

a. They can be ignored, in which case the signal is never sent to the process.

b. They can be caught, in which case the process must decide what action to take when the signal is received.

c. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the shell procedure; for example, by invoking it in the background (refer to the paragraph "Command Execution"); *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by the following modified version of the *touch*(1) command:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
  case $i in
  -c) flag=N ;;
  *) if test -f $i
     then
          ln $i junk$$; rm junk$$
     elif test $flag
     then
          echo file \ '$i\ ' does not exist
     else
          >$i
     fi ;;
  esac
done
```

The cleanup action is to remove the file **junk$$**. 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 the SYSTEM V/68 operating system, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to *trap*. The following:

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

is a fragment taken from the *nohup*(1) command which causes the operating system hangup, interrupt, quit, and software termination signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
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** procedure is an example of the use of *trap* where there is no exit in the *trap* command. The **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. Although interrupts are ignored when the requested commands are executing, they cause termination when **scan** is waiting for input. The **scan** procedure follows:

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

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

5.4.7 Command Execution. To run a command (other than a built-in command), the shell first creates a new process using the system call *fork*(2). 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 rare cases when no fork is required and simply replaces the 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 signals specified 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. Some input/output specifications are as follows:

> *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 shell input that follow up to, but do not include, a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to escape the characters \, **$**, `, and the first character of *word*. In the latter case, **\newline** is ignored (e.g., 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*. Another example,

... 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. Unpredictable results would occur otherwise. 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 that the quit and interrupt signals are turned off so the command ignores them. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the SYSTEM V/68 convention is that if a signal is set to 1 (ignored) then it is never changed, even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

5.4.8 Invoking the Shell. The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, commands are read from the file **.profile** .

−c *string*	If the **−c** flag is present, then commands are read from *string* .
−s	If the **−s** flag is present or if no arguments remain, commands are read from the standard input. Shell output is written to file descriptor 2.
−i	If the **−i** flag is present or if the shell input and output are attached to a terminal (as told by *getty*(8)), this shell is "interactive". In this case, SIGTERM is ignored (so that **kill 0** does not kill an interactive shell) and SIGINT is caught and ignored (so that *wait* is interruptible). In all cases, SIGQUIT is ignored by the shell.

Table 5-1. 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 nonblank characters
name	a sequence of letters, digits, or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

Table 5-2. Metacharacters and Reserved Words

(a) *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

(b) *patterns:*

*	match any character(s) including none

?	match any single character
[...]	match any of the enclosed characters

(c) *substitution:*

${...}	substitute shell variable
`...`	substitute command output

(d) *quoting:*

\	quote the next character
'...'	quote the enclosed characters except for the '
"..."	quote the enclosed characters except for the **$**, ` ,\ , and "

(e) *reserved words:*

if then else elif fi
case in esac
for while until do done
{ } [ ] test

NOTES

6. SOURCE CODE CONTROL SYSTEM (SCCS)

6.1 General

The Source Code Control System (SCCS) is a collection of SYSTEM V/68 commands that monitors changes to text files and creates an audit trail for each change. The source code and software system documentation are examples of text files for which users would want to monitor changes. The SCCS performs like a file custodian under SYSTEM V/68. The SCCS provides facilities for the following:

a. Stores files of text

b. Retrieves particular versions of the files

c. Controls updating privileges to files

d. Identifies the version of a retrieved file

e. For each change to each file, records the location, reason, time and identifies the user who made the change.

When programs and documentation undergo frequent changes because of maintenance and/or enhancement, backup procedures include keeping a version of each program or document as it existed before changes were applied. Keeping copies (on paper or other media) becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution by storing the original file on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This section, together with relevant portions of the *SYSTEM V/68 User's Manual* forms a complete user's guide to SCCS. The following topics are covered:

a. Creating an SCCS file, retrieving a version of the file and making changes to the file.

b. Tracing changes throughout a file using SCCS identification numbers.

c. SCCS commands conventions and rules.

d. Applications of SCCS commands.

e. Protecting, formatting, auditing and administering SCCS files.

SCCS installation and implementation are not described in this section.

6.2 SCCS For Beginners

The easiest way to understand SCCS is to use it. The paragraphs that follow assume the reader knows how to log on to the operating system, create files and use a text editor. Try the examples given below. To supplement the material in this section, consult the detailed SCCS command descriptions in the *SYSTEM V/68 User's Manual*.

6.2.1 Terminology. Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file. Each set of changes, called a "delta," usually builds on all previous sets. When a file edit is completed, a *delta* command will write the changes or delta back to the file. Each delta is assigned an identification number so that users can refer to particular versions of a file. Deltas are referred to by an *S*CCS *ID*entification string or SID. The SID is generally composed of two components, the "release" number and the "level" number, which are separated by a period. The first delta to an original file is called "1.1", the second "1.2", the third "1.3", etc. The release number can be changed as well, for example, deltas "2.1", "3.19", etc. A change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by building each set of changes (deltas 1.1, 1.2, etc., up to and including delta 1.5 itself) into the original SCCS file.

6.2.2 Creating an SCCS File via admin. Consider a file called **lang** that contains a list of programming languages:

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

Custody of the **lang** file can be given to SCCS. The following *admin*(1) command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*:

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

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

The *admin* command replies

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

This is a warning message that can be ignored for the purposes of this section. Its significance is described under the *get*(1) command in the section "ID Keywords." In the following examples, this warning message is not shown, although it may actually be issued by the various commands.

The file **lang** should now be removed so that all work involving the file will be monitored through SCCS:

```
rm lang
```

6.2.3 Retrieving a File via get. The removed file **lang** can be easily reconstructed with the following *get* command:

```
get s.lang
```

The command retrieves a copy of the latest version of file **s.lang** and prints the following messages:

```
1.1
5 lines
```

This means that *get* created a file that is a copy of **s.lang**, version 1.1 and contains five lines of text. The name of the new file is formed by deleting the "**s.**" prefix from the name of the SCCS file; hence, file **lang** is recreated.

The "**get s.lang**" command creates the file **lang** in read-only mode but keeps no information regarding its creation. If you want to *get* a file for editing, your *get* command must announce your intention to do so. This is done as follows:

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

The **-e** causes *get* to create a file **lang** with both read and write permissions (so it may be edited) and creates an edit information file called a *p-file*. The *p-file* contains one or two lines of information including the SID of the created version, the SID for the up-coming delta, the

editor's ID and the time the **get −e** command was executed. The *get* command to edit prints the same messages as before with the addition that the SID of the up-coming delta is also issued. For example:

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

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

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

6.2.4 Recording Changes via delta. To record changes within the SCCS file, **lang**, execute the following command:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

Enter a brief explanation of the changes. For example:

```
comments? added more languages
```

The *delta* command now determines what changes were made to the file **lang** by applying the *diff*(1) command to the original version and the edited version. Next, the *delta* command reads the *p-file* and incorporates the information into the file as part of its audit trail.

When the changes to **lang** have been stored in **s.lang**, *delta* outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

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

6.2.5 Additional Information About get. As shown in the previous example, the command

```
get s.lang
```

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

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** are SIDs. Omitting the level number of the SID (as in "**get -r1 s.lang**") defaults to the highest level number within the specified release. The second command requests the latest version in release 1, namely 1.2. The third command specifically requests a particular version, in this case, also 1.2.

A major change to a file is usually indicated by changing the release number (first component of the SID) of the delta. Automatic numbering of deltas proceeds by incrementing the level number (second component of the SID); therefore, the user must announce to SCCS the need to change the release number. This is done through the *get* command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2. In addition, the *get* program interprets the command as a request to change the delta release number to 2, causing the delta to be named 2.1, rather than 1.3. (There is no 0 level; all releases begin with level 1.) The *get* command will store the request to change the delta release number in the *p-file* where it will be read and carried out by the *delta* command when all edits are completed. The *get* command outputs

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

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

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

and *delta* executed:

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

the user will see *delta*'s version 2.1 is created:

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

Deltas will now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created.

6.2.6 The help Command. If the command:

```
get abc
```

is executed, the following message will be output:

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

The string "**co1**" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the *help*(1) command:

help co1

This produces the following output:

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

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

6.3 Delta Numbering

Deltas applied to an SCCS file can be thought of as the nodes of a tree; the tree root is the initial version of the file. The initial version of the file is normally named "1.1" and succeeding deltas or nodes are named "1.2", "1.3", etc. The first two components of the deltas' names are called the "release" and the "level" numbers, respectively. Normally, deltas are named by automatically incrementing the level number whenever a delta is made. Occasionally, a user will change the delta release number to indicate a major change. The new release number applies to all successor deltas until it is specifically changed. The evolution of any particular file may be mapped into a diagram referred to as an SCCS "tree". One example of an SCCS "tree" is represented in Figure 6.1.

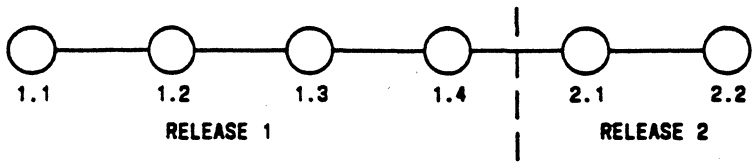

Figure 6-1. Evolution of an SCCS File

A progression of file versions in which each delta incorporates all the changes that preceded it is referred to as a "trunk" on an SCCS tree. An example of a "trunk" is illustrated in Figure 6.1.

Sometimes, it is necessary to cause a branching in the tree; that is, to create a new file version that incorporates only a portion of the changes to date. A branch on a SCCS tree is a delta that does **not** include all the changes that have been made to the original file. To explain why a branch might be desirable, consider a program in production use at version 1.3 for which release 2 development work is already in progress. Release 2 may already have deltas as shown in Figure 6.1., that is, deltas that are all dependent upon all the changes made to date. Assume that a production user reports a problem in version 1.3 and the user cannot

wait for release 2 for the problem to be fixed. Necessary changes to solve the problem must be written as a delta to version 1.3, the production version. This delta, which is released to the user, branches off version 1.3 and does not affect the deltas being written for release 2 (e.g., deltas 1.4, 2.1, 2.2, etc.).

To distinguish between deltas that lie along the trunk of the SCCS tree and deltas that branch away from the trunk, branch deltas are given names consisting of four components: a release and a level number, the same as a trunk delta; as well as an additional "branch" and a "sequence" number. Any branch delta name will appear as:

release.level.branch.sequence

The first branch from a particular trunk is branch 1, the next one 2, and so on. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 6.2.

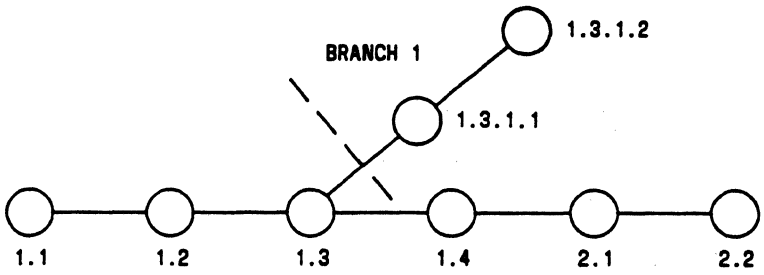

Figure 6-2. Tree Structure with Branch Deltas

Two observations about naming deltas are important. First, a branch delta may always be identified as such from its name. The names of trunk deltas always contain two components and the names of branch deltas always contain four components. Second, the first two components of a branch delta always specify the ancestral trunk delta. The next two component numbers are location independent; the branch and sequence numbers are assigned according to the order in which the deltas were created. Therefore, although the branch delta's name always identifies its ancestral trunk, it is impossible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch (1.3.1.n) has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Figure 6.3). The delta name 1.3.2.2 only defines it as the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. It is **not** possible to know from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

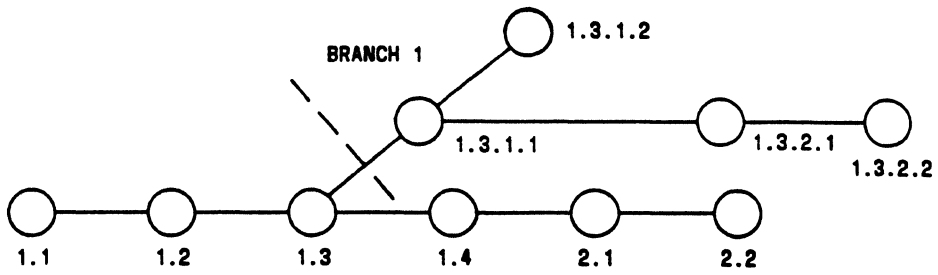

Figure 6-3. Extending the Branching Concept

Clearly, branch deltas can create extremely complex tree structures. Try to keep an SCCS tree as simple as possible. A tree's structure becomes difficult to follow as the number of branches increases.

6.4 SCCS Command Conventions

This section discusses conventions and rules that apply to SCCS commands, with exceptions indicated. The SCCS commands accept two types of arguments:

- keyletter arguments
- file arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (–), followed by a lowercase alphabetic character. In some cases, a value follows the letter. Keyletters control the execution of the given command.

File arguments (which may be names of files and/or directories) specify the file(s) to be processed. Naming a directory is equivalent to naming **all** of the SCCS files within that directory. Non-SCCS files and files that cannot be read because of permission modes are silently ignored.

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

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the *help*(1), *what*(1), *sccsdiff*(1), and *val*(1) commands.

Some SCCS commands are affected by flags appearing in SCCS files. Some of these flags are discussed in this section. For a complete description of all such flags, see the *admin*(1) section in the *SYSTEM V/68 User's Manual*.

When considering the actions of SCCS commands, draw a distinction between the real user [see *passwd*(1)] and the effective user of SYSTEM V/68. For now, assume that both the real user and the effective user are one and the same (i.e., the person who is logged onto SYSTEM V/68). This subject is discussed further in section "SCCS Files."

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

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

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

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

The code in parentheses may be used as an argument to the *help*(1) command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

6.5 SCCS Commands

This section describes the major features of SCCS commands. Detailed descriptions of the commands and their arguments are given in the *SYSTEM V/68 User's Manual*. The discussion below covers only the more common arguments.

The following is a summary of SCCS commands and their major functions, in approximate order of importance.

get(1)	Retrieves versions of SCCS files.
delta(1)	Incorporates changes to the text of SCCS files, i.e., creates new versions.
admin(1)	Creates SCCS files and makes changes to parameters of SCCS files.
prs(1)	Prints portions of an SCCS file in user specified format.
help(1)	Explains diagnostic messages.
rmdel(1)	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc(1)	Changes the commentary associated with a delta.
what(1)	Searches any SYSTEM V/68 file(s) for all occurrences of a special pattern and prints out what follows; it finds identifying information inserted by the *get* command.

sccsdiff(1)	Shows the differences between any two versions of an SCCS file.
comb(1)	Combines two or more consecutive deltas of an SCCS file into a single delta.
val(1)	Validates an SCCS file.

6.5.1 The get Command. The *get*(1) command creates a text file that contains a copy of a particular version of an SCCS file. The created text file, called a *g-file*, is created in the current directory and is owned by the real user. The mode assigned to the *g-file* and the particular SCCS version that was copied depends on the keyletters used to invoke the *get* command. The *g-file* name is formed by removing the "s" from the SCCS filename.

A common invocation of *get* is:

get s.abc

which normally retrieves the latest version of the SCCS file. Standard output might read:

1.3
67 lines
No id keywords (cm7)

which indicates:

a. Version 1.3 of file "**s.abc**" was retrieved (1.3 is the most recent trunk delta).

b. This version has 67 lines of text.

c. No ID keywords were substituted in the file. (For information about ID keywords, refer to section "ID Keywords.")

The generated *g-file* (file "**abc**") is given mode 444 (read-only) and may be used for inspection or compilation, not for editing.

If several file- or directory-name arguments are given in a *get* command, the SCCS filename precedes the information given for each file. For example:

get s.abc s.def

produces:

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

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

6.5.1.1 ID Keywords. Often, a user will want to keep records within a *g-file* so that historic information will appear in the load module when it is eventually created. For example, a user may want to record the date and time of creation, the version retrieved and the module's name. To do this, the SCCS provides 20 possible identification (ID) keywords which may be imbedded within a file. Each time an SCCS file is copied, the ID keywords within that file are replaced with the appropriate value. The format of an ID keyword is an uppercase letter enclosed by percent signs, (%). Twenty letters have been assigned specific meanings as ID keywords. For example, **%M%** is defined as the ID keyword that will be

replaced by the SID of the retrieved version of a file. (There is no space between the percent and the letter.) Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%F%** is defined as the SCCS filename. Thus, executing a *get* command on an SCCS file that contains an information line:

%M% %H% %F%

might give the following:

1.12 7/14/84 /systemv68/user_guide/s.sccs01

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

No id keywords (cm7)

This message is normally treated as a warning by *get*, although the presence of the **i** flag in the SCCS file will cause it to be treated as an error. For a complete list of the letters that have assigned ID keyword meanings, see *get*(1) in the *SYSTEM V/68 User's Manual*.

6.5.1.2 Retrieval of Different Versions. Keyletters allow different versions of SCCS files to be retrieved. In processing, the default version of the SCCS file is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of *get*.

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

get −r1.3 s.abc

retrieves version 1.3 of file **s.abc** and produces on the standard output:

1.3
64 lines

A branch delta may be retrieved similarly:

get −r1.5.2.3 s.abc

which produces on the standard output:

1.5.2.3
234 lines

When a 2- or 4-component SID is specified as a value for the **−r** keyletter and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

get −r3 s.abc

will retrieve the trunk delta with the highest level number within the given release. Thus, the above command might output:

3.7
213 lines

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

get −r9 s.abc

might produce:

7.6
420 lines

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

get −r4.3.2 s.abc

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

4.3.2.8
89 lines

The **−t** keyletter is used to retrieve the most recently created (top) version in a particular release. The top version of the SCCSW file tree is location independent. Thus, if the most recent delta in release 3 is 3.5,

get −r3 −t s.abc

might produce:

3.5
59 lines

However, if branch delta 3.2.1.5 was created after delta 3.5, the command would produce:

3.2.1.5
46 lines

6.5.1.3 Retrieval And Editing. Specification of the **−e** keyletter to the *get* command indicates you intend to edit the file. As part of the SCCS, use of this keyletter is restricted. Using the **−e** keyletter causes *get* to perform the following checks:

a. The user list (a list of login names and/or group IDs of users allowed to make deltas) is checked to ensure the user has editing permission. A null (empty) user list is read as containing all possible login names.

b. The release of the version being retrieved is checked to determine if it is a protected release. (This information is specified via flags in the SCCS file. Refer to section "SCCS File Protections.")

c. The release is not locked against editing. (The "lock" is specified as a flag in the SCCS file.)

d. Whether or not multiple concurrent edits are allowed for the SCCS file. (This information is specified by the **j** flag in the SCCS file.)

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

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

ID keywords appearing in the *g-file* are not substituted by *get* when the **−e** keyletter is specified. Replacement of ID keywords would cause them to be permanently changed within

the SCCS file following the execution of a *delta* command. Because *get* does not check for the presence of ID keywords within the *g-file*, the message

No id keywords (cm7)

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

In any case, the **–e** keyletter creates a *p-file* which is used to pass information to the *delta*(1) command.

The command:

get –e s.abc

produces on the standard output:

1.3
new delta 1.4
67 lines

The **–r** and/or **–t** keyletters can be used with the **–e** keyletter, to specify a particular version to be retrieved for editing.

The keyletters **–i** and **–x** can be used to specify a list of deltas to be included and/or excluded, respectively, by *get*. (Refer to *get*(1) in the *SYSTEM V/68 User's Manual* for the syntax of such a list.) "Including a delta" means forcing the changes that constitute the particular delta to be included in the retrieved version. Use this if you want to apply the same changes to more than one version of the SCCS file. "Excluding a delta" means forcing the delta to be ignored. Use this to undo the effects of a previous delta in the version of the SCCS file to be created. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. For example, two deltas can interfere when each one changes the same line of the retrieved *g-file*. Interference is indicated by a warning that shows the range of lines within the retrieved *g-file*. The user is expected to examine the *g-file* to determine if a problem actually exists and to take whatever corrective measures are necessary.

Warning: The –i and –x keyletters should be used with extreme care.

The **–k** keyletter facilitates regeneration of a *g-file* that may have been accidentally removed or ruined. The –k keyletter will also generate a special *g-file* that suppresses the replacement of ID keywords. Thus, a *g-file* generated by the **–k** keyletter is identical to one produced by *get* executed with the **–e** keyletter except that no processing related to the *p-file* takes place.

6.5.1.4 Concurrent Edits of Different SIDs. The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. That is, a number of *get* commands with the **–e** keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* created by the **get -e** command is named by replacing the **"s."** in the SCCS filename with **"p."**. The file, created in the directory containing the SCCS file, has permission mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. For each delta that is still in progress, the *p-file* contains:

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.

- The login name of the real user executing *get*.

Subsequent executions of **get -e** update the *p-file* with a line containing the above information. Before updating *get* reads the *p-file* to ensure that the specified SID is not currently being edited (unless multiple concurrent edits are allowed.)

If the check fails, an error message results. Therefore, different executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed because subsequent executions would attempt to overwrite a writable *g-file*, an SCCS error condition. (Section "SCCS File Protections" explains how different users may use SCCS commands on the same files.)

Table 6.1, which is located at the end of this chapter, shows the SID version to be retrieved by a *get* command, and the SID of the version to be eventually created by *delta*, as functions of the SID specified to *get*.

6.5.1.5 Concurrent Edits of Same SID. Under normal conditions, *get* editing commands based on the same SID are not permitted to occur concurrently. That is, a *delta* must be executed for each *g-file* before a second edit command is allowed. However, multiple concurrent edits (two or more successive executions of *get* for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. For example, if we assume the **j** flag is set in the SCCS file:

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

may be immediately followed by:

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

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

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

get -p s.abc > *arbitrary-filename*

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

send MOD=s.abc REL=3 compile

given that file **compile** contains:

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

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

The **-s** keyletter suppresses all output that is normally directed to the standard output. Messages to the diagnostic output are not affected.. This keyletter is used most often to prevent nondiagnostic messages from appearing on the user's terminal in conjunction with the **-p** keyletter to "pipe" the output of *get*, as in:

```
get -p -s s.abc | nroff
```

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

```
get -g -r4.3 s.abc
```

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

```
get -e -g s.abc
```

The **-l** keyletter creates an *l-file*, which describes the deltas used to construct a particular version of the SCCS file. The *l-file* is named by replacing the "**s.**" of the SCCS filename with "**l.**". This file is created in the current directory with mode 444, read-only, and is owned by the real user. (The table format is described in *get*(1) in the *SYSTEM V/68 User's Manual*.) For example:

```
get -r2.3 -l s.abc
```

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

```
get -lp -r2.3 s.abc
```

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

The **-m** keyletter is used to identify changes applied to an SCCS file, line by line.. As a result, a *g-file* is created in which every line is preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

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

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

When both the **−m** and **−n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the **%F%** ID keyword and a tab as a result of the **−n** keyletter. Second, each line in the generated file is followed by the line produced by the **−m** keyletter. Because the **−m** keyletter and/or the **−n** keyletter cause the *g-file* to be modified, the *g-file* must **not** be used for creating a delta. Therefore, neither the **−m** keyletter nor the **−n** keyletter may be used with the **−e** keyletter.

Refer to *get*(1) in the *SYSTEM V/68 User's Manual* for a full description of additional *get* keyletters.

6.5.2 The delta Command. The *delta*(1) command incorporates changes made to a *g-file* into the corresponding SCCS file; it creates a delta, creating a new version of the file.

The *delta* command will not execute without an existing *p-file*. The *delta* command examines the *p-file* to verify that an entry containing the user's login name exists. If none is found, an error message results. The *p-file* entry is required because the user who retrieved the *g-file* must be the one who will create the delta. If the login name of the user appears in more than one entry (i.e., the same user executed *get* with the **−e** keyletter more than once on the same SCCS file), the **−r** keyletter must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

The *delta* command performs the same permission checks that *get*(1) performs when invoked with the **−e** keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file* by comparing it with its own temporary copy of the unedited *g-file*. This temporary copy of the unedited *g-file* is called the *d-file* and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

In practice, the common invocation of *delta* is

delta s.abc

If work is being done on a terminal, a prompt

comments?

displays. The user replies with a description of why the delta is being made and terminates the reply with a newline character. The user's response can be 512 characters long. Newlines that are not intended to terminate the response must be escaped by \.

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

MRs?

Input that follows is read for Modification Request (MR) numbers, separated by blanks and/or tabs, and terminated with a newline character. In a tightly controlled environment, deltas may be created only as a result of some trouble report, change request, trouble ticket, etc. SCCS makes it possible to record such MR number(s) within each delta.

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

delta −y*descriptive comment* **−m***mrnum1 mrnum2* **s.abc**

In this case, prompts are not printed, and the standard input is **not** read. The **−m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when *delta* is executed from within a shell procedure. (Refer to *sh*(1) in the *SYSTEM V/68 User's Manual*.)

Commentary, whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation

of *delta*. This implies that when *delta* is invoked with more than one file argument and the first file named has a **v** flag, all files named must have a **v** flag. Similarly, if the first file named does not have a **v** flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

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

1.4
14 inserted
7 deleted
345 unchanged

Sometimes, the counts of lines reported as inserted, deleted, or unchanged by *delta* disagree with the user's perception. Usually, there are a number of ways to describe a set of changes, especially if lines are moved around in the *g-file*, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If *delta* finds no ID keywords in the edited *g-file*, the message

No id keywords (cm7)

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. One reason for this could be that the delta was created from a *g-file* that was itself created by a *get* without the **−e** keyletter. Remember that ID keywords are replaced by *get* in that case. A second reason for this message could be that the ID keywords were accidentally deleted or changed during editing. Another possibility is that the file never had any ID keywords. Whatever the reason for the message, the delta is created unless the **i** flag in the SCCS file is set. However, it is left up to the user to determine what remedial action, if any, is necessary. When the **i** flag is set, the system will treate the message as a fatal error and processing terminates without creating the delta.

After file processing is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy, the *q-file*, which is used in the same way as the *x-file* described in section 6.4. If there is only one entry in the *p-file*, then the *p-file* itself is removed.

Delta removes the edited *g-file* unless the **−n** keyletter is specified. Thus:

delta −n s.abc

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

The **−s** (silent) keyletter suppresses all output, other than the prompts **"comments?"** and **"MRs?"**, that is normally directed to the standard output. Use of the **−s** keyletter together with the **−y** keyletter (and possibly, the **−m** keyletter) will stop *delta* from reading the standard input and writing to the standard output.

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

6.5.3 The admin Command. The *admin*(1) command is used to administer SCCS files; that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters will detect and correct "corrupted" SCCS files. (These keyletters are discussed in section "SCCS File Auditing.") Newly created SCCS files are given read-only permission, mode 444, and are owned by the effective user. Only a user with write permission in the directory containing a particular SCCS file may use the *admin* command with that file.

6.5.3.1 Creation of SCCS Files. An SCCS file is created by executing the command

admin −ifirst s.abc

in which the value "**first**" of the **−i** keyletter is the file name of the initial delta for the SCCS file, **s.abc**. If no value is given for the **−i** keyletter, *admin* will read the standard input for the text of the initial delta. The command

admin −is.abc < first

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

No id keywords (cm7)

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

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1".

6.5.3.2 Inserting Commentary for the Initial Delta. When an SCCS file is created, the user may state the reason why. Supplying comments (**−y** keyletter) and/or MR numbers (**−m** keyletter) is accomplished here in exactly the same manner as for *delta*. Creating an SCCS file may sometimes result from an MR. If comments (**−y** keyletter) are omitted, a comment line of the form

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

is automatically generated.

If MR numbers (**−m** keyletter) are supplied, the **v** flag must also be set (using the **−f** keyletter described below). The **v** flag determines if MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" in the SCCS file. (Refer to *sccsfile*(1) in the *SYSTEM V/68 System User's Manual*.) In the example:

admin −ifirst −mmrnum1 −fv s.abc

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

6.5.3.3 Initialization and Modification of SCCS File Parameters. The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the **−t** keyletter. Descriptive text is intended as a summary of the contents and purpose of the SCCS file.

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

admin −ifirst −tdesc s.abc

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

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

admin −tdesc s.abc

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

admin −ts.abc

removes the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted with the **−f** and **−d** keyletters, respectively. The flags direct certain actions taken by SCCS commands. (Refer to *admin*(1) in the *SYSTEM V/68 User's Manual* for a description of all the flags.) For example, the **i** flag specifies that the warning message **"no ID keywords"** contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The **−f** keyletter is used to set a flag and, possibly, to set its value. For example:

admin −ifirst −fi −fmmodname s.abc

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

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

admin −dm s.abc

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

The SCCS files contain a "user list" of login names and/or group IDs of users who are allowed to create deltas. If this list is empty (default value), it implies that anyone may create deltas. To add login names and/or group IDs to the list, the **−a** keyletter is used. For example:

admin −axyz −awql −a1234 s.abc

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

6.5.4 The prs Command. The *prs*(1) command is used to print on the standard output all or part of an SCCS file in a format supplied by the user via the **−d** keyletter. The user specified format, called the output "data specification", is a string consisting of SCCS file data keywords interspersed with optional user text.

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

:I:

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS filename currently being processed, and **:C:** is defined as the comment line associated with a specified delta. Each part of an SCCS file has an associated data keyword. For a complete list of the approximately 50 available data keywords, refer to *prs*(1) in the *SYSTEM V/68 User's Manual*.

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

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

may produce on the standard output

2.1 this is the top delta for s.abc 2.1

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

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

may produce the following output:

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

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

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

prs –d:I: –r1.4 –e s.abc

may output

1.4
1.3
1.2.1.1
1.2
1.1

and the command

prs –d:I: –r1.4 –l s.abc

may produce

3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4

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

6.5.5 The help Command. The *help*(1) command prints explanations of SCCS commands and command messages. Arguments to *help* can be the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Multiple arguments to *help* can be processed and each argument is processed independently. An error resulting from one argument will not terminate the processing of the other arguments.

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

```
help ge5 rmdel
```

produces

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

rmdel:
rmdel -rSID name ...
```

6.5.6 The rmdel Command. The *rmdel*(1) command removes a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 6.3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, deltas 1.3.2.1 and 2.1 can be removed, etc.

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

The **-r** keyletter is mandatory and is used to specify the complete SID of the delta to be removed. That is, the SID must have two components for a trunk delta and four components for a branch delta. For example:

```
rmdel -r2.3 s.abc
```

specifies the removal of trunk delta **2.3** from the SCCS file. Before removing the delta, *rmdel* checks that the release being accessed is not a protected release (refer to section "SCCS File Protections"). The *rmdel* command also checks that the SID version specified is not already being edited by another user. In addition, the login name or group ID of the user must appear in the file's user list, or the user list must be empty. Finally, the release specified cannot be locked against editing. (Refer to *admin*(1) in the *SYSTEM V/68 User's Manual*.) If these conditions are not satisfied, processing is terminated, and the delta is not removed. If the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from **D** (delta) to **R** (removed).

6.5.7 The cdc Command. The *cdc*(1) command is used to change the commentary that was supplied when the delta was created. Invoking the *cdc* command is similar to invoking the *rmdel* command, except that the delta to be processed is not required to be the most recently created. For example:

```
cdc -r3.4 s.abc
```

changes the commentary of delta **3.4** of the SCCS file.

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

The *cdc* command also can delete selected MR numbers associated with the specified delta. Do this by preceding the selected MR numbers by the character "!". For example:

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

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

6.5.8 The what Command. The *what*(1) command finds identifying information within any SYSTEM V/68 file when the file name is given as an argument to **what**. *What* searches the given file for all occurrences of a specific 4-character pattern, **@(#)**, and prints out what follows the string up to the first " (double quote), > (greater than), new-line, \ (backslash), or non-printing character. The specific pattern, **@(#)**, is the replacement for the **%Z%** ID keyword. (Refer to *get*(1) in the *SYSTEM V/68 User's Manual.*) For example, if the SCCS file **s.prog.c** (a C language program) contains the following line:

```
char id[ ] @(#)%M%;
```

and the command

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

is executed, the resulting *g-file* is compiled to produce **prog.o** and **a.out**. Then the command

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

produces

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

What is intended to be used with the *get* command which automatically inserts the string **@(#)**, but *what* can also be used when the string is inserted manually.

6.5.9 The sccsdiff Command. The *sccsdiff*(1) command finds and prints (on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by the **-r** keyletter, following the format used with the *get* command. The two versions are the first two arguments to the command and listed in the order that they were created, i.e., the older version is specified first. Any keyletters that follow are interpreted as arguments to the *pr* command (which prints the differences) and must precede the SCCS files. Directory names or a name of "-" will not be accepted by *sccsdiff*.

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

sccsdiff −r3.4 −r5.6 s.abc

6.5.10 The comb Command. The *comb*(1) command generates a "shell procedure" (see *sh*(1) in the *SYSTEM V/68 User's Manual*) which attempts to reconstruct the named SCCS files to make them smaller than the originals. The generated shell procedure is written on the standard output. SCCS files are reconstructed by discarding unwanted deltas and combining specified deltas. *Comb* should be used judiciously.

In the absence of any keyletters, *comb* preserves only the most recently created deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. Middle deltas on the trunk and on all branches of the tree are eliminated. Thus, in Figure 6.3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be removed. Some of the keyletters are summarized as follows:

- The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.
- The **−c** keyletter specifies a list of deltas to be preserved. (Refer to *get*(1) in the *SYSTEM V/68 User's Manual* for the syntax of such a list.) All other deltas are discarded.
- The **−s** keyletter causes the generation of a shell procedure, which when run, produces a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. *Comb* should be run with this keyletter (in addition to any others desired) before any actual reconstructions.

The shell procedure generated by *comb* is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. In addition, the shape of the SCCS file tree may be altered by the reconstruction process.

6.5.11 The val Command. The *val*(1) command checks if a file is an SCCS file that meets the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The *val* command checks for a particular delta when the SID for that delta is explicitly specified via the **−r** keyletter. The string following the **−y** or **−m** keyletter is used to check the value set by the **t** or **m** flag, respectively. (Refer to *admin*(1) in the *SYSTEM V/68 User's Manual* for a description of the flags.)

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

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

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

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

6.6 SCCS Files

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

6.6.1 SCCS File Protections. The SCCS relies on the capabilities of SYSTEM V/68 for most of the mechanisms that prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). In addition to system protections, the SCCS adds another level of protection through the "release lock" flag, the "release floor" flag, the "ceiling" flag and the "user list". (Refer to *admin*(1) in the *SYSTEM V/68 User's Manual*; read the discussion of **-f** for specifics regarding the protection flags.)

New SCCS files created by the *admin* command are given mode 444 (read-only), preventing any changes by non-SCCS commands. Similarly, if directories containing SCCS files are given mode 755, only the owner of the directory can modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and temporary files created by SCCS commands to simplify protection and auditing. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

SCCS files must have only one name; they cannot be linked. (Refer to *cp* in the *SYSTEM V/68 User's Manual* for discussion of linked files.) In addition, all SCCS files *must* have names that begin with "**s.**".

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

However, in situations where several users share responsibility for a single SCCS file, one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS file and be the one with permission to use the *admin*(1) command. This user becomes the "SCCS administrator" for that project. Because other SCCS users do not have the same permissions as the administrator, they cannot execute commands that require write permission in the SCCS file directory. A project-dependent program must provide an interface to the *get*(1), *delta*(1), and if desired, *rmdel*(1), and *cdc*(1) commands.

The interface program is owned by the SCCS administrator. To make the effective user ID equivalent to the administrator's user ID, the program must have the "set user ID on execution" bit "on". (Refer to *chmod*(1) in the *SYSTEM V/68 User's Manual*.) The owner of an SCCS file can modify the file at will. The interface program invokes the desired SCCS command and provides necessary permissions for the duration of the command's execution. In so doing, other users on the "user list" for that file (but who are not the owner) are given necessary permissions for the duration of the interface program execution. These users are able to modify the SCCS files through *delta* and, possibly, *rmdel* and *cdc*. The project-dependent interface program, as its name implies, must be custom-built for each project.

6.6.2 SCCS File Format. The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including the checksum itself).
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.

Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by the SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile*(1).

Because SCCS files are ASCII files, they may be processed by various SYSTEM V/68 commands, such as *ed*(1), *grep*(1), and *cat*(1). This is convenient when an SCCS file must be modified manually (e.g., the date of a delta is incorrect because the system clock malfunctioned) or you want to look at the file.

CAUTION: Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

6.6.3 SCCS File Auditing. On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most SYSTEM V/68 commands) issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine if a file has been corrupted since it was last accessed. The only SCCS command that will process a corrupted SCCS file is the *admin* command used with the **−h** or **−z** keyletters, described below.

Audit SCCS files for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the *admin* command with the **−h** keyletter on all SCCS files:

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

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

corrupted file (co6)

is produced for that file. This process continues until all the files have been examined. Auditing by directories, the second example above, will not detect missing files. To detect files missing from a directory, periodically execute the *ls*(1) command on the directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed.

If a file has been corrupted, the best way to restore the file depends on the extent of the corruption. If damage is extensive, contact the local SYSTEM V/68 operations group and request that the file be restored from a backup copy. In the case of minor damage, the file might be repaired using the *ed*(1) editor. In the latter case, the following command must be executed after the repair:

admin −z s.file

The command recomputes the checksum to bring it into agreement with the actual contents of the file. After the command is executed, any corruption that existed in the file will no longer be detectable.

6.7 An SCCS Interface Program

6.7.1 General. An SCCS interface program provides several users with the "administrator" permissions necessary to work on SCCS files which they do not own. This section discusses the creation and use of an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands because it can perform operations on its arguments.

6.7.2 Function. When only one person uses the SCCS, the real and effective user IDs are the same, and the user's ID owns the directories containing SCCS files. Often, more than one user will need to make changes to the same set of SCCS files. In this situation, one user must be chosen "owner" of the SCCS files and "SCCS administrator", the person with the permissions necessary to use the *admin*(1) command. Because all other users will lack write permission in the directory containing the SCCS files, a project-dependent program is required to interface to the *get*(1), *delta*(1), and if desired, *rmdel*(1), *cdc*(1), and *unget*(1) commands. (Other SCCS commands either do not require write permission in the directory containing SCCS files or are generally reserved for use only by the administrator.)

The interface program is owned by the SCCS administrator but executed by nonowners. To ensure that the program will change the effective user ID to the administrator's user ID, the program must have the "set user ID on execution" bit "on". (Refer to *chmod*(1) in the *SYSTEM V/68 User's Manual*.) This program invokes the desired SCCS command and provides it with necessary permissions for the duration of the command's execution. Users whose login names are in the user list for a SCCS file (but who are not the owner) possess the necessary permissions for the duration of the execution of the interface program. These users can modify SCCS files using *delta*, and possibly, *rmdel*, and *cdc* commands.

6.7.3 A Basic Program. When a SYSTEM V/68 program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. An SCCS interface program uses this mechanism to determine which SCCS command it should subsequently invoke (refer to *exec*(2) in the *SYSTEM V/68 User's Manual*).

A generic interface program (**inter.c**, written in C language) is shown in Figure 6.4. The reference to the (unsupplied) function "filearg" is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

6.7.4 Linking and Use. The steps the SCCS administrator must take to create the SCCS interface program are described. Assume, for the purpose of discussion, that the interface program **inter.c** resides in directory "**/x1/xyz/sccs**". The command sequence

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (the "..." represents other arguments which may be required). The proper mode and the "set user ID on execution" bit are set by executing:

```
chmod 4755 inter
```

For example, new links are created by:

ln inter get
ln inter delta
ln inter rmdel

The names of the links may be arbitrary as long as the interface program can distinguish them from the names of SCCS commands to be invoked. Subsequently, any user may execute:

get—e/x1/xyz/sccs/s.abc

from any directory to invoke the interface program (via its link "get") when the user's shell parameter *PATH* specifies directory **"/x1/xyz/sccs"** as the one to be searched first for executable commands. (Refer to *sh*(1) in the *SYSTEM V/68 User's Manual*.) The interface program then executes **"/usr/bin/get"** (the actual SCCS *get* command) on the named file. As previously mentioned, the interface program could be used to supply the pathname **"/x1/xyz/sccs"** so that the user would only have to specify

get —e s.abc

to achieve the same results.

Table 6-1. Determination of New SID

CASE	SID SPECIFIED*	-b KEYLETTER USED	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
1	none##	no	R defaults to mR	mR.mL	mR.(mL + 1)
2	none##	yes	R defaults to mR	mR.mL	mR.mL(mB + 1).1
3	R	no	R > mR	mR.mL	R.1!
4	R	no	R = mR	mR.mL	mR.(mL + 1)
5	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7	R	--	R < mR and R does not exist	hR.mL**	hR.mL.(mB + 1).1
8	R	--	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9	R.L	no	No trunk successor	R.L	R.(L + 1)
10	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11	R.L	--	Trunk successor in release > R	R.L	R.L.(mB + 1).1
12	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1).1
15	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16	R.L.B.S	--	Branch successor	R.L.B.S	R.L.(mB + 1).1

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

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

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

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

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

```
main(argc, argv)
int argc;
char *argv[];
{
        register int i;
        char cmdstr[LENGTH]

        /*
        Process file arguments (those that don't begin with "-").
        */
        for (i = 1; i < argc; i++)
                if (argv[i][0] != '-')
                                                        arv[i] = filearg(argv[i]);
        /*
        Get "simple name" of name used to invoke this program
        (i.e., strip off directory-name prefix, if any).
        /*
        argv[0] = sname(argv[0]);

        /*
        Invoke actual SCCS command, passing arguments.
        */
        sprintf(cmdstr, "/usr/bin/%s", argv[0]);
        execv(cmdstr, argv);
```

Figure 6-4. SCCS Interface Program "inter.c"

7. UNIX-to-UNIX CoPy (uucp) TUTORIAL

7.1 Introduction

7.1.1 General. The UNIX-to-UNIX CoPy (uucp) Tutorial is a supplement to the discussions of uucp contained in the *SYSTEM V/68 User's Manual* and *SYSTEM V/68 Administrator's Manual.* The manuals are organized as alphabetized entries within tabbed sections. The tabbed sections 1 through 6 (including 1C) are contained in the *SYSTEM V/68 User's Manual* and sections 1M, 7 and 8 are contained in the *SYSTEM V/68 Administrator's Manual.* Throughout the documentation, references to entries within these manuals are given as *name*(section). For example, *uucp*(1C) is a reference to the *uucp* entry in section (1C) of the *SYSTEM V/68 User's Manual.*

The following conventions identify arguments, literals, and program names:

- **Boldface** strings are literals and are to be typed as they appear.
- *Italic* strings represent substitutable argument prototypes and program names.
- Square brackets ([ ]) indicate that an argument is optional.
- Ellipses (. . .) show that the previous argument prototype may be repeated.

7.1.2 Organization. Paragraphs 7.1 through 7.6 of this tutorial contain the information a user needs to send or receive files, mail, and commands over the international uucp network. Paragraphs 7.7 through 7.11 address issues that concern system administrators or users who are interested in the details of data transfers and command executions. Although the discussions in the later half of the tutorial offer greater depth, they do not assume more technical knowledge.

A brief description of each section in the tutorial is presented in the following paragraphs.

- "THE UUCP NETWORK"

  This introductory section provides a description of the network, its size and scope, and business applications. The links between different systems are explained in general terms.

- "UUCP PROGRAMS AND FILES"

  This section provides an operational overview of the uucp programs and files required for execution. A diagram of the system directories identifies the programs and their associated files by name and location. The function of each program and file is explained individually, followed by a discussion of the interaction among the files during data transfers and command execution.

- "USING UUCP"

  This section explains how to enter uucp commands. File naming conventions and command line syntax are provided. Examples for each program are included. A hypothetical uucp network is introduced as a point of reference for the tutorial examples.

- "JOB CONTROL"

  This section explains troubleshooting techniques that are available to the user. Users have access to a log of uucp activities. This section describes how to read the log, and provides a list of possible error messages and their meanings. Basic corrective actions are described.

- "USENET"

  This section describes Usenet, an international bulletin board maintained by uucp users world-wide.

- "ADMINISTRATOR'S OVERVIEW"

  The tutorial sections aimed at a system administrator audience begin here. This section describes the operation of the major uucp programs, *uucp*(1C), and *uux*(1C). The section focusses on the *uucico* and *uuxqt* programs, which execute the commands **rmail**, **mail**, **uucp**, and **uux**.

- "ADMINSTRATIVE CONCERNS"

  This section describes the design issues confronting an administrator who is ready to install uucp. Both hardware and software issues are covered.

- "MAINTENANCE AND ADMINISTRATION"

  This section describes the routine procedures required for uucp maintenance and general administration. The uucp programs *uuclean*(1M), *uulog*(uucp(1C)), and *uusub*(1M) are described here.

- "INSTALLATION"

  This section provides a step-by-step procedure for installing uucp. Information about uucp program variables is provided for administrators with source code.

- "DEBUGGING"

  This section describes some of the problems most commonly incurred during uucp operation and suggestions for corrective action.

7.2 The Uucp Network

7.2.1 Introducing uucp. The UNIX-to-UNIX CoPy (uucp) network is rapidly gaining acceptance in the private sector as an alternative form of electronic communication. A company that targets UNIX and SYSTEM V/68 users advertises its network address regularly in trade publications to stimulate inquiries from programmers. Authors of UNIX-related trade articles frequently provide their uucp addresses to readers to encourage comments and discussion.

The uucp communications network continues to flourish as an informal association of colleges, universities, research laboratories, and private corporations that have voluntarily linked their systems into a point-to-point network of users. For UNIX-based operating systems, the network represents the final step in system-to-system compatibility.

To join the network, a system administrator contacts the administrator of a system that is already part of the network. The adminstrators exchange telephone numbers and login information. After the information is entered into the appropriate files, the systems can contact each other over standard data communication lines, agree on a common protocol, and organize a data exchange. Systems that can contact each other directly over the uucp network are said to be "known" systems.

Direct data exchanges between dissimilar machines over the Direct Distance Dialing network is a huge step forward in system-to-system compatibility, and the foundation of the international network. Point-to-point communication between known systems is coupled with each system's ability to forward data to any other known system. If System-A is known to System-B, and System-B is known to System-C, mail can be forwarded from System-A through System-B to System-C. System-A does not need to be known to System-C.

Data that is forwarded through one or more systems before reaching its final destination passes through each system's "public" directory. Each data transfer is a point-to-point hop from one public directory to the next. The public directory is defined in every SYSTEM V/68 or UNIX-based operating system as **/usr/spool/uucppublic**. The public nature of the directory means that it is readable, writable and searchable by every user on the network. Yet, while the **/usr/spool/uucppublic** directory is accessible to more than 100,000 users, uucp's pathway restrictions effectively close off every other directory in the system, ensuring system integrity. There is no limit on the number of hops that data can make on its trip along the network.

For example, consider uucp network model shown in Figure 7-1. The solid lines connect "known" systems. The Sys-A administrator contacts the administrator of Sys-1, which is part of the uucp network. The administrators exchange information and Sys-A becomes known to Sys-1. Sys-A can now contact Sys-1. In addition, Sys-A can forward data to every other system in the model, and every other system can forward data to Sys-A.

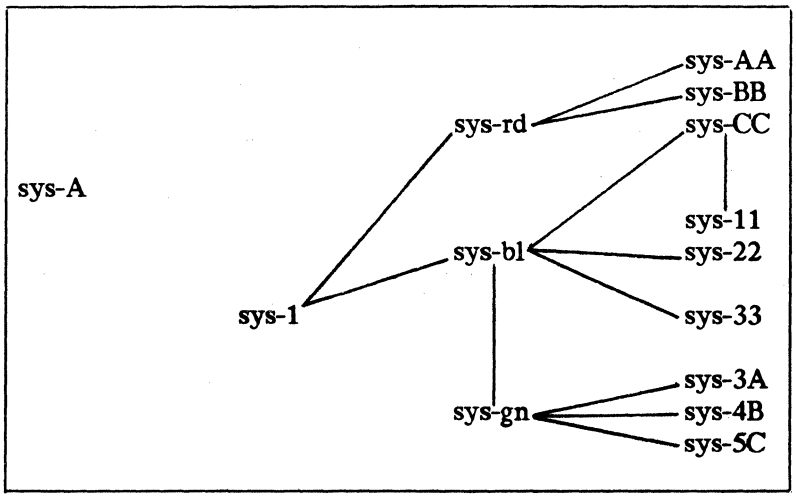

Figure 7-1. Basic Uucp Network

The uucp programs of primary interest to the user are:

mail(1), which sends messages and files;

uuname, which lists all known systems;

uuto(1C) and *uucp*(1C), which send and copy files;

uupick (refer to *uuto*(1C)), which helps locate arriving files;

uustat(1C), which checks the status of your uucp jobs; and

uux(1C), which directs remote command execution.

Other uucp programs of interest to system administrators are: *uuclean*(1M), *uulog*(refer to *uucp*(1C)), *uustat*(1C), *uusub*(1M), *uucico*, and *uuxqt*.

7.2.2 Network Communications. Three types of system-to-system communication are possible over the uucp network:

- Mail
- File Copy
- Remote Command Execution

7.2.2.1 Mail and Uucp. Uucp extends the range of the local operating system's *mail* program to include every user on the uucp network.

Recall that within the local system, users can mail messages and files back and forth, or redirect output to a file and mail it to one or more users. For two users on the same operating system, sending mail is a simple matter:

mail peter
Hello Peter. Is the game still on?

Refer again to the network model in Figure 7-1. After Sys-A becomes known to Sys-1, a user on Sys-A can send mail to a user on Sys-1 by incorporating the recipient's system-name into the mail command:

mail sys-1!mike
Hello Mike. Hamman and Wolff are enthusiastic. It's on.
.

The exclamation mark (!), which is referred to as "bang", delimits the system-name.

If mail must be forwarded through one or more known systems, the uucp programs follow the path specified in the mail command.

mail sys-1!sys-rd!sys-AA!bob
We're ready, Bob. Peter and I can play Saturday. How about meeting at the Cavendish?
.

The mail message passes through the "public" directory in each UNIX-based system until it reaches **bob** working on Sys-AA.

7.2.2.2 File Copy With Uucp. The uucp file copy program, *uucp*, appears to be an extension of the local system's copy program, *cp*(1). For example, if Mike enters the command:

uucp numbers sys-bl!sys-CC!sys-11!hamman

the file **numbers** is copied from Sys-1 to Hamman's home directory in Sys-11 (refer to Figure 7-1). However, system-to-system copies must take into account read and write permission protections enforced on both the sending and receiving system. These concerns are described in the next few sections.

7.2.2.3 Remote Executions. Remote command execution through uucp opens up the processing capabilities of an individual system to the entire network of remote users. Clearly, the use of remotely executable commands must be compatible with maintaining system security. Applications that exploit these possibilities without sacrificing system integrity are described in the sections that follow.

7.2.3 Business Applications of Uucp. Each UNIX-based system that joins the uucp network becomes a link. By traveling from one system to the next, users on the network can contact systems in North America, Europe, Japan and Australia. Uucp has created an international community of users who communicate regularly to exchange information and share expertise. Each system is protected from vandalism through a system of access limitations that allows unknown users to send mail but restricts these users from all non-public directories and from executing any other commands.

Uucp eliminates many of the risks and delays associated with sending hardcopy information or diskettes through the postal system. Data communication lines permit immediate transmission when time is crucial, or transmissions can take place overnight to take advantage of non-peak pricing considerations.

In practice, reaching someone through uucp mail is often more successful than calling on the telephone. Mail avoids the frustration of long-distance "telephone-tag" between two people continually trying to return each other's calls. Lengthy messages can be sent through the mail; no more deciphering cryptic six-word condensations left by whoever happened to answer the phone.

Using uucp, a system administrator can provide users from a remote system with limited access to a particular system. By identifying particular login names and granting these names permission to specified path-names, administrators can monitor the degree of access given to each user. One user may have permission to read one group of files; another user may have permission to execute commands throughout the system. For added security, administrators

can require a call-back to confirm the identity of logins from remote locations.

Geographic limitations virtually disappear within the uucp network. Since uucp mails information to a system address, not a physical address, mail sent to a home directory reaches recipients as soon as they log on, whether they are at home, at the office, or in the field.

Product teams can draw on resources available at remote locations, as well as locally. File transfers enable information to be shared among users quickly and efficiently. Remote command execution enables users to gather information from several remote locations, process the data, and mail the results to any system on the network.

Despite the sophistication of the system, the majority of uucp commands follow the syntax of simple mail and copy commands. Documentation specialists can send on-line documentation to remote locations for typesetting or graphics development. Secretarial support personnel can forward mail and daily reports to field personnel and regional sales offices. Within a small system of only a few users, uucp can archive information to or from a mainframe, or do remote testing of development software.

7.2.4 Network Characteristics. The network has been evolving and expanding for more than a decade, and the programs that make up uucp reflect this fact. The benefits of file transfer and inter-system communications must be accompanied by some additional security precautions.

A crucial part of making your system secure is to forbid access to your files by unknown users. A standard practice is for each administrator to create a wall of permissions that keeps uucp's directories isolated from the rest of the system. Administrators may also limit uucp activities to mailing and receiving files. Within a single company, two systems can be fully integrated under uucp, but restrictive permissions may be needed to control copying of confidential material.

One consequence of the informal nature of the network is that no organized address book for network systems exists. Few resources have been spent developing a network directory because it is difficult to keep such a directory completely up-to-date.

Each system's address is obtained by tracing a path, system-by-system, from it to a major hub or backbone site on the network. Deducing the address from my system to your system involves combining the pathway from my system to a backbone site and the pathway from your system to another backbone site, and then linking the two backbone sites. In practice, this information can be difficult to obtain since the partial, informal maps that exist change dynamically without warning. One system may connect to another for a short time to exchange information, then dissolve the link. Learning the links between systems is generally done by word-of-mouth. Start by using the **uuname** command for a list of systems that are known to your system. (Refer to paragraph 4.3.5.)

A second source of information is the Usenet (Users' Network) bulletin board. Usenet is a logical network that spans several physical networks, including uucp and ARPANET. A group of programs, referred to as *netnews*, provides access to the bulletin board and transfers the information between machines. If Usenet is installed on your machine, you can read posted information and decipher many of the return addresses. In addition, partial maps are published periodically. (Refer to paragraph 7.6 for further information about Usenet.)

Even with a valid pathway between systems, minor obstacles may arise. Uucp communication is somewhat time-dependent, typically for economic reasons. When a user sends mail, the sending system pays for the call to the first system in the pathway. When the data arrives at the first system, the calling system is relieved of the responsibility to forward

the call, or to pay for any more calls. The first system either initiates a a call to the next system in the pathway, or waits to be polled. Each active system along the path assumes the cost of the call from its site to the next site given in the pathway. Many sites make and receive calls anytime during the day. However, each system administrator has the option of restricting the hours that incoming calls are accepted or outgoing calls are made, either because of cost considerations or to keep shared voice/data lines open. Occasionally, a message may be delayed at a particular site until a time restriction is lifted. Users report that a message from Phoenix takes from three to five days to reach Europe or Australia through a typical chain.

7.3 Uucp Programs and Files

7.3.1 Overview. Uucp is a collection of programs that permit communication between UNIX-based systems through dial-up or hardwired communication lines. Occasionally, users may want to follow the progress of a particular command, or may need to troubleshoot a command that did not succeed as intended. For these reasons, uucp is most useful to users who understand the steps involved in command processing. This section describes the function of each command in general terms, and explains how uucp files interact during processing. Actual command syntax and naming conventions are described in paragraph 7.4, "Using Uucp". The depth of information presented here is geared to the uucp user; detailed descriptions appropriate to a system administrator audience are included in paragraph 7.7.

7.3.2 The Spool Directory. Uucp operates in a batch environment. Jobs submitted to the network are assigned a sequence number for transmission. Each job is represented as a file in the common spool directory, **/usr/spool/uucp**. (The common spool directory should not be confused with the "public" directory, **/usr/spool/uucppublic**.) When the file transfer mechanism begins operation, it selects a system to contact and transmits all jobs waiting in the spool directory. Before breaking communication with the remote system, the local system accepts any jobs waiting for it at the remote site. If work is waiting in the spool directory for more than one remote system, the local system makes a random selection of the system to call first. Uucp can communicate with several systems simultaneously; the limiting factor is the number of available communication lines and hardwired connections.

Users trying to exchange information with a known system may experience difficulties or delays. For example, if a remote system is connected through a dial-up connection, all lines to the remote system may be busy or the remote system may be down. If the local system cannot contact a remote system immediately, the job remains in the spool directory and the local system continues trying to execute the job until the file is removed by a clean-up program (typically after 48 hours). (For further information about the uucp clean-up programs, refer to paragraph 7.9.)

7.3.3 Uucp Directories. A branch diagram of uucp-associated files is illustrated in Figure 7-2. The diagram represents a system frozen at one point in time; the number and names change constantly during processing. For clarity, some SYSTEM V/68 files that may be found in these directories have been omitted from the diagram because they are of no concern to the uucp user. (For a more complete diagram, see Figure 7-4.)

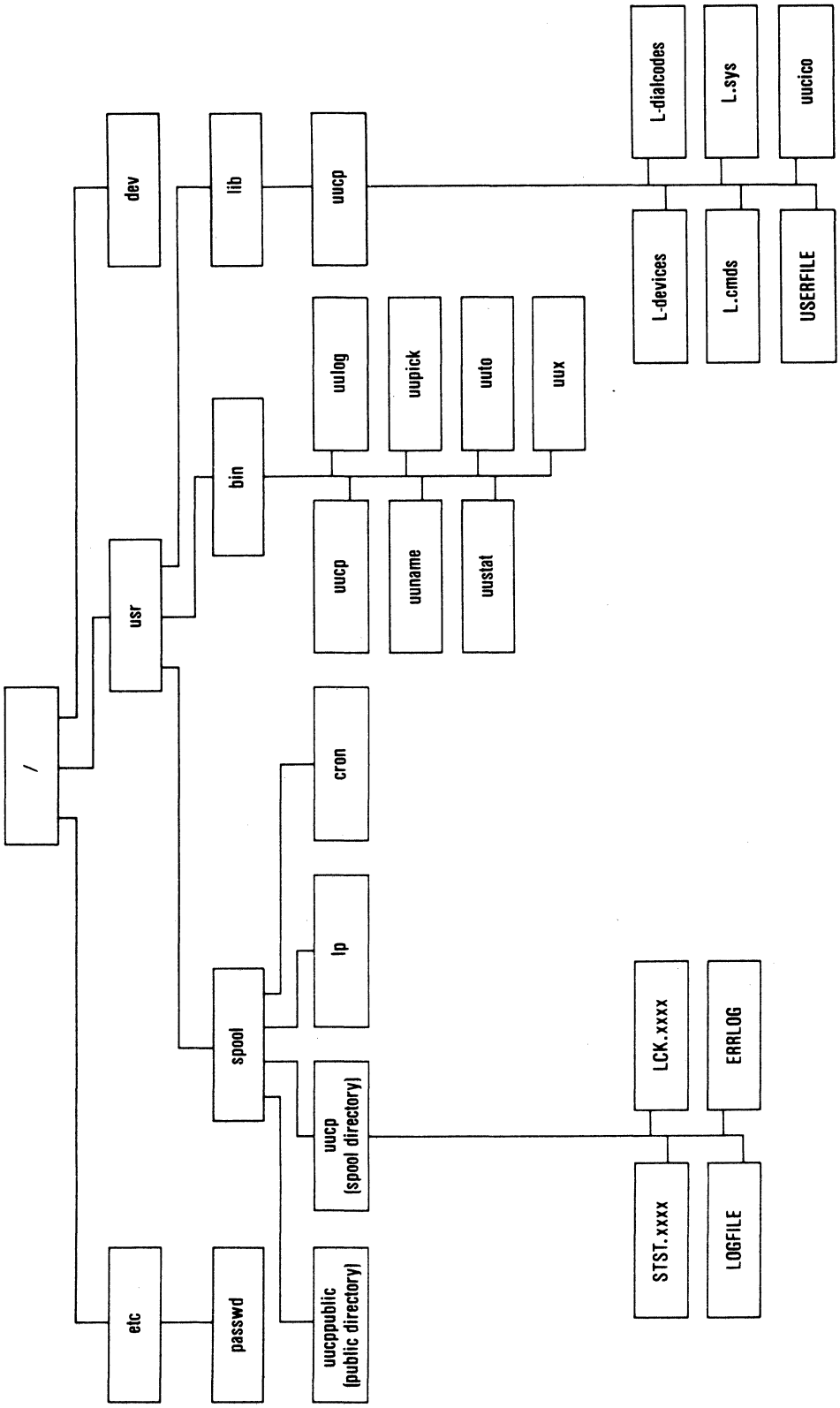

Figure 7-2. Branch Diagram of uucp Directories and Files

7.3.4 Uucp Programs. The **/usr/bin** directory contains the executable code for the uucp programs. The name and function of each program are described in the paragraphs that follow.

7.3.4.1 uucp. *Uucp*(1C) is one of a cluster of programs that are collectively referred to as the UNIX-to-UNIX CoPy programs (uucp). *Uucp* developed as the precursor to the complete communications package; it copies a named source file to a named destination file. The files specified can be expressed as pathnames to any system on the uucp network.

For a copy command to succeed, the named source file must be readable by *uucp*. To ensure that the source file, the directory containing the file, and every directory leading up to it are readable by *uucp*, all these files must have "world" permissions. That is, all these files must be readable and searchable by everyone. Similarly, every directory leading up to the destination file must be readable and searchable by everyone. If the destination file exists, it must be writable by everyone. If the command requires creating a new file, the destination directory must be writable by everyone, in addition to being readable and searchable.

Because of these permission requirements, most system administrators configure *uucp* to send and receive data through a specially designated public directory: **/usr/spool/uucppublic. A** second approach is to transfer files with *uuto*, which copies any file for which the user has read permission. *Uuto* is described in paragraph 3.4.5.

7.3.4.2 uuname. *Uuname*(uucp(1C)) lists the remote systems that are known to the local system. For example, the command **uuname** on a system might generate the list:

sundog
varecn
mornin
hist30

The list indicates that data can be forwarded to any system whose address begins with one of these four system-names.

A **—v** option (**v**erbose) reads the file **/usr/lib/uucp/ADMIN** (if the administrator maintains it). The command **uuname —v** may provide additional descriptions of the known systems. Often the system-name does not contain enough clues to define what or where it is.

The **—l** option displays the uucp network system-name used by the local system. In most of the examples that follow, the local system is referred to by the system-name **home.**

7.3.4.3 uulog. *Uulog*(uucp(1C)) reads and prints entries from the file **/usr/spool/uucp/LOGFILE**, which contains a summary of uucp-associated transactions. The **—u** option displays user-specific entries; the **—s** option displays system-specific entries.

7.3.4.4 uustat. *Uustat*(1C) displays the status of uucp requests issued by the user. Optional invocations of *uustat* can cancel on-going jobs, report the status of specified commands, or provide information about *uucp* connections to other systems. Currently, when *uustat* cancels a job, the status list is not updated to reflect that the job was killed.

7.3.4.5 uuto and uupick. *Uuto* and *uupick* are complementary programs: *uuto* is invoked to send a file **to** someone, and *uupick* is invoked by the receiver to **pick** up the file.

The distinction between *uuto* and *uucp* is that *uuto* enables *uucp* to copy any file for which the user has read permission; *uucp* permissions are not required.

The **uuto** command specifies a source file or directory **and** a destination user-id. If a directory is specified, the command copies the complete directory tree. Most commonly, the command **is** invoked with options that first copy the source file (or directory) to the public directory in **the**

local system. From there, the source file (or directory) is copied to the public directory in the recipient's remote system. Specifically, the source file is copied into the subdirectory

/usr/spool/uucppublic/receive/*user-id/sending-system/filename*

where *user-id* is the recipient's user-id and *sending-system* is the system where the source file originated.

The recipient learns through *mail*(1) that a file has arrived and executes **uupick** to retrieve the file. *Uupick* searches the local system's directory

/usr/spool/uucppublic/receive/*user-id*

for files. For every file found in the directory, *uupick* displays a message giving the filename, the system where it originated, and a request for instructions about what to do with the file. Options available to the user under *uupick* include moving the data, printing it, deleting it, or taking no action. If no action is taken, the file remains in the directory and with the next invocation, **uupick** will repeat the message. If desired, a user may **cd** to the public directory and execute commands. *Uupick* is a convenience feature.

(**NOTE:** Currently *uupick* fails in its attempt to change permissions and owners if the public directory and the destination directory are in the same filesystem. Files are transferred without errors, but the owner remains uucp. An incorrect error message indicating "0 blocks copied" may also appear.)

7.3.4.6 uux. *Uux*(1C) provides for remote program execution, and is the most powerful of the uucp programs. Several programs can be executed remotely by piping the output of one program to the input of the next. Standard input and standard output can be redirected to any system on the network. In general, *uux* can gather any number of files from any number of systems, execute a command on a specified system, and then send the standard output to one or more files on any specified system(s). For example, *uux* can gather accounting data from five different systems, ship it to a sixth system, where it is processed, and send the output file to a seventh or eighth system. If you need a list of filenames from a remote system, *uux* can execute an **ls -l** on the remote system and redirect the output to your directory.

Each system administrator maintains a **/usr/lib/uucp/L.cmds** file that lists the commands the local system may execute on behalf of an incoming *uux* request. For security reasons, most system administrators limit this list of commands to two: **rmail**, a restricted version of *mail*(1); and **rnews**, for sending and receiving Usenet.

7.3.5 Files Involved in Program Execution. Users rarely need to read any of the uucp-associated files. However, if a program fails to execute as expected, a well-informed user is much better prepared to troubleshoot the problem, or cancel and re-initiate the task. The following paragraphs describe the files involved in uucp program execution, and their functions.

7.3.5.1 Required Files. Four files are required for the uucp commands to function, all of which reside in **/usr/lib/uucp**:

L.sys
L-devices
L-dialcodes
USERFILE

Background information describing the contents of each file may help diagnose problems.

L.sys

Each entry in this file represents a known remote system. The **uuname** command reads this file to generate its list of known systems. Because the **file** contains passwords for some remote systems, the file should be owned by the uucp administrative login and should not be readable by all users.

Each entry provides the local system with sufficient information to initiate a conversation. The information includes the times when the system may be called, the type of device used for the call (e.g., an automatic calling unit, an auto-dial modem, or a direct line), the line speed, the telephone number, and the login sequence.

L-devices

The entries in this file describe the devices and hardwired connections used by uucp. Each entry line describes the device type, the special device filename, the associated calling-unit (if there is one, else **0**), and the line speed. Examples of entry lines are:

DIR tty21 0 9600
ACU cu10 cua0 1200

The first entry indicates the local machine is directly connected to device **/dev/tty21** at 9600 baud. The second entry describes a setup where an automatic calling unit device, **/dev/cu10**, is wired to a call-unit, **cua0**, for use at 1200 baud. (The programs included in Release 2, Version 1 support auto-dial modems, or "smart" modems, but do not support automatic calling unit devices. Refer to paragraph 10.3 for further information about auto-dial modems.)

L-dialcodes

This file may be used when automatic calling units are part of the uucp system. The file contains reference telephone numbers, typically area codes or exchange prefixes, for some of the known remote systems. An example entry line is:

sys-1 503

where the area code **503** is prefixed to the **sys-1** telephone number contained in **L.sys**.

USERFILE

USERFILE limits user accessibility by placing selective constraints on the uucp operations. **USERFILE** correlates the login of the requesting user or remote system with a list of accessible pathnames. Since the system administrator assigns login-ids and passwords to each remote system and user, the administrator has an effective means of controlling who has access to the directories and files within the local system. In addition, the administrator places an entry in **USERFILE** if a call-back is required for a remote system.

7.3.5.2 Files Created by Uucp Program Execution. Several files are created during command processing which are later deleted. One way to troubleshoot a command is to watch the process and monitor each file as it is created. Three files bear watching:

/usr/spool/uucp/LOGFILE
/user/spool/uucp/LCK. ***xxxx***
and
/user/spool/uucp/STST.*xxxx*

LOGFILE

LOGFILE is a record of actions taken by the uucp group of programs. Contained in the file are one-line entries for each transaction. These transactions include calls made by the local system, calls received from remote systems, requests for files to be transferred, and requests for programs to be executed remotely. Remote mail transfer requests can be traced indirectly by watching the **LOGFILE** for records of file transfer or remote program execution.

Instructions for accessing and reading **LOGFILE** are contained in paragraph 7.5, "Job Control".

LCK. *.xxxx*

LCK. *.xxxx* files are "lock" files and are created for each device and system when they are in use. The lock file prevents duplicate conversations and foils attempts to use a single device for more than one connection. The *xxxx* identifies the particular device or system. If an execution aborts, a lock file may be left containing misinformation that the device or system is busy. If this happens, the lock file must be removed manually by the user or administrator.

Instructions for locating and removing lock files are included in paragraph 7.5, "Job Control".

STST.*xxxx*

The **STST.***xxxx* files serve two purposes. First, an **STST.***xxxx* file is created when a connection fails because of a bad login, password or dial-up line. Second, an **STST** file containing a "talking" status is created while two machines are communicating. The *xxxx* identifies the remote system to which the local system is talking (e.g., **STST.sys-1**). For login or dial-up failures, the existence of an **STST.***xxxx* file prevents the local system from attempting another connection with machine *xxxx* for some period of time, usually an hour. The length of time is defined by the administrator in the **L.sys** file, and can vary among systems.

Users can remove the **STST.***xxxx* file and re-enter a command before the default time period elapses. If a program aborted while the **STST.***xxxx* file contained a talking status, the system will prevent any calls to the system indefinitely. The user must remove the file manually.

Instructions for locating and removing **STST.***xxxx* files are contained in paragraph 7.5, "Job Control".

7.3.6 File and Program Interaction. The information contained in the uucp-associated files affects command processing at several key junctures. This paragraph describes the interaction among the programs and files in general terms.

For this discussion, assume that you have entered a command to send a mail message to a user on another system. You are working on system **sys-A** and you are sending a file to user **bob** on system **sys-AA**. The pathway from **sys-A** to Bob is through **sys-1** to **sys-rd** to **sys-AA**:

mail sys-1!sys-rd!sys-AA!bob
What did you get for a lead on Board 7?

The first step taken by the executing programs is to try to contact the first system in the pathname: **sys-1**. System **sys-A** reads the **L.sys** file for the **sys-1** entry containing:

The time of day the system can be called
The login and password **sys-A** should give **sys-1**
The type of calling device to use to call **sys-1**
The telephone number

If the phone number contains an abbreviation, the **L-dialcodes** file is read to get the complete number.

(The local system may read the entry in the **L.sys** file and discover that there is no time when the local system is allowed to call the remote system. This will be the case whenever the local system is a polled system. Instead of initiating a call, the local system will queue the job in its spool directory and wait for the remote system to call. For further information about polled connections, refer to paragraph 4.1)

Before making the call, **sys-A** scans its spool directory to see if an **STST.sys-1** file is present. If the file exists, **sys-A** waits the required time before calling. Each time a call fails, the attempt is recorded in the **STST** file. After the number of tries reaches a maximum (defined by the administrator), an error message reports **NO CALL (MAX RECALLS)**. Although the job remains in queue, no further processing takes place until the **STST** file is removed. Old jobs that have not been executed are purged from the spool directory periodically by *uuclean*(1M). (For further information on error messages, refer to paragraph 7.5. For further information on the uucp maintenance programs, refer to paragraph 7.9.)

Once the system has the calling information, and has checked that all is clear for a call, it scans the **L-devices** file to learn **sys-1**'s device requirements. The local system searches for an available device, ignoring all devices with a **LCK.** *.xxxx* file. If no devices are free, the system waits to call later.

When the call is made to **sys-1**, the standard login sequence takes place. Once **sys-A** has logged in (probably under a shared login, such as **uucp**), the systems identify themselves to each other by their unique system-names. Each system reads its own **USERFILE** to determine the degree of access entitled to the other. System **sys-A** sends **sys-1** the message that it wants forwarded to **sys-rd!sys-AA!bob**. System **sys-1** checks its **L.cmds** file and confirms it may forward mail on behalf of **sys-A**. **Sys-1** starts up an *mail* command to **sys-rd** on behalf of **sys-A**. **Sys-A** searches its spool directory for any other jobs for **sys-1**. If there are none, **sys-A** asks **sys-1** it has any jobs waiting for **sys-A**. If not, the conversation ends.

Sooner or later, **sys-1** repeats the same process when it contacts **sys-rd** to forward the mail message, and **sys-rd** will repeat the process once again when it contacts **sys-AA** to deliver the message to **bob**.

7.4 Using Uucp

This section describes the correct syntax for entering each uucp command and its available options.

The *mail*(1) utility is also described with respect to its applications on the uucp network through the uucp program *uux*. Although **mail** and **rmail** are not uucp commands, the uucp network allows *mail* to move information through systems that place restrictions on all other uucp commands.

The discussions that follow assume your system administrator has configured your system for all the uucp commands. However, many administrators may place restrictions on the *uucp* utility. Meet with your administrator to learn the limitations, if any, of your specific environment. For all command descriptions, Figure 7-3 serves as a hypothetical uucp map provided by a system administrator. Refer to the map often as you read through these examples.

7.4.1 Network Architecture: Understanding the Links. From Europe to South Korea, Australia to Canada, information flows across the uucp network along a machine-to-machine pathway. Four types of links are possible: any combination of a hardwire/dial-up connection (hardware), and an active/polled connection (software). Once data is received by the next system along the path, the responsibility for forwarding the job shifts from the sending system to the receiving system. The crucial link, from the user's perspective, is the first one in the path, the link from the local system to the known remote system.

SYSTEM V/68 supports all four types of connections. Two systems can be directly connected by cross-coupling two of the computer ports through a null modem. Hardwired connections are successful only for short distances up to several hundred feet. To create a dial-up connection, a user can employ an auto-dial (smart) modem that enables the local system to contact remote systems by direct dial. SYSTEM V/68 Release 2, Version 1 supports an auto-dial modem; it does not support an automatic calling unit (acu).

Whether a site uses an active or polled connection is independent of the type of line used. Choosing between an active or polled connection is an administrative decision, based on the planned applications and system load considerations. Since the polled system cannot initiate calls, a job in the spool directory must wait until the polling system calls. Typically, administrators arrange for an active system to use *cron*(1M) to schedule a periodic *uusub* poll. (Further information about *uusub*(1C) is contained in paragraph 7.9.)

The type of connection, active or polled, is specified in the **L.sys** file. (Detailed information about the **L.sys** file is contained in paragraph 10.4.) The **L.sys** file, which lists all remote systems to which you can direct jobs, includes an entry that defines the correct time for the local system to call the remote. For a polled system, the **L.sys** file entry makes it the "wrong time to call" all the time. The effect is that your uucp jobs are accepted and queued in the spool directory because they are directed to "known" systems, but calls are never made. Instead, jobs remain queued until the active system calls your local system. One by-product of this arrangement is that when a uucp job is executed on a passive system, the error message **WRONG TIME TO CALL** routinely appears in the **LOGFILE**.

Consider the network map in Figure 7-3 that describes the network from the perspective of system **home**. For the discussions that follow, user **pat** on system **home** is working in login directory **/usr/pat**. All systems shown on the map have datasets so remote systems can be connected. The following observations can be made about the network:

- The connection from **home** to **polled** is passive; **home** calls **polled**, but **polled** has no means to initiate communication with **home**. Transfers from **polled** cannot leave the

system until it is called by **home**. The polled connection is represented on the map as a short-dashed line.

- The link from **home** to **drect1** is a hardwired connection, as are the links from **home** to **drect2** and from **drect1** to **drect2**. The direct connection is represented on the map as a solid line.
- The types of connections between all other systems on the map are unknown. Typically, uucp users only know where the links between known system are; details are nice, but not necessary. Connections between remote systems are represented as long-dashed lines on the map.
- Each line (solid, short-dashed, and long-dashed) indicates that the two connected systems are known to each other. Systems that are not connected are not known to each other.
- Every system on this map can be reached by every other system.

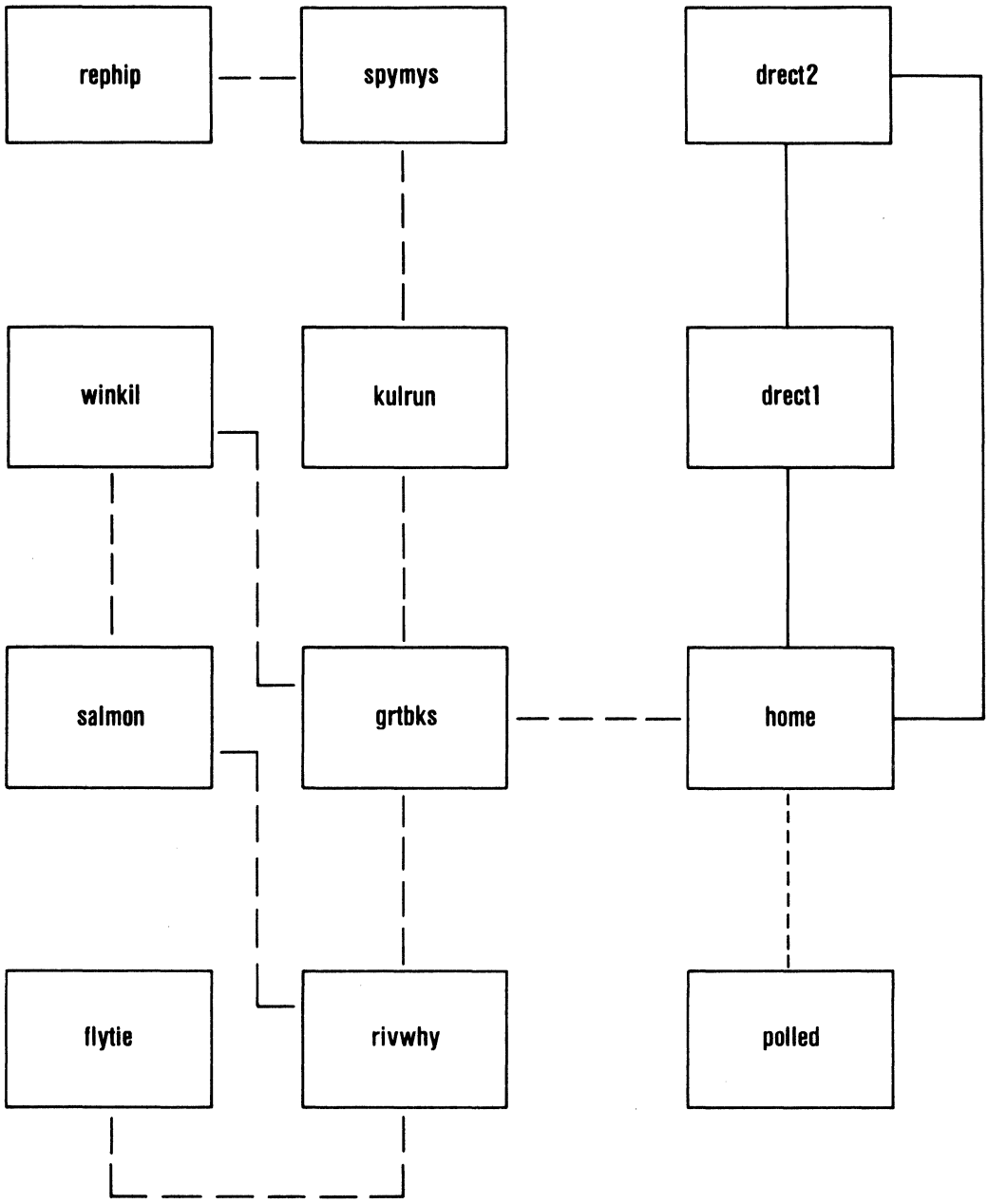

Figure 7-3. View of uucp Network from the System home

7.4.2 Naming Conventions. The command syntax for uucp varies from program to program but general conventions are employed for referring to files and directories on the local and remote system. The convention for specifying a file in a uucp command is

system-name!path-name

where the *system-name* is a path through a known remote system. Thus, reference can be made to a file in a remote system that is not known directly to the local system, but that can be reached by passing through intermediate systems.

The syntax for naming a file or a directory is independent of the type of connection between the two machines. For user **pat** on **home** to send mail to user **jrh** on **rephip**, the command line is:

mail grtbks!kulrun!spymys!rephip!jrh

Similarly, mail to **alex** on the system **salmon** is sent with either of the commands:

mail grtbks!winkil!salmon!alex
or
mail grtbks!rivwhy!salmon!alex

The *system-name* is always the path from the local system to the remote.

Path-names included in uucp commands may take five different forms:

1. Complete paths, beginning with root and ending with a filename;
2. Paths that end with a directory name;
3. A login directory specified with the abbreviation ~ (tilde);
4. The public directory specified with the abbreviation ~ (tilde); or
5. A single filename that assumes the current working directory.

For example, the expression

winkil!/usr/sam/*filename*

locates a particular file *filename* in the directory **/usr/sam** on system **winkil.** The expression

winkil!/usr/sam/memos

identifies a directory **memos** in **winkil** where a file can be copied, keeping its current filename.

The ~ character (tilde) has a special meaning in uucp commands. When used immediately before a user-id, the tilde expands to the pathname to the user's login directory. For example, the expression

winkil!~sam

expands to the login directory **/usr/sam** on the system **winkil.** In the same way, the expression

winkil!~sam/memos

expands to the complete pathname **/usr/sam/memos.** When the ~ (tilde) is followed by a */user-id*, the tilde expands to the system's public directory. For example, the expression

winkil!~/sam

expands to the directory **/usr/spool/uucppublic/sam** in the system **winkil.** An unlimited

number of subdirectories are available in a system's public area. The expression

winkil!~/sam/memos/december

expands to the directory **/usr/spool/uucppublic/sam/memos/december.**

Filenames that do not use any of the combinations described above are prefixed with the current directory on the local system or the user's login directory on the remote. The user's login directory on a remote system is usually defined as **/usr/spool/uucppublic/***user-id*.

The naming conventions include two restrictions. First, pathnames that use the .. (dot dot) abbreviation are not permitted. The system cannot easily check a directory's permission modes when it is referenced as **../**. The second restriction is actually a shell convention, but it affects uucp. Files that begin with a dot, for example **.profile**, are not picked up by special shell characters unless they are qualified with a dot. For example, the abbreviations

.prof*
.profil?

find the file **.profile**. The abbreviations

prof
?profile

do not.

7.4.3 Uucp Commands.

7.4.3.1 uuname *Uuname* lists the *system-names* of remote systems known to the local system. The **-l** (lowercase L) option returns the local system's *system-name*. The **-v** option prints additional information about each system, if it is available. The additional description information is drawn from the file **/usr/lib/uucp/ADMIN**. Not all administrators maintain this file.

The command format is

uuname [*options*]

7.4.3.2 mail For security reasons, most administrators limit the list of commands executable for another system to two: **rnews**, for sending and receiving the network news; and **rmail**, a restricted version of *mail*. By not including *uucp*, a remote system is prevented from initiating a copy of a file on the local system.

Standard UNIX and UNIX-based software includes **/bin/mail**, the *mail*(1) program. If user **pat** on **home** decides to send mail to **con** on system **flytie**, the command is

mail grtbks!rivwhy!flytie!con
Hello Con. Did you get my file?

The *mail* program takes the message from the standard input and uses the uucp programs to send the message to **grtbks**. Once the message arrives at **grtbks**, the *rmail* program is invoked (as a link to **/bin/mail**). The system removes its own name from the address and starts up the uucp programs to forward the mail to the next system in the pathname. The sequence repeats itself at each site until the mail reaches **con**.

The command that sends a mail message can be easily revised to to take input from a file instead of the standard input. The command:

mail grtbks!rivwhy!flytie!con < materials

mails the contents of **materials** from the current directory to **con** through all intermediary systems.

One characteristic of mail is that the burden of correctly addressing mail lies with the sender. Because of the informal nature of the network, there are no maps that list all connections between sites. Administrators are not required to notify anyone when connections between systems are established or dissolved, and with more than 3,000 systems on the network, the overall structure is in a constant state of flux. The **mail** command requires an explicit path from the sending system to the receiving system. The difficulty of obtaining such an address varies greatly, depending on the size and location of the two sites.

A second observation is that mail is not well-suited for sending "object" or non-ASCII files since the recipient cannot read the mail when it is delivered. Use *uucp* instead.

7.4.3.3 uucp The syntax of the **uucp** command is modeled after the **cp** (*copy*(1)) command. The command takes the form

uucp *from-here to-there*

where the source file arguments and the destination file arguments follow the uucp naming conventions. Options to the **uucp** command direct the system to create needed directories, send mail to the user when the transfer is complete, notify the recipient when the transfer is complete, or queue the job without initiating it immediately. (Refer to *uucp*(1C).)

Each system on the network has a public directory **/usr/spool/uucppublic**. An entry in **/usr/lib/uucp/L.sys... enables any remote system that calls in to have access to the public** directory. Since the local system is providing the remote with a destination directory that is writable by everyone, files can be copied between systems. The command

uucp —m declr grtbks!~/hayden

copies the file declr into the directory **/usr/spool/uucppublic/hayden** on the system **grtbks. The -m option requests that mail be sent to pat when the copy is complete.**

Shell characters retain their special meanings in the **uucp** command syntax. If **pat** enters the command

uucp -m compile/*.c drect1!~/bob

all files in **/usr/pat/compile** that end with **.c** are copied into the directory **/usr/spool/uucppublic/bob** in system **drect1**. Recall that **uucp** prefixes a file or directory with the current directory, here, Pat's login directory **/usr/pat.**

Uucp's application value for transferring files directly from one system to a remote can be extended to allow transfers that flow through intermediary systems. To allow for transfers that pass through intermediary systems, administrators must place an entry in the **USERFILE** that grants remote systems access to the local system's spool directory **/usr/spool/uucp.** If the remote has access, it can place the file(s) to be transferred and a generated Send command directly into the local's spool directory. The local system executes the Send command for the file, forwarding the file on to the next remote system in the path. The security of the intermediary system is not threatened because the remote systems do not have access to any files outside of the pathway **/usr/spool/uucp.** (For further information, refer to the discussion of uucp operation in paragraph 7.7, and the discussion of **USERFILE** in paragraph 7.10.)

7.4.3.4 uuto and uupick *Uuto*(1C) and *uupick* are a complementary pair of programs that execute *uucp* while preserving the permissions on files and directories in both the sending and receiving system. *Uuto* maintains the integrity of both systems by copying files from the sending system to the public directory of the receiving system. The **uuto** command includes a user-id as the specified recipient of the file; when the file arrives, it is identified as

/usr/spool/uucppublic/receive/*user-id/sending-system /filename_or_directory*

The intended recipient, *user-id*, is notified by mail that the file has arrived. The user executes **uupick** to access the file in the public directory. *Uupick*'s options include moving the file to a new directory, printing the file, leaving the file where it is, or deleting it. If a directory is named in the command line, the entire tree is moved.

The command format for *uuto* follows the same design as *uucp*. For example, if user **pat** on **home** enters the command

uuto -m -p econ/forecst drect1!bob

the file **forecst** in **/usr/pat/econ** is copied to the **drect1** system as **/usr/spool/uucppublic/receive/bob/home/forecst**. The **-m** option tells **home** to send mail to **pat** when the copy is completed. The **-p** option copies the source file, **forecst** into the **home** spool directory before initiating the transmission to **drect1**. The file copy remains in the sending system spool directory until the **uuto** copy is executed or deleted.

Invoking the **-p** option involves a tradeoff for the user. Although requiring a copy is somewhat wasteful of cpu time and storage, it frees the user to continue modifying a file or to delete the file completely. Without the **—p** option, the system makes a copy of the file when it is ready to initiate the transfer, which could be several hours after the **uuto** command was entered. By using the **-p** option, the user knows when the file was copied and what the copy contains. In addition, if the **—p** option is not used, the user initiating the **uuto** command must have read permission for the directory and source file.

When the file arrives at **drect1**, user **bob** receives mail that a file has arrived via uucp. (*Uuto* always notifies the recipient of a file transfer.) Bob enters the command

uupick

The options for moving, printing, deleting or ignoring the file are described in the *uuto*(1) manual pages, or may be displayed on the screen by entering a star * for a usage summary.

Since *uuto* uses *uucp* to execute the copy, *uuto* can transfer files from a sending system through an intermediary when the sending system has access to the remote system's spool directory. (For further information, refer to the discussion of uucp operation in paragraph 7.7, and the discussion of **USERFILE** in pragraph 7.10.)

7.4.3.5 uux The remote aspect of *uux* is twofold. First, *uux* can gather files from one or more remote systems, and copy the files back to the local system for some command execution. The same command can then copy the output of the execution to any number of remote systems. Second, *uux* can direct a remote system to initiate a command execution. Whether or not the command is executed depends on the configuration of the remote system. Whenever a remote system receives an incoming *uux* request, it checks the **/usr/lib/uucp/L.cmds** file to see if the request is "executable". If the command that *uux* wishes to initiate is listed in **L.cmds**, the remote system executes it; if not, the remote system disallows the command and notifies the sending system with mail.

By combining these two *uux* capabilities, first, to gather files to or broadcast files from a particular site, and second, to direct another system to initiate a command, an extremely powerful set of tools is available for developing environment-specific applications. The

L.cmds file format allows an administrator to list commands but specify that only particular systems are allowed to execute them.

Consider the situation in a single company where a VAX (**home**) and two VME/10s (**polled** and **drect1**) are operating under SYSTEM V. If a user on **home** enters the command

uux "diff polled!/usr/jake/driv1 drect1!/usr/jake/driv2 > !driv.diff"

the system **home** is directed to copy the files **/usr/jake/driv1** on system **polled** and **/usr/jake/driv2** on system **drect1** to **home**; perform a **diff** on the files, and redirect the output to the file **driv.diff** in the current directory in **home.**

The *uux* command follows the format

uux [—] [*options***] "***command-string***"**

where the — indicates the standard input for the *command-string* is inherited from the standard input of the *uux* command. The *command-string* is composed of one or more arguments that look like a shell command line, except that the command and filenames may be prefixed by *system-name***!**. A null *system-name* is interpreted as the local system. (An argument that does not contain an **!** is not recognized as a file and is not copied to the execution machine. All special shell characters, such as " < | ^ or > must be quoted, either by quoting the entire command-string or by quoting the character as a separate argument.)

For example, the commmand

pr econ/forecst | uux — drect1! lpr

takes the output of **pr econ/forecst** as standard input to an **lpr** command executed on **drect1.**

As another example, the command

uux "drect1!ps (—ef 1) | rmail (home!pat polled!jake)"

directs **drect1** to execute a **ps** with options **e, f,** and **l** invoked, and to pipe the output through mail to **pat** on **home** and **jake** on **polled**. Remember, for this command to succeed, **drect1** must have a line in **/usr/lib/uucp/L.cmds** that specifies it is allowed to execute a **ps** for **home.**

To execute a command on a particular directory or file, follow the example:

uux "grtbks!rivwhy!ls (/usr/isaac/techn) | rmail (home!pat)"

The command directs **rivwhy** to execute an **ls** command on **/usr/isaac/techn** and to mail the output to **pat** on **home.** Refer to *uux*(1) for further information about available options.

7.5 JOB CONTROL

Job control, as addressed in this section, refers to information and commands that can be used to monitor and track a job. The specific areas of job control described here include:

- Notification of Job Completion
- Monitoring a Job Through **LOGFILE**
- Job ID Numbers and *uustat*(1C)
- Job Termination, Requeuing

7.5.1 Notification. Each uucp command can report the success or failure of a transmission to the user asynchronously through the *mail*(1) command. A request that notification be sent to the user's system is made by invoking the **—m** option with the *uucp* and *uuto* commands. The notification is especially useful when initiating a *uuto* command. Often it is inconvenient to logon to the remote system to check if the intended receiver has mail. (Notification is automatically sent to the user whenever a *uux* command is invoked, unless it is specifically suppressed by the **—n** option.)

If desired, the notification report may be directed to a particular file by invoking the option as **—m***filename*, where *filename* follows the general *uucp* naming conventions. The **—m***filename* option is available with *uucp*, *uuto*, and *uux* commands. It is especially convenient when a user wishes to notify a third person of the job completion.

For example, the command

uucp —mpop /max/chklst grtbks!/dev/null

sends the file **/max/chklst** to the bit bucket, **/dev/null** in the system **grtbks**. The status of the transfer is reported to the file **pop** in the sending system's current directory.

An example status report produced by the **uucp** command might be:

uucp job 0306 (8/20-23:08:09) (0:31:23) /max/chklst copy succeeded

where the information supplied follows the format

job-number command-time status-time status

The status report is described in detail in paragraph 5.5 and under *uustat*(1C) in the *SYSTEM V/68 User's Manual.*

7.5.2 Monitoring a Job Through LOGFILE. A log of ongoing actions taken by the *uucp* system is continually updated in the file **/usr/spool/uucp/LOGFILE.** A user can access this file and observe the actions as they occur with the command:

tail -f /usr/spool/uucp/LOGFILE

Tail(1), invoked with the **-f** option, copies the last 10 lines of **LOGFILE**, and then enters an endless loop. Inside the loop, *tail* sleeps for a second, then attempts to read and copy new records from the input file. **Tail -f** monitors the growth of **LOGFILE** record by record. To break the loop and return to the shell prompt, press the Interrupt key.

LOGFILE contains records of requests for file transfers, local system attempts to contact remote systems, remote system attempts to contact the local system, and an abbreviated log of the communication that occurred, if any. An entry is also made when remote program execution occurs.

An edited version of **LOGFILE** from system **zek** is shown below. The system **zek** has links to **zekisv, usavax, bluhil, zek30,** and **unix.** The entries follow the format:

system!user-id (entry-time) (sequence-no.) (status) (more-info)

where the fields are defined as:

system!user-id
The system name and intended recipient of a uucp job.

entry-time
The date and time of the record entry. The date is expressed as (*mm/dd-hh:mm:ss*). (The time may be incorrectly reported if the time zone variable is improperly set. The **TZ** variable is set by the administrator in the login shell of the uucp account.)

sequence-no
The sequence number is assigned by the system in the spool directory.

status
If the entry is an action entry, the status field indicates if the action was successful. If the field contains a name followed by **XQT**, the entry describes a remote program execution that occurred. The added information that follows in the fifth field is the executed command. If the field contains **QUE'D**, the name of the queued command is contained in the additional information field.

more-info
Additional information. This field contains useful diagnostics that may explain why a command has failed, and may lead a user to the appropriate corrective actions.

A detailed discussion of the diagnostic messages included in the **LOGFILE** follows the **LOGFILE** example.

Edited Version Of **/usr/spool/uucp/LOGFILE**

```
zekisv!uucp (12/5-11:41:57) (C,1010,234) COPY (SUCCEEDED)
zekisv!uucp (12/5-11:41:59) (C,1010,234) REQUESTED (S D.zekBC0324 D.zekBC0324 uucp)
zekisv!uucp (12/5-11:42:44) (C,1010,245) OK (conversation complete  tty09 948)
usavax!uucp (12/5-11:42:47) (Q,2683,0) uucp XQT (PATH=/bin:/usr/bin:/usr/lbin LOGNAME=uucp rnews )
zekisv!uucp (12/5-11:42:56) (Q,2683,0) uucp XQT (PATH=/bin:/usr/bin:/usr/lbin LOGNAME=uucp rmail al )
zekisv!uucp (12/5-11:43:57) (Q,2683,0) uucp XQT (PATH=/bin:/usr/bin:/usr/lbin LOGNAME=uucp rmail al )
zekisv!uucp (12/5-11:43:59) (Q,2683,0) ret (1000) from zekisv!uucp (MAIL FAIL)
zek30!uucp (12/5-11:56:48) (C,3329,0) FAILED (DIALUP LINE open cul0 P3232< 8)
zek30!uucp (12/5-11:57:08) (C,3329,0) FAILED (DIALUP LINE open cul0 P3232< 11)
zek30!uucp (12/5-11:57:16) (C,3329,0) FAILED (call to zek30 )
bluhil!uucp (12/5-11:57:31) (C,3329,0) WRONG TIME TO CALL (bluhil)
bluhil!uucp (12/5-11:57:31) (C,3329,0) FAILED (call to bluhil )
usavax!uucp (12/5-11:57:43) (C,3329,0) WRONG TIME TO CALL (usavax)
usavax!uucp (12/5-11:57:43) (C,3329,0) FAILED (call to usavax )
unix!uucp (12/5-11:57:48) (C,3329,0) NO CALL (MAX RECALLS)
zekisv!al (12/5-10:17:50) (X,4193,0) XQT QUE'D (rmail brian )
zekisv!uucp (12/5-10:18:08) (C,4200,0) WRONG TIME TO CALL (zekisv)
zekisv!uucp (12/5-10:18:08) (C,4200,0) FAILED (call to zekisv )
```

7.5.3 Diagnostic Messages From LOGFILE. The messages most commonly contained in the status and further information fields of the **LOGFILE** entries are listed below, along with brief explanations. Suggested corrective actions are provided where applicable.

SUCCEEDED (call to *system*)
A call from the local system to *system* succeeded.

FAILED (call to *system*) or (DIAL cu*nnnn*...)
A call from the local system to *system* did not succeed. Possible reasons for the

failure are:

1. Phone was busy, not answered, or answered by a person instead of a modem.
2. The communication pathway between the two systems was dissolved.
3. The remote system was shut down.

FAILED (LOGIN)

The remote system answered the call from the local system, but the local system could not login. Possible reasons are:

1. The local system has incorrect login or password information.
2. The systems are unable to negotiate a common baud rate (line speed).
3. Excessive noise on the line.
4. A remote modem answered but the remote system was down.

TIMEOUT (LOGIN)

The remote system was probably heavily loaded and took too long to respond.

NO CALL (MAX RECALLS) or **(RETRY TIME NOT REACHED)**

The first time the local system fails to connect with a remote system, the uucp job remains queued in the spool directory and an **STST** file is created. The file inhibits calls to the remote system for a some period of time defined by the system administrator. (The time variable is located in the *time* field of the **L.sys** file. Refer to paragraph 7.10 for further information.) If a job for the remote is sent to the spool directory while the **STST** file is "active", the message **NO CALL (RETRY TIME NOT REACHED)** appears in the log. When the time period is up, the local system attempts to execute the queued job. If the second call fails, the **STST** file increments its record of failed attempts. If the number of failed attempts is less than the number defined as "maximum" by the administrator, the **STST** delays the call for the specified time. If the number of failures reaches the "maximum", the **STST** file causes the **NO CALL (MAX RECALLS)** message to appear. No calls will be attempted until the **STST** file is removed, either manually or by the uucp housekeeping programs.

The **STST** files reduce system load and high phone bills that could be incurred if numerous calls were made, and aborted, before correcting a problem.

To initiate a conversation before the required waiting time has elapsed, first check that no lock file for the remote system exists (ensuring that no conversation is in progress). Then, remove the **STST** file and re-enter your request.

WRONG TIME TO CALL (*system*)

According to the information in your system's **L.sys** file, the remote system has limited the hours during which incoming calls can be received. Your job will be queued in the spool directory until the time restriction is lifted.

(NOTE: This message is always true when you send a uucp job to the spool directory and your system is a polled system. The **L.sys** file, which lists all remote systems to which you can direct jobs, includes an entry that defines the correct time for your system to call the remote. For a polled system, the **L.sys** file entry makes it the "wrong time to call" all the time. The effect is that your uucp jobs are accepted and queued in the spool directory because they are directed to "known" systems, but calls are never made. Instead, jobs remain queued until the active system calls your local system.)

/usr/spool/uucp/LCK. *xxxx* (CAN'T LOCK)

Whenever a conversation is in progress between the local system and a remote system, a lock file is created to prevent another line from initiating a second conversation with the same remote system. This message tells you a conversation is in progress with the remote system (or device) *xxxx* and the request will be re-tried later.

Occasionally, a conversation aborts and leaves a lock file lying around. Execute the command **ps -aef** to check if a *uucico* process is using a line to the remote system. (The *uucico* process indicates a conversation is taking place. For further information on *uucico*, refer to paragraph 7.3.) After you verify no conversation is taking place, remove the lock file and re-enter your request.

NO (DEVICE)

An attempt to call the remote system was made but there was no modem (or dial-up line) available. For a direct connection, this message is received when the hardwired line is in use.

REQUESTED (CY) or **(CY5)**

The request succeeded if **CY** is in the additional information field, or was successful in a limited sense if the field contains **CY5**. The **CY5** diagnostic generally means that the file transfer did not succeed as requested, but the file was copied into the remote system's public directory by default. The file is probably located in a subdirectory of the public directory; the subdirectory is the name of the account or directory to which the transfer was initially intended.

REQUEST (S *local-sequence-no remote-sequence-no*)

A file transfer requested by *system*!*user-id* to copy the local file to the remote system succeeeded.

REQUEST (R *remote-sequence-no local-sequence-no*)

A file transfer requested by *system*!*user-id* to fetch the remote file and copy it to the local system succeeded.

COPY (SUCCEEDED)

A file transfer to this system succeeded.

BAD READ (*further information***)**

The system gave up on the transfer because of too many errors. Data lost in transmission was not recovered. Typically, the conversation would have terminated prematurely.

user-id **XQT (PATH=/bin:/usr/bin;** *export PATH command*)

A user requested that a command be executed and the execution was attempted. Usually, the command attempted is **rmail**. The *user-id* is shown as **uucp** if the mail was forwarded through a remote system to arrive at the local system.

user-id **XQT DENIED (***command***)**

A user request for a remote execution failed because permission for the execution was denied. This message results when a requested command is not on the authorized list contained in the remote system's **L.cmds** file.

QUE'D (*more information***)**

A *uucp* command request has been queued for file transfer with the *system*. The request is stored in the spool file **/usr/spool/uucp** or the specific file indicated in the information field.

7.5.4 Job ID Numbers and uustat. *Uustat*(1C) displays the status of previously specified uucp commands, cancels a command, or provides general status on connections to other systems. One option for specifying a particular uucp command is to identify the command by its job number.

Although job numbers are not printed by default, they can be obtained in one of three ways. Setting the environmental variable **JOBNO=ON**, and exporting the variable causes job numbers to be printed for all **uucp** and **uux** commands. To force job numbers to be printed for specific **uucp** and **uux** commands, execute the command with the **—j** option invoked. For example, the command

uucp —j /etc/passwd winkil!/dev/null

returns the message

uucp job 282

on the standard output. Finally, a list of all uucp commands and their job numbers can be obtained by using the **uustat** command without any options invoked:

uustat

Uustat returns a one-line entry for each job in progress. The format of each line is:

job-number user remote-system command-time status-time status

All status messages that users may be concerned with are self-explanatory. Refer to the *SYSTEM V/68 User's Manual* for a complete listing of all status messages and their meanings.

7.5.5 Job Termination, Requeuing. A job that transfers many files from several different systems can be killed with the *uustat* **—k** option. If any part of the job has left the system, only the remaining parts of the job on the local system are terminated. (Note that while the **—k** option of *uustat* kills a job, it does not update the *uustat* listing. The job continues to appear for a short period of time.)

The uucp housekeeping program, *uuclean*(1M), removes undeliverable jobs from **/usr/spool/uucp** on a regular basis (usually every 72 hours). Jobs can be "rejuvenated" with the **—r** option of *uustat*, which changes the job date to the current date.

(For further information on the uucp housekeeping programs, *uuclean, uudemon.hr, uudemon.day, and uudemon.wk*, refer to paragraph 7.9)

7.6 The Uucp User's Network: Usenet

Usenet, informally referred to as "the net", is a bulletin board shared among thousands of UNIX-based systems and tens of thousands of users in the United States, Canada, Europe, Australia and Japan. The bulletin board contains more than 140 categories of information; Usenet readers post articles and notices to the categories that interest them.

There is no typical Usenet reader. Many users are casual readers who occasionally scan one or two of the categories, called newsgroups. Other readers actively participate in on-going debates concerning UNIX intricacies, international politics and the benefits of different microcomputers. On the less serious end of the spectrum are the newsgroups for chess, ham radios, science fiction, and the world-famous **jokes** newsgroup. Because bulletin board communication is a unique hybrid, less formal than trade publications but far more widely distributed than in-house memos, a network etiquette has evolved with which new readers would do well to familiarize themselves. The "Emily Post for Usenet" is one of the first articles new net subscribers receive.

The size of the Usenet network is the key to its usefulness. A user wishing to announce a new program, solve a problem or locate a piece of equipment can reach a mass audience in a matter of days. Discussions involving experts with a wide range of experience occur without a formal meeting. However, the heavy flow of communication over the network means a person could easily spend several hours a day "reading the news." A site that subscribes to every newsgroup can expect to receive an average of 360 articles containing 530,000 bytes of data each day. Because of the number of programs, control files and directories required to move, sort and store this information, Usenet is not easy to install.

Usenet uses the *uucp* remote execution facilities to send information to receiving sites. The decision whether or not to subscribe to the network is made by the system administrator at each site. To join Usenet, the system administrator should contact the nearest UNIX-based installation to discover who among the local systems is connected to the network and is willing to allow another system to connect to it as a known system. In theory, any system can connect to any other, regardless of the distance between the systems, but economic considerations (e.g., the cost of forwarding mail) suggest local connections are best. Usenet software is distributed freely over the network and can be obtained from a Usenet site near you. Included in the software are articles that provide extensive documentation for posting and reading articles, and selecting the categories to which users may have access. The software is not distributed through Motorola Microsystems.

7.7 Administrator's Overview of Uucp Programs

7.7.1 Background. Beginning with this section, the tutorial addresses an audience of system administrators. Casual system users may end the tutorial here. All the knowledge needed to use the uucp programs effectively is contained in the first six sections.

System administrators are strongly urged to read the tutorial straight through. Do not start with this section. Background information describing uucp applications, system links, security concerns, naming conventions, and network architecture is covered in sections 7.1 through 7.6 of the tutorial; it is not repeated here. Further, users expect detailed information from their administrator describing their system links and approved applications. Administrators may have an easier time designing uucp maps and anticipating user questions if they have an understanding of their users' perception of the network.

Administrators should have a thorough understanding of the *SYSTEM V/68 User's Manual* and *SYSTEM V/68 Administrator's Manual* pages referring to uucp commands. Particular attention is directed to the following commands: *uucp*(1C), *uustat*(1C), *uuto*(1C), *uux*(1C),

uuclean(1M), and *uusub*(1M).

7.7.2 uucp Subdirectory. Most of the uucp programs reside in **/usr/bin.** However, other files required for uucp operations are located in **/dev**, **/etc**, **/usr/lib**, and **/usr/spool.** An expanded version of Figure 7-2, "Branch Directory of uucp Directories and Files", is illustrated in Figure 7-4. Table 7-1 lists the name and suggested permission mode for each file read, executed, or created during uucp execution. Detailed information about the permission modes and system security is addressed in paragraph 7.8, "Administrative Concerns". Detailed information about the maintenance and administrative programs *uuclean*, *uulog*, *uustat* and *uusub*, and the *cron* files and daemons, is included in paragraph 7.9, "Maintenance and Administration". Detailed information about the files in **/usr/lib/uucp** is included in paragraph 7.10, "Installation".

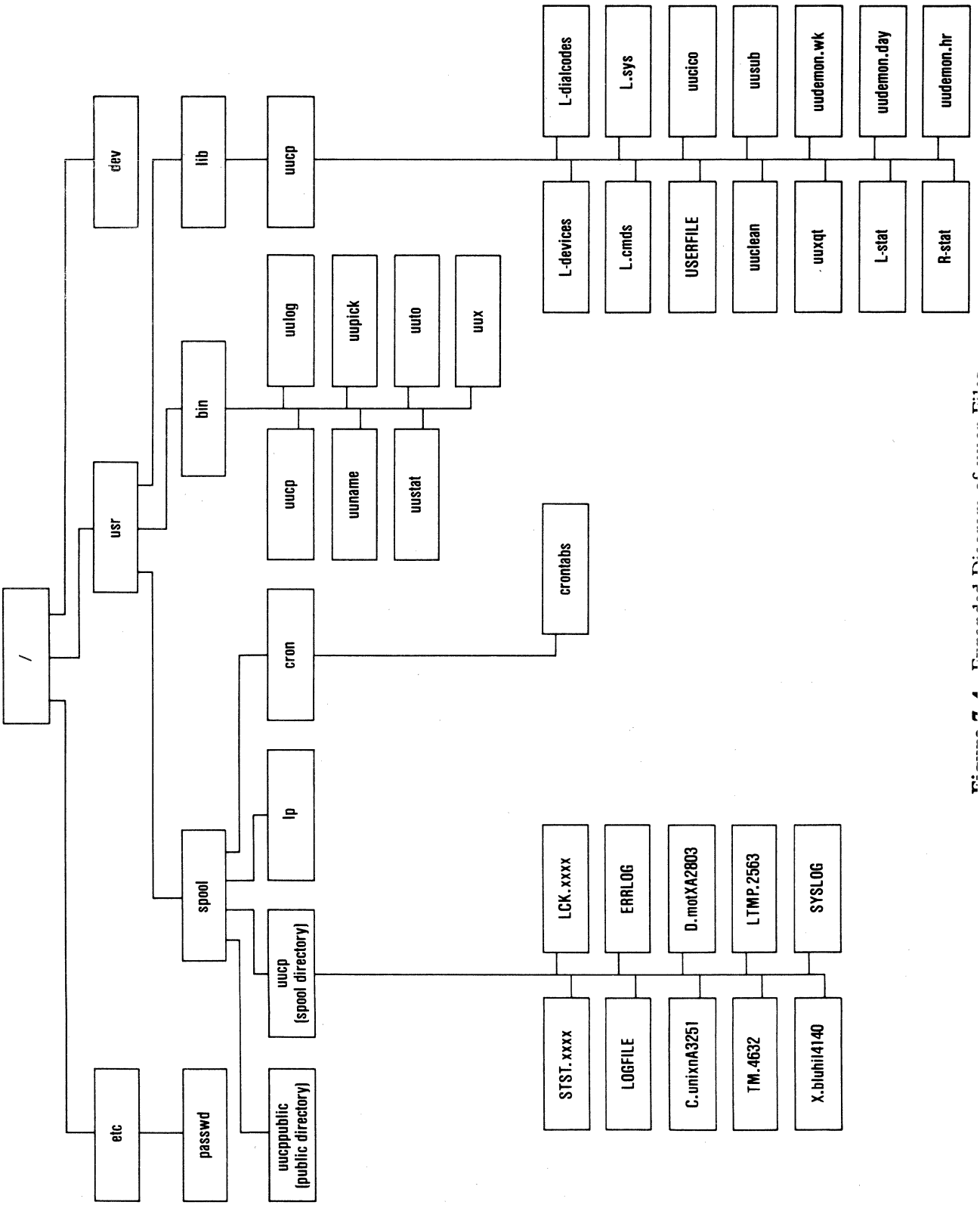
/
dev
etc
passwd
usr
bin
uucp
uulog
uupick
uuto
uux
uucp
uuname
uustat
lib
uucp
L-dialcodes
L.sys
uucico
uusub
uudemon.wk
uudemon.day
uudemon.hr
L-devices
L.cmds
USERFILE
uuclean
uuxqt
L-stat
R-stat
crontabs
cron
spool
lp
uucp
(spool directory)
uucppublic
(public directory)
LCK.xxxx
ERRLOG
D.motXA2803
LTMP.2563
SYSLOG
STST.xxxx
LOGFILE
C.unixnA3251
TM.4632
X.bluhil4140

Table 7-1. Permission Modes for Uucp Program Files

/bin						
-rwxr-sr-x	2 bin	mail	25576	Aug 13	12:04	mail
-rwxr-sr-x	2 bin	mail	25576	Aug 13	12:04	rmail
/dev						
crw-rw-rw-	2 root	sys	3,2	Jan 14	11:05	null
crw-rw-rw-	2 root	sys	7, 0	Dec 21	09:06	cua0
crw-rw-rw-	2 root	sys	44, 14	Dec 20	19:04	cu10
/etc/passwd						
-rw-r--r--	1 root	other	1770	Dec 19	11:30	group
-r--r--r--	1 root	sys	137170	Dec 19	08:42	passwd
/usr/bin						
---s--x--x	1 uucp	bin	38070	Oct 8	13:13	uucp
---s--x--x	1 uucp	bin	20554	Oct 8	13:13	uulog
---s--x--x	1 uucp	bin	15436	Oct 8	13:13	uuname
-rwxrwxr-x	1 bin	bin	1595	Oct 8	13:13	uupick
---s--x--x	1 uucp	bin	37728	Oct 8	13:13	uustat
-rwxrwxr-x	1 bin	bin	983	Oct 8	13:13	uuto
---s--x--x	1 uucp	bin	38896	Oct 8	13:13	uux
/usr/lib/uucp						
-r--r--r--	1 uucp	bin	126	Aug 8	09:46	L-devices
-rw-r--r--	1 uucp	bin	24	Dec 1	19:82	L-dialcodes
-r--r--r--	1 uucp	other	34	Oct 14	08:41	L.cmds
-r--------	1 uucp	other	1936	Nov 20	15:47	L.sys
-rw-rw-rw-	1 uucp	bin	4	Dec 21	09:08	SEQF
-r--r--r--	1 uucp	other	675	Dec 5	14:08	USERFILE
---s--x--x	1 uucp	bin	69364	Oct 8	13:17	uucico
---s--x--x	1 uucp	bin	26300	Oct 8	13:17	uuclean
-rwxr-xr-x	1 uucp	bin	399	Aug13	11:22	uudemon.day
-rwxr-xr-x	1 uucp	bin	139	Aug13	11:22	uudemon.hr
-rwxr-xr-x	1 uucp	bin	279	Aug13	11:22	uudemon.wk
---x-----	1 uucp	bin	23332	Oct 8	13:17	uusub
---s--x--x	1 uucp	bin	36798	Oct 8	13:17	uuxqt
/usr/spool/uucp						
-rw-------	1 uucp	other	0	Dec 21	08:57	AUDIT
-rw-rw-rw-	1 uucp	uucp	13849	Dec 21	09:11	LOGFILE
-rw-rw-rw-	1 uucp	uucp	428316	Dec 21	09:11	SYSLOG
/usr/spool/uucppublic						
drwxrwxrwx	50 root	other	800	Dec 20	11:00	receive
drwxrwxrwx	2 root	other	432	Dec 14	04:02	root

Note: The owner "uucp" referred to in this listing is the uucp administrative login, not the remote site.

7.7.3 UNIX-to-UNIX CoPy Operation. The work done by the commands **uucp, uux, uuto, mail,** and **rmail**, can be divided into two parts:

classification, performed by *uucp* and *uux*;
and
execution, performed by *uucico* and *uuxqt*.

For **uucp** and **uuto** commands, the *uucp* program classifies the work and sets up the files for transfer. Next, the *uucico* program takes over and executes the transfer. For **uux, mail** and **rmail** commands (that are executed over the network), the *uux* program classifies the work and sets up the files for command execution. Next, the *uuxqt* program takes over and executes the command.

7.7.3.1 uucp Command Execution. When a **uucp** command is received, the *uucp* program classifies the type of work requested. Five classifications are defined:

1. Copy a local file to a directory on the local system
2. Receive a file into a local directory from a remote system
3. Send a file from the local system to a remote system
4. Send a file from one remote system to another remote system
5. Receive a file from a remote system, where the filename specified contains a special shell character.

Once the request is defined by type, *uucp* creates files in the spool directory that contain the information needed by *uucico* to execute the request. To set up the file transfer, *uucp* creates two kinds of special files: data files and work files. Data files contain a copy of the source file for transfer to remote systems. Work files contain the detailed coordination information needed to perform a file transfer between systems.

(Different types of work need different information available in their work files. The work files created in the spool directory can be distinguished by naming conventions. A definition of each work file type and its contents is provided in paragraph 7.7.3.4.)

As part of the file creation process, the *uucp* program starts the *uucico* program. *Uucico* scans for work by looking in the spool directory for work files, then begins the execution.

For example, the local system receives the command

uucp autobio grtbks!~/crews

to send the file **autobio** on the local system to the public directory on the remote system, **grtbks**. *Uucp* classifies the work as Type 3, "send local file to remote." In response, *uucp* copies the source file into a data file, and writes directions for the transfer into a (Type 3) work file.

The second part of the work process is the command execution. To continue with the example, *uucp* starts the *uucico* program. *Uucico* finds the (Type 3) work file in the spool directory and initiates a call to the remote system. The connection is made, the systems agree on a protocol, and the file is sent. When the transfer is complete, *uucico* receives any jobs waiting at the remote site. Individual log files are created for each transaction, and a record of each step in the execution process is compiled in **/usr/spool/uucp/LOGFILE**. When the transfer is complete, the *uucico* program moves on to the next work file waiting in the spool directory.

If the **uucp** command is the Type 2 request, a copy of a remote file must be received by the local system. To initiate a copy at the remote system, the *uucp* program generates a workfile and sends it to the remote machine's spool directory. The remote system's *uucico* program finds the work file and executes the request. Type 4 and 5 requests are initiated by the local

system generating its own *uucp* command and sending it to the remote. The remote machine receives the command, organizes the work into data and (Type 3) work files, and executes.

7.7.3.2 uux Command Execution. **Uux** command execution follows the same two-step process as *uucp* execution: classify the work by creating special files, then execution. When a **uux** command is received, *uux* creates an execute file in the spool directory that contains the information needed by *uuxqt* to process the request.

The execute file contains several lines, each of which begins with an identifying character and one or more arguments. The execute file describes each of the files needed for execution (by both the real filename, and the data or work filename as found in the spool directory), and the command to be executed (e.g., **ls -l**, **who**, or **diff**). (Refer to paragraph 7.7.3.5 for a line-by-line definition of the execute file contents.)

The execute file is moved into the spool directory for local execution, or is sent to the remote system if the execution is to take place on another system. If *uux* sees that some of the files required for execution are located on remote systems, the program generates a work file like the ones created by *uucp*. The work files are sent to the remote system's spool directory for execution by the remote's *uucico* program.

When all files are available, the *uuxqt* program on the executing machine processes the execute file. All required files are moved into an execution subdirectory of the spool directory, and the particular command is executed using the shell specified in the *uucp.h* header file.

Uuxqt processing involves executing a **sh** **—c** of the command line after appropriate standard input and output have been opened. After processing, the command output is copied or readied to be sent to the designated standard output.

These paragraphs describe the basic operation of the major uucp programs: *uucp*, *uucico*, *uux*, and *uuxqt*. However, these operations only occur if the local system has permission to place work files and execute files in the spool directory of another system. The accessibility of the remote system's spool directory remains under the control of the remote system's **USERFILE**. When access is denied, the transfer, or remote execution, fails.

7.7.3.3 Spool Directory Filenames. The spool directory files share a naming convention:

type . *system-name* *grade* *number*

where

type Is an uppercase letter:
C indicates a work file
D indicates a data file
X indicates an execute file

system-name Refers to the remote system

grade Is a single character

number Is a four-digit, padded sequence number

No spaces separate the fields in the filename. For example, the following are spool directory filenames:

D.winkilA0325
C.kulrunC0328
X.rivwhyX0045

7.7.3.4 uucp-Created Files: Data Files, Work Files. *Uucp* creates data files and work files in the spool directory to organize the processing done by *uucico*. The number and type of files created depends on the kind of work requested by the **uucp** command received by the system. As mentioned in the previous paragraph, **uucp** commands can be classified into five types of requests:

1. Copy a file to a directory on the local system

   The *cp* program does the work for a simple copy request. No data files or work files are created.

   The **uucp** options **—d** (make all necessary directories), **—m** (send mail on completion), and **—n** (send mail to remote user on completion) are not honored for this type of request.

2. Receive a file into a local directory from a remote system

   A "receive" work file is created for each file requested from a remote system. The "receive" work file contains a single line of five fields that follows the format:

   R *source-pathname destination-pathname user's-login —options*

   where

   R
   Indicates the "receive" type work file

   source-pathname
   is either the full pathname of the source file on the remote, or the pathname expressed as *~user/pathname*. The *~user* pathname notation is expanded on the remote system.

   destination-pathname
   is either the full pathname of the destination file on the local system, or the pathname expressed as *~user*. The *~user* notation is immediately expanded to the user's login directory.

   —options
   lists the requested options.

3. Send a file from the local system to a remote system.

   A "send" work file is created for each source file. The source file is copied into a data file, unless the **uucp** option **—c** was invoked. (The **—c** option directs *uucp* to copy the source file when the transfer takes place.) The "send" work file contains one line with seven fields, following the format:

   S *source-path dest-path user-id —options data file source-mode*

   where

   S
   indicates a "send" type work file

   source-path
   is the full pathname of the source file

dest-path
is the full pathname to the destination file, or the abbreviated notation, *~/user/filename.*

—options
lists the requested options

data-file
is the name of the data file in the spool directory that contains a copy of the source file

source-mode
is the file mode of the source file, expressed as an octal number (e.g., 0664).

4. Send a file from one remote system to another remote system.
5. Receive a file from a remote system, where the filename specified contains a special shell character.

The last two work types both require a remote system to initiate a copy on behalf of the requesting system. Processing involves sending a **uucp** command to the remote system containing the source file. The remote system then processes the **uucp** command for the sending system. If the remote system does not include **uucp** as an executable command in its **L.cmds** file, the work requests will fail. (Transferring files through an intermediary system with *uucp* does not require a uucp entry in the **L.cmds** file since the remote systems do not initiate the copy; the copied file arrives in the remote system's public directory.)

7.7.3.5 uux-Created Execute Files. Execute files are created by *uux* and processed by *uuxqt* on the execution machine. Although the number of lines in the file may vary, each line begins with an identification character and contains one or more arguments. The order in which the lines occur in the file is not significant. The line identification characters and the argument definitions for each line are:

U *user system*

This line defines the requestor's system and login name.

F *generated-filename real-filename*

Every file needed for the command execution must appear in an **F**-line, one file per **F**-line. The generated filename refers to the name of the file as it is stored in the spool directory, either as a data file, or a file received from another system. The real filename is the file portion of the filename, without any path information. If no files are required for execution, no **F**-lines are included in the execute file.

I *filename*

This line refers to the standard input. Standard input is designated in the **uux** command with a < symbol, or from the — option, which specifies that standard input for the commmand-string is inherited from the standard input of the **uux** command. If no standard input is specified, **/dev/null** is used. Notice that if a standard input is specified, it also appears as a required file in an **F**-line.

O *filename system-name*

This line refers to the standard output. Standard output is designated in the **uux** command with a >. If standard output is not specified, **/dev/null** is assumed. The use of >> to append the output to another file is not implemented.

C *command* [*arguments*]
The arguments included in this line are the arguments specified in the command.

7.7.4 uucico Processing. The *uucico* program processes the spool directory files created by the *uucp* program. *Uucico* may be started in any of four ways:

- By a local system daemon, such as the cron daemon
- By a uucp program
- By the system administrator (usually for debugging)
- By a remote system (if the local system permits)

When *uucico* is invoked by the remote system, it is said to be operating in "slave" mode. When started by any other means, *uucico* operates in "master" mode. When *uucico* is started, several options may be invoked:

—**r1** Start the program in master mode. When *uucico* is initiated by a program or a daemon, the option is invoked. (The 1 is the numeral 1.)

—**s***sys* Do work for system *sys* only. When specified, *uucico* initiates a call to system *sys* regardless of whether or not work is found in the spool directory. The option may be used to poll a remote system.

—**d***dir* Use the specified directory as the spool directory. This option has applications for debugging. Refer to paragraph 7.11 for further information.

—**x***num* Specifies a debugging level between 1 and 9. Refer to paragraph 7.11 for further information.

Uucico processing performs five specific functions:

- Searches spool directory for work
- Initiates a call to the remote system
- Negotiates an acceptable line protocol
- Executes all requests from calling system and from the remote system
- Logs all transactions

Each processing task is described in the paragraphs that follow.

7.7.4.1 Locate Work. Once started, *uucico* searches the spool directory for work files, designated by a filename that begins with **C .** A list of systems to be called is drawn up from the information contained in the work files. A random selection determines which will be the first system called.

7.7.4.2 Contact Remote System. Before *uucico* calls another system, several files are consulted to obtain information. The selected system, as named by a work file, is checked against the system names contained in **L.sys**. From the **L.sys** file, *uucico* obtains calling information, including:

System name

Times when calls are accepted (by day and hour)

Device type needed

Line speed or class

Phone number (if an ACU is used) or device name (if no ACU is used)

Login id and remote system password

If the phone number in **L.sys** contains an abbreviation, *uucico* reads **L-dialcodes** and interprets the abbreviations as dialing sequences.

Next, *uucico* scans the **L-devices** file to find a device that meets the requirements specified in **L.sys.**

After the calling system has logged into the remote, both systems identify themselves with their unique system-names. Each system consults its **USERFILE** to obtain access information.

7.7.4.3 Select Line Protocol. The slave system, the system called, initiates a handshake to begin the conversation. The slave tells the master system that it is ready to receive the master system identification and the conversation sequence number. The master sends its response. The slave system may now reply that a "call-back" is required, and the current conversation is terminated. Alternatively, the slave may verify the master system response and, if it is accepted, begin protocol selection.

Protocol selection begins with a message from the slave system that takes the form:

P*protocol-list*

where the *protocol-list* is a string of characters, each of which represents a line protocol. The master system checks the list for a character corresponding to an available line protocol and returns a message of the form:

U*code*

where the *code* is a one-character protocol letter, indicating a common protocol, or an **N**, indicating there is no available protocol common to both systems.

7.7.4.4 Execute Requests. Once a protocol is selected, the master system begins sending messages to the slave system. Five types of messages are sent; each message is recognized by the first character of the message:

S Send a file

R Receive a file

C Copy complete

X Execute a uucp command

H Hang up.

For each message sent by the master system, the slave system responds with a message of its own. The slave system messages tell the master if it has permission (as outlined in the slave system's **USERFILE**) to access the file or directory requested; and, having crossed the first hurdle, if the file and directory read and write permissions are set so that the work can proceed. The slave system response to a send message from the master system is either **SY** or **SN**, corresponding to "Yes, work can proceed" or "No, work cannot proceed." The pattern is the same for all messages from the master: **RY** or **RN**, **XY** or **XN**, **HY** or **HN**.

After a file is copied into the spool directory of the slave system, the slave sends a copy complete message. The **CY** indicates the copy was completely successful, the file is at the specified destination. A **CY5** message usually means the transfer was only partially successful; the file may be located in the spool directory of the slave system with a filename beginning with **TM.**

Some diagnostic information is available from the slave system responses. For example, a **CN5** response to a copy request usually indicates the slave system ran into trouble (typically permission problems) while trying to transfer the file into the destination file. An **SN4** response from the slave to the master system's send request usually means the request is denied because the slave cannot create the required work files. An **SN2** response indicates the target file cannot be created, typically because of permission restrictions. In response to a receive request from a master system, an **RN2** response indicates the slave system can't find the file, or cannot send it, typically, because of permission problems.

When the master system completes processing its list of work files for the slave system, it sends an **H** request to slave: Ready to Hang Up? The slave system searches its own spool directory to see if it has any work waiting for the master system. If it finds a work file that requires processing, a **HN** message is sent and the programs switch roles. Master becomes slave, slave becomes master. When the new master system completes its processing, it now sends a **H** message to the new slave. A scan of the spool directory is made and if no work files are found, a **HY** reply is made. When a **HY** reply is received by the master, it is echoed back to the slave system and the protocols are turned off. Each machine sends a final **00** message to the other. The original slave system cleans up any left files and terminates. The original master system continues processing and starts to gather the information needed to contact the next system on its list of work files.

7.7.4.5 Log Transactions. Throughout the conversation between the two systems, each request and its result are logged on both systems in **LOGFILE.** To watch the transactions as they occur, enter the command

tail -f /usr/lib/uucp/LOGFILE

To exit the loop, press the Interrupt key.

7.8 Administrative Concerns

Administrative concerns about the impact of the network on system operations generally fall into three areas:

- System security
- Interconnection methods
- Administrative workload

Each of these areas is addressed in this section.

7.8.1 System Security. The security threat represented by the uucp network to any system can be significant. Any command that is executable by a particular remote system user, or any file that can be copied by a particular remote system user, is, potentially, available to every user on the uucp network. Bugs exist in the programs; malevolent users can disguise their user-ids. Local users may become lax about protecting sensitive information.

However, security features can be installed that reinforce and supersede the normal file protections available through **USERFILE**, **/etc/passwd**, and **L.sys**. (Refer to paragraph 7.10, "Installation".) A list of recommended precautions is presented below. Administrators are encouraged to read through the list carefully and adopt the ones best suited to their site.

Suggested security measures that can be implemented by system administrators when uucp is installed are as follows:

1. The owner of the uucp programs should be a unique administrative login. Make this login different from the login used for remote system access to the uucp programs.
2. The programs *uucp*, *uucico*, *uux*, *uuxqt*, *uulog*, and *uuclean* should be owned by the uucp administrative login. The *setuid* bit should be set, and the programs' file mode permissions should be "execute" only.
3. The login for uucp programs should not gain access to a standard shell. Instead, start the *uucico* program so that all work must be done through *uucico*.
4. A path check should be done on the name of every file that is to be sent or received as part of a uucp program command. The **USERFILE** can be set up to require call-back for certain login-ids. (The call-back requires that both systems have "dial-out" and "dial-in" modems, or are hardwire connected.)
5. A conversation sequence count can be established between systems so that the slave system can be more confident that the calling system is who it says it is. (If a system goes down during a conversation, it can lose the sequence count. If this happens, calls will fail until the count is corrected manually by the administrator.)
6. The *uuxqt* program is installed with a list of executable commands. A "PATH" shell statement is added to the beginning of the command line according to instructions specified in the *uuxqt* program. The installer has the option of modifying the list of executable commands, or changing the restricted PATH, as desired.
7. The file **L.sys** should be owned by the uucp administrative login. In addition, the file should have permission mode 0400, owner read-only, to protect the remote sites' login information.
8. Establish a dial-up password by creating files **/etc/dialups** and **/etc/d_passwd**. Creating these files must be done carefully. The procedure is outlined in the *SYSTEM V/68 Administrator's Guide*.

From a corporate perspective, the uucp network news might represent a security problem for product information. Any information posted on the Usenet bulletin board is considered "public knowledge" and, as such, cannot be protected by trade secret agreements.

7.8.2 Interconnection Methods. Two means of interconnection are supported by the uucp programs:

Direct connection using a null modem

Connection over the Direct Distance Dialing (DDD) network

SYSTEM V/68, Release 2.1 supports auto-dial modems ("smart" modems) for access to the DDD network; it does not support automatic calling units (acus).

In choosing hardware, the equipment used by other processors on the network must be considered. For example, most data sets available on systems are 1200-baud data sets. If a system on the network has only 103-type (300 baud) data sets, communication with them is not possible unless the local system connects a 300-baud data set to a calling unit.

If hard-wired connects are used between systems, the distance between the systems is critical. A null modem cannot be used when the systems are separated by more than several hundred feet.

7.8.3 Administrative Workload. Overall, the impact of uucp on an administrator's time is minimal. The amount of added work depends on the size of your system and the number of users you have. In practice, bookkeeping chores may be the greatest annoyance. Passwords, telephone numbers and logins change frequently on the network, and, if an administrator does not keep the system files up to date, data lines can be tied up trying and re-trying to make connections with bad information.

A second source of extra work is Usenet. If a site signs up to receive articles for all newsgroups, an average of 530,000 bytes of data must be processed daily. The installation of Usenet is a one-time chore, but it is time-consuming, none the less.

7.9 Maintenance and Administration

Most of the work involved in managing and maintaining uucp revolves around cleanup. Some jobs use *cron*(1M) to start shell scripts and shell files which call up the *uuclean*, *uulog*, and *uusub* programs. Direct intervention by the administrator is required for other jobs. The maintenance programs are:

uuclean:

A program to purge old and undeliverable jobs

uulog:

A program to merge individual **LOG** files into the current **LOGFILE**

uusub:

A program to monitor the connections and traffic among the members of a defined uucp subnetwork

uudemon.day:

A program invoked daily by the *cron* daemon to perform uucp administration and maintenance. Similar programs, *uudemon.hr* and *uudemon.wk* are also distributed with this release.

The files that need attention are:

/usr/spool/uucp/ERRLOG: The uucp system error file

/usr/spool/uucp/LCK. *.xxxx*: Lock files

/usr/spool/uucp/LOG.*xxxx*: Individual log files

/usr/lib/uucp/*SQFILE*: Sequence check file

/usr/spool/uucp/STST.*xxxx*: Failure log file

/usr/spool/uucp/TM.*xxxx*: Temporary files created during an unsuccessful transfer

7.9.1 Maintenance Using cron. *Cron*, the clock daemon, reads entries in the file **/usr/spool/cron/crontabs/uucp** and executes shell scripts and shell files at the dates and times specified. *Cron* is recommended for the performance of several routine uucp tasks.

First, the *uucp* program spools work and attempts to start the *uucico* program each time a **uucp** command is entered. Occasionally, *uucico* fails to start. The **crontab** file should contain an entry to start the *uucico* program in master mode on an hourly basis.

Second, individual **LOG** files are created in the spool directory during uucp program execution. The files contain information about queued requests, calls to remote systems, command executions, and file copy results. These individual files should be merged periodically with **/usr/spool/uucp/LOGFILE** by executing the *uulog* program.

Third, *cron* is extremely useful for managing the routine cleanup of **TM** (temporary), **STST** (failure report), and **LCK** (lock) files that accumulate in the spool directory if failures occur during a conversation. Work files and data files that cannot be executed because of bad telephone numbers or obsolete login information also must be cleaned out after a reasonable period of time. Shell files that are executed on an hourly, daily, and/or weekly basis contain *uuclean* commands with options that remove files after a specified length of time. For example, a shell file executed hourly might contain the invocation

/usr/lib/uucp/uuclean -pST -pC. -n48

to remove work files and old status files older than 48 hours.

(Refer also to the files **uudemon.day**, **uudemon.hr**, and **uudemon.wk** in **/usr/lib/uucp.**)

Fourth, an entry in **crontab** should initiate a daily or weekly procedure to save or remove old copies of **LOGFILE**.

Fifth, an administrator may want statistics monitoring the traffic between specific systems. The *uusub* program provides the number of files and bytes sent to a particular system over a specified period. The program can be initiated daily by placing the command

uusub -c all -u 24

in the **crontab** file.

7.9.2 uucp File Maintenance. This paragraph contains background information about uucp-associated files that require maintenance.

7.9.2.1 ERRLOG. **ERRLOG** records uucp system errors in the spool directory. Entries rarely should be found in this file. **ERRLOG** reports incorrect modes on required files or directories, missing files, and read/write system call failures on the transmission channel.

7.9.2.2 LCK.xxxx. Lock files may be left in the spool directory if runs abort. Lock files are ignored after 24 hours. To call before the 24 hours elapse, remove the lock file with an **rm** command.

7.9.2.3 LOG. The **LOG** files are created with permission mode 0222. If the program that created the file terminates normally, the **LOG** file permission mode changes to 0666. Aborted runs may leave the files with permission mode 0222, and *uulog* will be unable to read, merge and remove the files. The administrator must intervene with a **rm** command, or change the permission mode to 0666 and wait for the *uulog* program to do the merge.

7.9.2.4 Sequence Check File. Each remote system that performs conversation sequence checks has an entry in the **SEQF** file. Initially, the entry contains only the system-name. The first conversation adds two items to the entry line: the conversation count, and the date and time of the most recent conversation. Each conversation updates these entries. If a sequence check between the two systems fails, the administrator must intervene manually to correct the entries.

7.9.2.5 STST.xxxx When a call to a system fails, the **STST** file forces a delay before the local system tries the call again. The **STST** files are created following an "ordinary" failure, such as busy devices or dial-up or login misinformation. **STST** files contain a "talking" status when two systems are communicating. If an abort occurs during a conversation, an **STST** file containing a "talking" status may be left in the spool directory indefinitely. The administrator must intervene to remove this file before another conversation can be attempted.

7.9.2.6 TM. **TM** files are temporary files created in the spool directory while files are being copied from a remote system. The filenames follow the format:

TM.*pid.nnn*

where *pid* is a process-id, and *nnn* is a sequential, three-digit number that is reset at zero for each invocation of *uucico* and incremented for each file received.

After the remote file is received, the **TM** file is moved or copied to the destination directory on the local system. If processing terminates prematurely, or if the final move or copy command fails, the **TM** file remains in the spool directory until it is removed by *uuclean*.

7.10 Installation

After planning for security and system applications, administrators define the extent of their system's interaction with the network by making the appropriate entries in the following object files:

/etc/passwd

/usr/lib/uucp/L.sys

/usr/lib/uucp/L-devices

/usr/lib/uucp/L-dialcodes

/usr/lib/uucp/L.cmds

/usr/lib/uucp/USERFILE

/usr/lib/uucp/FWDFILE, /usr/lib/uucp/ORIGFILE (optional)

A change must also be made to the kernel to define the system's *system-name*.

Administrators with source distribution may want to reconfigure the workings of the uucp programs at some later date. For these administrators, a section is included at the end of this section that provides necessary information on changing the *makefile*, the header file, and compiling the system.

7.10.1 Modifying the Kernel. The operating system kernel must include the *system-name*, or nodename, that will identify your system. The procedure for making this change is described in the following steps.

1. Create a file called **zrp.c** in the directory appropriate to your machine:

   For an EXORmacs: **/usr/src/uts/m68k/M68000/MACScf**

   For a VM03: **/usr/src/uts/m68k/M68010/VM03cf**

   For a VME10: **/usr/src/uts/m68k/M68010/VME10cf**

   For a VME1000: /usr/src/uts/m68k/M68020/VME131cf

   The file should contain:

```
#include <sys/utsname.h>
#define NODENAME    "xyzzy"    /*8 characters max, excluding " marks */
#define MACHINE     "abcdef"   /*6 characters*/
struct utsname utsname = {
        "unix",
        NODENAME,
        "sysV/68",
        "r2v2.1",

};
```

   where "*xyzzy*" refers to a unique nodename for the system and "*abcdef*" refers to the name of the machine for which the kernel is configured, e.g., M68000 or M68010.

2. Type the following commands to install this file:

```
cc -c zrp.c
mv zrp.o name.o
```

If you are using the 68020-based system, skip the rest of this section and refer to the SYSTEM V/68 Operator's Guide, Section 3.2 for further installation instructions.

3. Incorporate this new information into the kernel. To generate a new kernel, a *make*(1) command is executed from inside the build directory.

   cd /usr/src/uts/m68k
   make *kernel*

   where *kernel* is the name of the system kernel. The names of the system kernels and the new kernels that result from the *make*(1) command are listed in the table that follows.

KERNEL NAME	KERNEL PRODUCED BY MAKE COMMAND
macs16	**/usr/src/uts/m68k/M68000/MACS16unix**
macs1622	**/usr/src/uts/m68k/M68000/MACS1622unix**
macs25	**/usr/src/uts/m68k/M68000/MACS25unix**
macs2522	**/usr/src/uts/m68k/M68000/MACS2522unix**
macs80	**/usr/src/uts/m68k/M68000/MACS80unix**
macs8022	**/usr/src/uts/m68k/M68000/MACS8022unix**
vme1015	**/usr/src/uts/m68k/M68010/VME1015unix**
vme1040	**/usr/src/uts/m68k/M68010/VME1040unix**
vm0316	**/usr/src/uts/m68k/M68010/VM0316unix**
vm031622	**/usr/src/uts/m68k/M68010/VM031622unix**
vm0325	**/usr/src/uts/m68k/M68010/VM0325unix**
vm032522	**/usr/src/uts/m68k/M68010/VM032522unix**
vm0380	**/usr/src/uts/m68k/M68010/VM0380unix**
vm038022	**/usr/src/uts/m68k/M68010/VM038022unix**

4. Move the kernel into the **root** directory for testing:

   mv *new_kernel* **/unix.test**
   chmod 0664 /unix.test

5. Move into the **root** directory to perform the test with the command

   cd /

6. Update the "superblock" with *sync*(1). Enter the command **sync** three times.

7. Press the RESET button and boot off the new kernel for testing. Make sure that the system boots normally and simple operations such as changing directories and listing file permissions are working. The **—n /unix.test** option is needed for *ps*(1) and *crash*(1M), and the **—N /unix.test** option is needed for *ipcs*(1) to work properly. (The options are required whenever a non-default kernel is used.)

8. After the kernel has been tested and proven, rename it with the command

   mv /unix.test *final_name*

9. Make the newly created kernel the default kernel. Be aware that once the **ln** command is executed, the previous contents of the files are lost. Therefore, as an extra precaution, some administrators may want to save the old **/unix** kernel as some other filename before installing the new kernel.

   To install the new kernel, type the commands

   ln [*final_name*] **/unix**
   ln [*final_name*] **/stand/unix**

7.10.2 Initiating Terminal Lines: Getty. The **/etc/inittab** file supplies the script for *init*(1M) to perform as a general process dispatcher. *Init* reads **/etc/inittab**, which lists each line by tty number and describes how *init* should treat the process specified for that line. For each line, init is directed either to start a *getty* process for the line, or to turn the process off. The line process, **/etc/getty**, initiates individual terminal lines.

If *init* invokes *getty* for a particular line, *getty* reads **/etc/gettydefs** to obtain the login prompt and, when a user logs in, attempts to read the user's name while adapting the system to the speed and terminal settings. The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line.

For uucp operation, the **L-devices** file lists the lines that are directly connected to other systems, or are available for calling systems. The **/etc/inittab** file must have corresponding entries for dial-out lines that turn off **/etc/getty**. For example, if **L-devices** lists tty401 as a uucp line, then the **/etc/inittab** file must have a corresponding entry:

nn:2:off:/etc/getty tty401 1200UUCP

where *init* turns off the *getty* for the line. If *getty* and *uucp* are running together, both processes try to read from and write to the same file, **tty***xxx*. The two processes compete for input, and the result is havoc. Another reason for turning off the *getty* is permissions. *Getty* sets the permissions for a line to be **crw - r - - r - -** , for system security. However, *uucico* requires the line permissions on a dial-out line to be **crw - rw - rw -** or uucp will not function. The field **1200UUCP** is read only when the **/etc/inittab** action field indicates the *getty* process should be started.

To receive a call from a remote system, one tty line in the **/etc/inittab** file must be labelled for a uucp line. A sample **/etc/inittab** line is:

nn:2:respawn:/etc/getty tty402 1200UUCP

The label **1200UUCP** would be defined in the file **/etc/gettydefs**. A sample definition found in **/etc/gettydefs** might be:

1200UUCP # B1200 UPCL -PARENB -INPCK CS8 # B1200
SCTOPB CREAD ISTRIP ICRNL IXON BRKINT OPOST ONLCR
ICANON ECHO ECHOK ISIG -PARENB -INPCK CS8 IXANY
TABS # login # 1200UUCP

For further information, refer to *init*(1M), *gettydefs*(4), *inittab*(4), and *getty*(1M).

7.10.3 Passwords. Remote systems that dial-in require a login-id and an entry in the local **/etc/passwd** file. Administrators may also add a second layer of protection and require specific dial-up passwords.

The remote system **rivwhy** login-id is **nuucp,** which commonly is shared by systems across the network. An example **nuucp** entry in the **/etc/passwd** file is:

nuucp:zaaAA:6:1:UUCP.Admin:/usr/spool/uucppublic:/usr/lib/uucp/uucico

After the **nuucp** login and password is verified, the remote system gains access to the *uucico* program, not the shell. The spool directory is used as the working directory.

For security, the login-id used by a remote system should not be used by any local user. Several remote systems may share the same login-id. (At the start of each conversation, the systems exchange unique system-names to identify themselves.)

(Because *uucico* runs setuid-to-uucp, it makes little difference from an operational perspective if remote systems share a login-id. However, one reason for giving different sites different login-ids and passwords is security. If any site's login-id/password is compromised, a new password can be assigned without affecting every other site.)

The **/etc/passwd** file requires an entry for the administrative uucp login. The login-id identifies the owner of all uucp object and spooled data files, and is commonly **uucp.** An example entry for the **uucp** login-id is:

uucp:zAvLCKp:5:1:UUCP.Admin:/usr/lib/uucp:

The uucp administrator gains access to the standard shell, not the *uucico* program. If a login-id other than *uucp* is chosen, the *make* file entry **OWNER=uucp** must be edited to reflect the new login-id.

7.10.4 L.sys. The **L.sys** file contains information needed by the calling system to contact another machine on the network. Incoming calls are not checked against this file. Each entry represents a system and contains six fields:

system-name time device speed/class phone-no login

(Multiple entries for one system may be present when the system can be reached by more than one communication path, e.g., by direct connection or an acu. Each entry defines a different path. The paths are tried in sequential order until the system is reached.)

The fields in **L.sys** are separated by one or more spaces, not tabs, and are defined as follows:

system-name

The name of the remote system.

time

A string indicating when the system may be called. The string gives the day-of-week, and the time-of-day expressed on a 24-hour clock, and contains an optional subfield to indicate a minimum time (in minutes) before a retry. The days are specified as: Su, Mo, Tu, We, Th, Fr, Sa, and Wk (any weekday). Uucp recognizes "Any" to mean a remote system can be called at any time. If the local system is a polled system and can never initiate calls, any non-recognizable sequence can be used; by convention, "Never" is common.

Time is specified as a range. If no range is given, calls are made at any time. Time ranges may cross 0000, spanning one day to the next, for example:

Wk0800-0600,45

describes a system that can be called on weekdays at any time from 8 a.m. to 6 a.m. the next morning. Calls cannot be made from 6 a.m. to 8 a.m. any day. The system must wait 45 minutes before retrying after a failed attempt to connect.

device

Either **ACU** for automatic calling unit or **tty*xx*** where the tty number is the special filename used for the hardwired device name, e.g., **tty0**.

speed/class

Typically, the line speed for the call (e.g., 300). When the "C" library routine "dialout" is available, the "dialout class" is indicated.

phone

An abbreviated version of the telephone number for the remote system. *Phone* is composed of an optional alphabetic abbreviation (which is defined in the **L-dialcodes** file), and a numeric part. An example is **pd3448**. For a hardwired-connected system, the field contains the same information as the *device* field.

login

Login information arranged in a series of fields and subfields, separated by spaces, that mirrors the expected conversation sequence between the calling system and the called system. For example, consider the login information fields:

ogin:--ogin:- EOT -ogin:- BREAK1 - ogin: nuucp sword: secret

The local system interprets this string as follows:

A. When the remote system answers, expect to receive a string ending in the letters "ogin:". If expected string is received, send the string contained in the next field, **nuucp**.

   1. If the expected string "ogin:" is not received, send the string in the subfield: - -. The hyphens are not read as characters themselves, but as delimiters for the subfield. Thus, the hyphens delimit a null field, i.e., a line feed. (A null field also can be specified with the string: "" .) After the line feed is sent, the local system should expect to receive a string ending in "ogin:". If the expected string is received, send the string contained in the next field: **nuucp**.

   2. If the expected string is not received, send the string in the subfield: "EOT" (which translates as a CNTRL-D). Expect to receive a string ending in "ogin:". If the expected string is received, send the string contained in the next field: **nuucp**.

   3. If the expected string is not received, send the string in the subfield: BREAK1. (BREAK1 is a break sequence created through a simulated BREAK character, explained later in this paragraph.) Expect to receive a string ending in "ogin:". If the expected string is received, send the string contained in the next field: **nuucp**.

   4. If the expected string is not received, an error message is reported to **LOGFILE**: FAILED (LOGIN). An **STST** file is created, and the job remains in the spool directory to be retried later.

B. After the **nuucp** login is sent, expect to receive a string ending in the letters "sword:" If the expected string is received, send the string contained in the next field: **secret**.

Thus, the generalized form of the login information follows the format:

expectA sendA expectB sendB expectC sendC

The expect fields may contain several subfields expressed as:

expectA[-send-expect-send-expect...] sendA expectB ...

where the *send* string is sent if the prior *expect* is not read successfully, and the *expect* following the *send* is the next expected string. Because the fields are sequential, the strings themselves need not be unique.

Special strings can be sent during the login sequence. The string EOT sends a CNTRL-D character that exits one login sequence and initiates another. The EOT string is used to clear a jam or noise on the line. The string BREAK sends a simulated BREAK character, using line speed changes and null characters. Because the BREAK is simulated, it may not work on all devices or systems. A number from 1 to 9 may follow the BREAK, indicating the number of null characters to be sent (default is 3). BREAK1 usually gives the best results for lines that are 300-1200 baud.

Use the null string in the first expect field if no characters are expected from the remote machine initially, or to immediately send a string (e.g., line feed) to the called machine.

Four character strings may be included in the *login-id* send string to force specific actions. For example, the login-id **nuucp\sTZ=MST7** contains a backslash-s that sends a space character, followed by a new assignment for the timezone variable. The four characters are:

backslash-s	Send a space character.
backslash-d	Delay one second before sending or reading more characters.
backslash-c	If placed at the end of a string, suppress the newline that is normally sent; else, ignore.
backslash-N	Send a null character.

Typical entries in the **L.sys** file are:

winkil Any,45 ACU 300 pd3448 ogin:--ogin:-EOT-ogin: nuucp ssword: cg>lee
kulrun Any tty12 9600 tty12 in--in nuucp ssword: sysV68

A typical entry for an active local system that is connected to a remote system through an auto-dial modem is:

rivwhy Any tty01 1200 tty01 "" "" "" AT OK- AT OK ATDT3332 NECT "" in--in nuucp ssword: sysV68 ssword: canduc

Notice in the auto-dial example the called system **rivwhy** expects the calling system to respond with two passwords.

When the local system is passive, or polled, the entry in the might be

spymys Never

This entry makes the system **spymys** known to the local system and allows jobs to **spymys** to be queued in the spool directory, but the local system must wait to be polled by **spymys.**

In general, when an entry is placed in the **L.sys** file, do not include any newline characters, or if they must be used, use a backslash to escape them.

7.10.5 L-devices. The **L-devices** file contains the list of lines that are directly connected to other systems, or are available for calling systems. There is one entry per line; each entry describes the line attributes and capabilities. Fields should be separated by one or more spaces, not tabs. The format of each entry is:

type line call-device speed protocol

where the fields are defined as follows:

type
: One of two keywords: **DIR** indicates the line is directly connected to another system; **ACU** indicates the line uses an automatic calling unit. An X.25 permanent, virtual circuit is classified as **DIR**.

line
: The device name for the line. For example, **tty***ab* names a direct line; **cu***10* names a line connected to an acu.

call-device
: For a line with keyword **ACU**, this field contains the acu device name. For a line with keyword **DIR**, the field is ignored. However, a placeholder must be used so that the *protocol* field is interpreted correctly.

speed
: The line speed at which the connection should run. If an X.25 link is used, this field is ignored.

protocol
: An optional field used only when a non-default terminal protocol is required. The X.25 protocol is the only non-default protocol supported, and is indicated with an **x**.

Typical entries in the **L-devices** file are:

```
ACU cul0 cua0 1200
DIR tty33 0 9600
DIR tty29 0 1200
DIR x25.s0 0 300 x
```

where the first entry describes a line for a 1200-baud acu, and the last entry is for an X.25 synchronous direct connection between two systems. In the last entry, the protocol field is filled in and the *call-device* and *speed* fields are meaningless.

7.10.6 L-dialcodes. The **L-dialcodes** entries define the abbreviations for telephone symbols used in the **L.sys** file. Fields should be separated by one or more spaces, not tabs. The entry format is:

abbreviation dialing-sequence

where the dialing sequence is prefixed to the phone number listed in the **L.sys** file. For example, an entry in **L-dialcodes**:

```
ore 503-241
```

would direct a system that read **ore-2463** in the **L.sys** file to send **503-241-2463** to the dialing unit.

7.10.7 L.cmds. The **L.cmds** file contains a list of commands that the local system is empowered to execute for a calling system. Entries in the file are entered one line per command. For example, an **L.cmds** file might contain the following entries:

```
rmail
rnews
```

These entries indicate the local system can execute **rmail** and **rnews** commands on behalf of all remote systems.

To provide particular machines with execute permission for a command, the entry in **L.cmds** takes the form:

cmd, machine1, machine2, ... machineX

where only the listed machines can execute the command *cmd*.

7.10.8 USERFILE. The **USERFILE** controls file accessibility by defining the pathways within the local system available to the local and remote uucp user.

USERFILE entries follow the format:

login,system **[c]** *pathname* [*pathname*]

where each field is separated by a single tab, and

login is the login name of the uucp user, remote computer, or a null field.

system is the remote computer's *system-name*. If null, the local system is assumed.

c is an optional "call-back required" flag.

pathname is the pathname prefix that defines the files to which *login,system* may have access. For example, if *pathname* was defined as **/usr/spool**, only files with pathnames that begin **/usr/spool** can be accessed by *login,system*.

The **USERFILE** is read and interpreted for each uucp command submitted to the system, but different entries in the file are read, depending on the uucp user and the type of command. In general, if a login-id or system-name identifies a user or system that does not have a specific entry, the local system grants access according to the first entry that contains a null login-id or null system-name. Refer to the specific explanations that follow.

The local system uses the **USERFILE** information to limit file access as follows:

If a remote system logs in:

> The system checks the **USERFILE** for an entry that begins with a *login,system* that matches the remote system. Many lines in the **USERFILE** may have the same *login* name. If the remote system's name is specifically included in an entry, the local system provides access through the pathname described in the entry. If the remote system-name is not found, the local system reads the first entry that contains a matching login with a null system-name.

If the line matched by a remote system logging in contains **c**:

> The **c** indicates a call-back by the local system before any transactions occur.

If a uucp command originates in the local system:

> The system reads the **USERFILE** for the first entry that matches the *login* of the requesting user. If no entry lists the user by login name, then the system accepts the first entry that contains a null login name with a null system-name. (Null system-name is read as "local system".) The pathnames listed in the entry line are the directories accessible to the user. If the file required for the uucp command is not located within the accessible pathway, the uucp command fails. A message appears in the **LOGFILE**: REQUEST DENIED.

If a uucp command originates in a remote system:

> The system reads the **USERFILE** for the first entry that matches the *login,system* of the remote system. If no entry specifies the remote system by name, the system reads the first entry that contains a matching login and a null system name. The

pathnames listed in the entry line are the directories accessible to the remote system.

For example, the **USERFILE** entry in the system **grtbks**

nuucp,rivwhy /usr/spool/uucppublic

limits **rivwhy** access to files contained in the public directory.

Consider the lines

nuucp,winkil /usr/bask /usr/spool/uucppublic
nuucp, /usr/spool/uucppublic

When **winkil** logs in as **nuucp**, it is given access to **/usr/bask** and the public directory, **/user/spool/uucppublic.** When any other system logs in as **nuucp**, it receives access only to the public directory.

USERFILE restrictions are in addition to the permission modes; they do not replace the normal uucp file permission requirements.

NOTE: USERFILE accepts up to 25 entry lines. If more than 25 entries are included in the file, uucp ceases to function.

7.10.9 FWDFILE, ORIGFILE. Two files allow an administrator to restrict the forwarding mechanism that operates for **mail** and **rmail. FWDFILE** and **ORIGFILE** can prevent remote systems from forwarding mail through a local link to sites that are expensive to reach (e.g., an overseas site).

The format of the files is the same:

system, [*user*] , [*user*]

ORIGFILE contains a list of systems (and users) for whom the local system forwards mail. The system-name refers to the system where the mail originated, not the last system that forwarded the mail to the local system. Using **ORIGFILE** to control the mail is advised only if the administrator is designing a highly restrictive system.

FWDFILE contains a list of systems (and users) to whom the local system will forward mail. The system-name refers only to the next system in the forwarding chain; it need not be the final destination. For example, the entry

burlne,jm,rch,

restricts mail to users **jm** and **rch** at an overseas site, **burlne.**

FWDFILE is a subset of the **L.sys** file of known systems.

7.10.10 Reconfiguring uucp. The information provided in this paragraph is useful only to administrators with source code.

Several source modifications may be required before the system programs are compiled. The changes are made to the local *system-name* and to directories used during compilation and execution. The directories affected are:

lib (Default location: **/usr/src/cmd/uucp**) This directory contains the source files for generating the uucp system.

program (Default location: **/usr/lib/uucp**) This is the directory used for the executable system programs and the system files.

spool (Default location: **/usr/spool/uucp**) This is the spool directory used during uucp execution.

xqtdir (Default location: **/usr/spool/uucp**) This directory is used during execution of execute files.

Two files may require changes: the *makefile* file, **uucp.mk**, and the **uucp.h** file. The directories *spool* and *xqtdir* should have 0777 for their permission modes.

7.10.10.1 uucp.mk. Several variable definitions may need changes:

INSDIR This is the *program* directory (e.g., INSDIR = **/usr/lib/uucp**). This parameter is used if "make cp" is used after the programs are compiled.

IOCTL This must be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library. The statement **IOCTL=ioctl.o** is required.

PKON The statement **PKON=pkon.o** is required if the packet driver is not in the kernel.

PUBDIR This is a public directory for remote access, and the login directory for remote uucp users. This should be the same as the public directory defined in **uucp.h**.

SPOOL This is the uucp spool directory. It should be the same directory as defined in **uucp.h**.

XQTDIR This is the directory that *uuxqt* uses during command execution. It should be the same directory as defined in **uucp.h**.

OWNER This is the administrative login for uucp.

7.10.10.2 uucp.h. The *program* and *spool* names should be changed if they are different from the default values. Other **uucp.h** variables that may be defined are:

UNAME Define this if the *uname* function is available.

MYNAME Change to the name of the local system if UNAME is not defined.

ACULAST This is the character required by the acu as the last character. For most systems, it is a — .

DATAKIT This should be defined if the system is on a datakit network.

DIALOUT This should be defined if the "C" library routine *dialout* is available.

7.10.10.3 System Compile. The command

make install

makes the directories, compiles the entire system, sets the permission modes, and copies the programs to the appropriate directories.

7.11 Debugging Uucp

Typically, problems with uucp develop because of restrictive permission modes, incorrectly defined variables, or bad hardware connections. This section provides a checklist for troubleshooting problems.

If a previously successful connection fails, look first for **LCK** and **STST** files that may have been left in the spool directory from a communication failure. **LCK** files should be removed from the directory during the system boot by the **/etc/rc** file. If a heavily used remote system is down for an extended period, the local sytem may advance the **STST** "recall" counter past the MAX RECALL number. Removing the **STST** file allows calls to be made. (Refer to paragraph 7.5 for information about reading the **LOGFILE**, locating **LCK** and **STST** files, and removing them.)

Communication problems may be traced to mis-matched baud rates, incorrect cabling or the lack of a null modem. If conversations are consistently failing, a lower baud rate for both systems may be necessary. A 9600-baud rate may not work for all systems. Noise on the line also may be responsible; a change to a limited distance modem may be required. Occasionally, if two systems are hardwire connected, the sheer volume of data transferred may overwhelm the receiving machine. Interactive terminal sessions are less demanding for the machine; it may be that a previously acceptable baud rate must be lowered for data transfer applications.

When you establish the line for a dial-out connection (through a modem or a direct connect), it is helpful to verify the physical connection with *cu*(1). For active connections that perform the dial out, *cu* can track down:

- A competing *getty*(1) on a dial-out line that should be "off" in **/etc/inittab.** (Refer to paragraph 7.10 for more information about line initialization and the required entries in **/etc/inittab** and **/etc/gettydefs.**)
- Incorrect permissions on the tty device in **/dev.** The permissions should be set to: **crw-rw-rw-.**
- Erroneous baud rate declarations in **L-devices** file.

To verify that a remote system can be contacted, invoke the *uucico* program directly from your terminal. For example, to verify that **winkil** can be contacted by **home**, a job would be queued with the command:

uucp —r checkfile grtbks!winkil!~/sam

The **—r** option forces the job to be queued, but does not call *uucico* to process the job. Instead, *uucico* is invoked directly with the command:

/usr/lib/uucp/uucico —r1 —x4 —swinkill

The **—r1** (r, numeral 1) option directs **home** to call *uucico* in master mode. The **—x4** option specifies the level of debugging that should be printed. Debugging levels of 1 through 9 (with 9 being the highest level) may be specified. The —s option identifies the system.

Another area that can be debugged readily with the output is the login sequence. Redirect the error output to a file if the login sequence you are trying to verify contains embedded control characters. (Embedded control characters are used when logging on through most "smart" modems and terminal concentrators, or switches.) You can watch the output and store it simultaneously by pipe fitting with *tee*(1).

For tricky connections through smart modems, look carefully for erroneous expect-send sequences (in the **L.sys** file) while initiating the connections through the modem. Verify the uucp password on the remote system by inspecting the debug output from *uucico* or by trying

to log in directly through *cu*.

When several jobs are queued for the remote system, it is impossible to assign any particular job a top priority as far as *uucico* is concerned. Monitor **LOGFILE** for error messages. The **ERRLOG** may also contain error messages that may isolate a particular area of difficulty.

Another source of problems could be the **USERFILE**. The file is limited to 25 lines. If more than 25 lines are entered, uucp ceases to function.

USER'S COMMENTS

SYSTEM V/68 USER'S GUIDE

Product Code 72903
Part Number 41966-00

Motorola welcomes your comments and suggestions. Please use this form.

- Does this manual provide the information you need? ☐ Yes ☐ No
  - What is missing?

- Is the manual accurate? ☐ Yes ☐ No
  - What is incorrect? (Be specific.)

- Is the manual written clearly? ☐ Yes ☐ No
  - What is unclear? (Be specific.)

- What other comments can you make about this manual?

- What do you like about this manual?

- Was this manual difficult to obtain? ☐ Yes ☐ No

Please include your name and address if you would like a reply.

Name_____
Company_____
Address_____
_____

• What is your occupation?

☐ Programmer	☐ Operator	☐ Manager
☐ Systems Analyst	☐ Instructor	☐ Customer Engineer
☐ Engineer	☐ Student	☐ Other_____

• How do you use this manual?

☐ Reference Manual	☐ Introduction to the Subject
☐ In a Class	☐ Introduction to the System
☐ Self Study	☐ Other_____

fold **fold**

MOTOROLA INC.
3013 S. 52nd Street
Tempe, AZ 85282

Attention: Software Publications, X4

---

fold **fold**

Staple Here

MOTOROLA
Computer Systems

3013 S. 52nd St.
Tempe, AZ 85282