ALLIANT COMPUTER B.V.
Weverstede 1
3431 JS NIEUWEGEIN
Tel. 03402 - 46624
Fax: 03402 - 31707

# CONCENTRIX

## System User's Guide

May 1987

This manual provides fundamental information for all users of the Concentrix operating system. It introduces the system, the documentation set, the user interface, and commonly used utility programs such as editors and electronic mail. This manual is revised to include an index.

Part Number: 301-00002-C

Concentrix Version: 1.0

ALLIANT COMPUTER SYSTEMS CORPORATION
One Monarch Drive ▪ Littleton ▪ Massachusetts 01460
617-486-4950

Alliant Computer Systems Corporation reserves the right to make changes in specifications and other information contained in this document without prior notice.

Although due care has been taken to present accurate information, ALLIANT DISCLAIMS ALL WARRANTIES WITH RESPECT TO THE CONTENTS OF THIS DOCUMENT (INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE), EITHER EXPRESSED OR IMPLIED. ALLIANT SHALL NOT BE LIABLE FOR DAMAGES RESULTING FROM ANY ERROR CONTAINED HEREIN, INCLUDING, BUT NOT LIMITED TO, FOR ANY SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF, OR IN CONNECTION WITH, THE USE OF THIS DOCUMENT.

The information contained in this document is summary in nature. More detailed information is available from Alliant.



Revision History:

A - May 1985
B - January 1986
C - May 1987



Printed in the United States of America

# TABLE OF CONTENTS

## CHAPTER 13 EDIT TUTORIAL

## CHAPTER 16 ADVANCED GUIDE TO ED

# PREFACE

This manual provides fundamental information for all users of the Concentrix operating system. It introduces the system, the documentation set, the user interface, and commonly used utility programs such as editors and electronic mail.

## Document Organization

This manual is organized into the following chapters and appendices:

1 INTRODUCTION TO CONCENTRIX

This chapter introduces the Concentrix operating system. It provides an overview of the system, the documentation set, the software development environment, the text editors, and the documentation preparation facilities.

2 GETTING STARTED

This chapter explains how to get started on the Concentrix operating system: getting an account, logging in, typing conventions, and so forth.

3 USING THE C SHELL

This chapter discusses the use of the C shell as an *interactive* command language interpreter: command syntax, job control, and the history mechanism.

4 THE FILE SYSTEM

This chapter discusses the organization of the Concentrix file system. It covers types of files, pathnames, directories, files, and filename substitution, file protection, and links.

5 PERIPHERAL DEVICES

This chapter provides fundamental information about the use of peripheral devices on the Concentrix operating system.

6 PROGRAMMING THE C SHELL

This chapter describes how to program the C shell by writing command files called *shell scripts*. Any command that you can use interactively can be executed in a shell script. In addition, the C shell provides several built-in commands that are used primarily for writing scripts.

7 C SHELL REFERENCE

This chapter provides detailed descriptions of the various aspects of the C shell.

8 USING THE BOURNE SHELL

This chapter describes, with examples, the Bourne shell. Like the C shell, the Bourne shell is both a command language and a programming language that provides an interface to the operating system.

9 BOURNE SHELL REFERENCE

This chapter provides detailed descriptions of the various aspects of the Bourne shell.

10 USING MAIL

*Mail* is a program for sending and receiving "electronic mail." It divides incoming mail into messages and allows you to deal with them in any order, provides a set of commands for manipulating messages and sending mail, offers simple editing capabilities, provides the ability to define convenient names for groups of users,

and sends and receives messages across networks such as the ARPANET, UUCP, and Berkeley network.

11 MAIL REFERENCE

This chapter provides detailed descriptions of the various aspects of *Mail.*

12 DISPLAY EDITING WITH VI

This chapter describes *vi,* a display-oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file that you are editing. Changes that you make to the file are reflected in what you see.

13 EDIT TUTORIAL

This chapter consists of a series of sessions that lead you through the fundamental steps of creating and revising text with *edit*, a simplified version of the line editor *ex*. It is recommended only for users who have no prior experience in text editing.

14 EX REFERENCE

This chapter describes the command-oriented part of the line editor *ex*; the display editing features of *ex* are described in Chapter 12, *Display Editing with Vi.*

15 INTRODUCTION TO ED

This chapter is a tutorial introduction to the line editor *ed.* It is recommended for users who have little or no prior experience in text editing.

16 ADVANCED GUIDE TO ED

This chapter provides detailed information about the line editor *ed,* including explanations and examples of special characters, line addressing and global commands, commands for "cut and paste" operations on files and parts of files, and the *r*, *w*, *m* and *t* commands of the editor.

17 ED REFERENCE

This chapter provides detailed descriptions of the various aspects of *ed.*

A GLOSSARY

This appendix explains various terms used in the Concentrix documentation set.

B THE ASCII CHARACTER SET

This appendix shows the contents of the file `/usr/pub/ascii`, which contains a map of the ASCII character set and can be printed as needed.

## Documentation Conventions

The UNIX documentation set was written by many people over many years. Although an effort has been made to use a consistent set of documentation conventions, you may notice variations.

| | |
|---|---|
| **boldface** | Boldface words are literals. Type them exactly as they appear. |
| [ ] | Square brackets indicate an optional item or set of items. |
| { } | Braces indicates a set of items from which you must select one. |
| ... | Ellipses indicate that the previous argument-prototype can be repeated. |
| - | Hyphen (-) most often indicates an option, even if it appears in a position where a file name could appear. In some cases, it is used to specify the standard input. Therefore, it is most unwise to have files whose names begin with a hyphen. |

| | |
|---|---|
| CTRL D | This notation means that you hold down the control key (usually marked CTRL) and press the specified alphanumeric key, in this case, D. Other notations used are ^D and control-d. |
| `this font` | Monospaced type indicates computer input or output: file names, example programs, and so forth. |

# CHAPTER 1
# INTRODUCTION TO CONCENTRIX

This chapter introduces the Concentrix operating system. It provides an overview of the system, the documentation set, the software development environment, the text editors, and the document preparation facilities.

## 1.1 THE CONCENTRIX OPERATING SYSTEM

The Concentrix operating system is Alliant's implementation of UNIX, a general-purpose, multi-user, interactive operating system. It is based specifically on "Berkeley 4.2BSD," the extended UNIX developed at the University of California, Berkeley.

There have been many versions of the UNIX time-sharing system. Developed at Bell Laboraties in 1969–70, UNIX quickly became popular for applications such as research in operating systems, languages, and computer networks, computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. UNIX became widely available in 1975. Since that time, it has become the "industry standard"system for scientific and engineering computing.

### 1.1.1 If You Are a Beginner

From a beginner's point of view, UNIX can be somewhat bewildering. It is oriented toward experienced programmers rather than casual users. For example, the user interface (called the "shell") is more like a programming language than a conventional "command language." The system includes hundreds of programs, most of which have a unique set of cryptic one- or two-character arguments.

If you are unfamiliar with UNIX, it is recommended that you set this manual aside for the moment and begin by reading *A Practical Guide to the UNIX System,* a softcover book that is part of the Concentrix documentation set. The time you take to read it will

be well spent. The *Practical Guide* is concise, well-written, and contains useful information about the C Shell. If you prefer a shorter introduction, the *FX/Fortran Programmer's Handbook* contains concise introductory material.

If you are familiar with UNIX but not the C shell, begin by reading the first six chapters of this manual. They cover the C shell, the file system, and other information necessary for daily use.

It is also recommended that you scan through the *Commands and Applications Manual.* The Concentrix system includes more than 250 commands. Although you may use only a small fraction of them in your day-to-day work, knowing what commands are available can help you solve future problems. As you work, keep a copy of the *Commands and Applications Manual* close by.

### 1.1.2 System Overview

The Concentrix operating system offers a number of features seldom found even in larger operating systems, including:

- a user interface (shell) that is actually a concise and powerful programming language based on the C Language.
- a hierarchical file system that allows each user to build a personal directory tree.
- the ability to use programs as *filters* to quickly solve data manipulation problems.
- the ability to run several processes at the same time.
- a high degree of device-independence and portability.

Besides the operating system proper, some major subsystems are:

- FX/Fortran provides access to the computation power of the Alliant hardware.
- C and Pascal languages.
- CCA EMACS and vi display editors.
- assembler, linking loader, debuggers.
- document preparation programs.
- a host of maintenance, utility, recreation and novelty programs.

## 1.2 THE CONCENTRIX DOCUMENTATION SET

The Concentrix documentation set consists of a total reorganization of the Berkeley 4.2 distribution. Some sections have been rewritten or replaced; others remain virtually intact. All of the manuals were typeset on an Interleaf Office Publishing System.

> *Note: You may find some things that are different on your system from what is described in the documentation set. When in doubt, consult with your system administrator.*

The Concentrix documentation set consists of 12 manuals. They are grouped by function:

### 1.2.1 Basic System Use

- **A Practical Guide to the UNIX System**

  A commercially-available primer on Unix, *A Practical Guide to the Unix System* is the place to start if you have no prior UNIX background. Author: Mark G. Sobell, The Benjamin/Cummings Publishing Co., Menlo Park, CA.

- **Commands and Applications Manual**

  This manual is an encyclopedia of the commands, application programs, and games available through user interaction with a shell.

- **System User's Guide**

  This manual provides fundamental information for all system users. It introduces the operating system and documentation set, the shells, and the display editor *vi*. It also covers line editing (*ed, edit,* and *ex*), and *mail*.

- **CCA EMACS Manual**

  This manual describes the use and simple customization of the display editor CCA EMACS. It also describes Elisp, the EMACS extension language, and provides an installation guide and release notes.

- **CCA EMACS Command Index**

  This manual is a concise guide to the use of the display editor CCA EMACS. It includes a comple list of EMACS commands organized by function.

- **CCA EMACS Command Chart**

  This pocket chart is a complete list of CCA EMACS commands, organized by prefix.

- **Document Preparation Guide**

  This manual describes the Concentrix document preparation facilities, including macro packages, text processors and preprocessors, text analysis programs, and fonts.

- **Release Notes**

  This manual provides information specific to the current release of the Concentrix operating system.

### 1.2.2 Software Development

- **Berkeley Pascal Manual**

  This manual describes the Pascal programming language as implemented in Berkeley 4.2BSD.

- **The C Programming Language**

  This is the definitive manual on the C language by its creators, Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall Software Series, Prentice-Hall Inc., Englewood Cliffs, NJ.

- **System Reference Manual**

  This manual is an encyclopedia of system calls, library subroutines, special files (devices), and file formats. It is intended for systems and applications programmers.

- **Systems Programming Guide**
  This manual describes how to interface with the Concentrix operating system and describes the implementation of various systems programs. It is intended for systems programmers.
- **Programming Tools Guide**
  This manual describes the following programming tools: *adb, awk, bc, dbx, dc, lex, lint, m4, make, ratfor, sdb, sed, sccs,* and *yacc.*

### 1.2.3 System Administration

- **Administration Commands**
  This manual is an encyclopedia of system administration commands.
- **Administrator's Guide**
  This manual describes the functions of a system administrator: installation, operation, configuration, accounting, file system checking, and so forth.

## 1.3 SOFTWARE DEVELOPMENT

The Concentrix system is a productive programming environment because there is a rich set of languages and tools available.

### 1.3.1 The Shell

A shell is an interactive command language interpreter that resembles a programming language with variables, control flow (if-else, while, for, case), subroutines, and interrupt handling. You can often get a job done merely by piecing together existing program with shell scripts. The pipe mechanism lets you fabricate quite complicated operations simply and elegantly.

The C shell is a powerful shell that was developed at the University of California, Berkeley. It provides a history mechanism, extended job control, and other features useful for software development. The Bourne shell is the original UNIX shell that was developed at Bell Laboratories.

### 1.3.2 FX/Fortran

FX/Fortran is an optimizing compiler and run-time system that makes use of the Alliant architecture: concurrent execution on multiple processors and the vector instruction set. FX/Fortran fully supports standard Fortran-77 and provides a large number of extensions. See the *FX/Fortran Programmer's Handbook* and *FX/Fortran Language Manual* for more information.

### 1.3.3 The C Language

C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978), which is part of the Concentrix documentation set. The Concentrix system itself is written in C, as are most of the programs that run on it.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it is wisest to confine the system interactions in a program to the facilities provided by the standard I/O library. Read the chapter on *UNIX Programming* in the *Systems Programming Guide* for more detail.

C programs that do not depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily.

There are a number of supporting programs that go with C. For example, *lint* (described in the *Programming Tools Guide*) checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

### 1.3.4 Pascal

The Concentrix system provides Berkeley Pascal (compiler only), which is an implementation of the Pascal language as described in the Jensen-Wirth *Pascal Report*. Extensions include a separate compilation facility and the ability to link to object modules produced from other source languages. In addition, Wirth's cross-reference program is provided.

### 1.3.5 Debuggers

*Adb* is a powerful assembly-level debugger that can be used for examining core dumps and patching files, but is rather hard to learn to use effectively. *Dbx* is a source-level debugger and is easier to use. Both are described in the *Programming Tools Guide*.

### 1.3.6 Make

*Make* is a program that builds other programs. It uses a special file (a *makefile*) that defines the commands used to create the program modules as well as the dependencies among the modules. Thus, *make* does only as much work as necessary to build a new version.

A *makefile* also can contain definitions of commands for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs. This is superior and preferable to using groups of shell procedures to maintain programs.

Similarly, a *makefile* can define how different versions of a document are to be created and which document preparation tools to use. See the *Programming Tools Guide* for details.

### 1.3.7 SCCS (Source Code Control System)

*SCCS* is a source code management system. It provides a rational way to have several people work on the same project at the same time. Concentrix provides two interfaces to SCCS: Berkeley and System V. Both are described in the *Programming Tools Guide*.

### 1.3.8 Yacc and Lex

If a software development project requires you to translate a language or a complex command interface into a set of actions or another language, you are in effect building a compiler. The program *yacc* (Yet Another Compiler-Compiler) is designed to help you develop a compiler quickly. The program *lex* (lexical analyzer generator) does the same job for simpler languages that can be expressed as regular expressions. *Lex* can be used by itself, or as a front end to recognize inputs for a *yacc*-based program. Both *yacc* and *lex* require some sophistication to use, but the effort required to learn them can be repaid many times over in programs that are easy to change and enhance. Both are described in the *Programming Tools Guide*.

## 1.4 TEXT EDITORS

The Concentrix system provides CCA EMACS, a powerful screen-oriented editor, as well as the standard UNIX editors.

### 1.4.1 CCA EMACS

CCA EMACS is an advanced, self-documenting, customizable, extensible, display-oriented text editor closely based on the EMACS that was written at the MIT Artificial Intelligence Laboratory. Although EMACS is an incredibly powerful editor, it is possible to use it quite successfully with the knowledge of only a few commands. If you have not used EMACS before, it is recommended that you begin by using the on-line tutorial; log in and type the command `teach-emacs`.

### 1.4.2 Vi

*Vi* (pronounced vee-eye) is a display-oriented, interactive text editor (based on the line editor *ex*) that also can be used on hardcopy terminals. Although much less powerful than EMACS, *vi* is quite popular among long-time UNIX users. *Vi* is described in this manual.

### 1.4.3 Line Editors

Concentrix provides the standard UNIX line-oriented text editors *ed, edit,* and *ex,* which are described in this manual.

### 1.4.4 SED

*Sed* provides many of the editing facilities of *ed*, but can apply them to arbitrarily long inputs. It is normally used as a "script editor" to apply source patches. *Sed* is described in the *Programming Tools Guide*.

## 1.5 DOCUMENT PREPARATION

The Concentrix system provides all of the standard UNIX document preparation tools. They are described in detail in the *Document Preparation Guide*.

### 1.5.1 Nroff and Troff

*Nroff* (pronounced "en-roff") and *troff* (pronounced "tee-roff") are text formatting programs. *Nroff* produces formatted output printable on terminals, line printers, and so forth. *Troff* is a superset of *nroff* that drives a phototypesetter.

Input files for *nroff* and *troff* contain text interspersed with "requests" that indicate in detail how the output is to look. For example, there might be requests that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

### 1.5.2 Macro Packages

By themselves, *nroff* and *troff* are difficult and tedious to use because the level of detail is very fine. In order to work at a higher level of detail, you can choose one of several packages of predefined formatting macros, such as -me and -ms. Macro packages reduce operations that would take several requests in *nroff/troff* to single macro calls. They do not, however, offer a complete set of functions. Most documents include some *nroff/troff* requests as well as macro calls.

For example, a source document containing -ms macro calls looks something like:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the -ms macro calls. .PP starts a new paragraph. Its precise meaning depends on the output device being used (typesetter or terminal, for instance), and the mode of operation (-ms has modes for articles, books, reports, and

so forth). -ms normally assumes that a paragraph is preceded by a space (one line in *nroff*, half a line in *troff*), and the first word is indented. Such rules can be changed, if you like, by changing the interpretation of .PP, not by retyping the document.

### 1.5.3 Other Documentation Tools

In addition to the text formatters, there is a host of supporting programs that help with document preparation. This section is far from complete, so browse through the *Commands and Applications Manual* for other possibilities.

- *Eqn* and *neqn* let you integrate mathematics into the text of a document, in a language that closely resembles the way you would speak it aloud. For example, the *eqn* input:

  ```
  sum from i=0 to n x sub i ~=~ pi over 2
  ```

  when used with *troff*, produces the output:

  $$\sum_{i=0}^{n} x_i = \frac{\pi}{2}$$

- *Tbl* provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.
- *Refer* prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.
- *Spell* detects possible spelling mistakes in a document. It works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job.
- *Grep* looks through a set of files for lines that contain a particular text pattern (rather like the *ex* editor's context search does, but on multiple of files). *Grep* is often useful for finding out in where misspelled words detected by *spell* are actually located.
- *Diff* prints a list of the differences between two files, so you can compare two versions of something automatically.
- *Wc* counts the words, lines and characters in a set of files.
- *Tr* translates characters into other characters.
- *Sort* sorts files in a variety of ways.
- *Cref* makes cross-references.
- *Awk* provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines.

Most of these programs are either independently documented (like *eqn* and *tbl*), or are sufficiently simple that the description in the *Commands and Applications Manual* is adequate explanation. The last few are for more advanced users, and are not limited to document preparation.

# CHAPTER 2
# GETTING STARTED

This chapter explains how to get started on the Concentrix operating system: getting an account, logging in, typing conventions, and so forth.

## 2.1 GETTING AN ACCOUNT

If you do not already have an account on your system, contact the system administrator. Most systems have one or more administrators who are responsible for adding new users, installing software, and other system maintenance functions. He will establish the following for you:

- A login name. Your login name is a word that identifies you to the system as an authorized user. The format of your login name is site-dependent. You may be assigned a number or permitted to use your own name.
- A temporary password. A password allows the system to verify that you are indeed the person entitled to use your login name. This step is not strictly necessary and is often omitted.
- A login directory (disk space in which to keep your files).
- A user environment (information specific to you that is known to the system and its programs).

## 2.2 LOGGING IN

1. Be sure to set the switches appropriately on your terminal: speed, upper/lower case mode, full duplex, no parity, and any others recommended by your system administrator.
2. Establish a connection using whatever procedure is needed for your terminal and site. If your terminal is wired directly to the system, simply turn on the power. If

you are using a modem, establish a telephone connection. When you hear a high-pitched "whistling" sound, engage the modem.

3. Press the RETURN key one or more times, slowly. You should see this prompt:

```
login:
```

4. When you get a `login:` message, type your login name, *using the exact upper- and lower-case letters*, and press RETURN; the system does nothing until you press RETURN.
5. If a password has been set for you, this prompt appears:

```
Password:
```

Echo is turned off (if possible) while you type your password. Do not forget to press RETURN.

6. When you have successfully logged in, a prompt, typically a single per cent sign (C shell) or dollar sign (Bourne shell), indicates that the system is ready to accept commands.

### 2.2.1 Possible Problems and Solutions

- Random "garbage" characters appear. Make sure that your terminal (and modem) is set to the right speed. Some terminals have independent transmit and receive settings. Press RETURN or BREAK several times, slowly. It can take a few seconds for the system to recognize your speed.
- Only upper-case characters appear. Check your terminal settings. Some terminals have an upper-case-only mode.
- Everything appears twice. Change your terminal setting from half-duplex to full-duplex.

If you have checked everything and still cannot get a `login:` message, ask your system administrator for help.

## 2.3 YOUR FIRST SESSION

The first time you log in, set your password: use the *passwd* command as shown below. The passwords you type in response to the prompts do not echo.

```
% passwd
Changing password for ...
Old password:
New password:
Retype new password:
%
```

It is recommended that you select a password that would be difficult for someone to guess, particularly if you will be working with confidential data. Use at least six characters, preferably mixed upper-case, lower-case, and numeric. Avoid names, initials, and correctly-spelled English words.

Each time you log in, you may see a message of the day and/or a notification that you have mail. The provides an "electronic mail" program that allows you to communicate with other users. To read your incoming mail, type the command:

```
% mail
```

The *mail* program prints a list of your messages and waits for a command. Some basic *mail* commands are:

| | |
|---|---|
| RETURN | prints the next message. |
| **dt** | deletes the current message and prints the next message. |
| **q** | exits from mail. |

For a complete description, refer to the chapter on *Mail* in this manual.

When your system administrator created your account, he set up your user environment (terminal type, login directory name, and so forth) based on the information he had at the time. As you work, you may find that adjustments are needed to meet your requirements.

You can make some adjustments yourself. Your login directory contains special command files (*shell scripts*) that execute automatically at specific times (see Chapter 5). You can edit these files and add, delete, or change commands.

You may also require changes to system files. For example, if your system has specific terminal types bound to its communication lines, the wrong settings can prevent programs such as video editors from working properly. You cannot correct system files yourself; it must be done by your system administrator.

## 2.4 LOGGING OUT

To log out, type `logout`. If you are working over a phone line, hanging up the phone logs you out.

It is insufficient just to turn off your terminal. The system does not supply a default time-out mechanism to log unused terminals out.

## 2.5 YOUR TERMINAL

The Concentrix operating system requires terminals that can send and receive lower-case letters. If your terminal does not support lower-case letters, get another terminal.

The system has full read-ahead (also known as typeahead), which means that you can type as fast as you want, whenever you want, even when the system is printing. If you type during output, your input characters appear intermixed with the output characters, but they are stored away and interpreted in the correct order. Thus, you can type several commands, one after another, without waiting for the first to finish or even begin.

### 2.5.1 Recovering From Problems

Sometimes your terminal can get into a state in which it behaves strangely. For example, each letter may appear twice, or the RETURN key may not work as expected. When that happens, try each of the following if necessary:

- If your terminal has a "reset" function, use it.
- Type the following sequence of characters, even if you can't see them:

  LINE FEED `reset` LINE FEED

- If you can, log out and log back in. If not, log in on another terminal and kill your process (see Chapter 3 for instructions).

If tab characters seem to be working incorrectly, type the command:

```
stty -tabs
```

If your terminal has programmable tabs, the command sets the stops correctly for you. Otherwise, it instructs the system to convert each tab into the right number of space characters.

## 2.6 KEYSTROKE CONVENTIONS

The Concentrix operating system provides some simple keystroke conventions to help you work. Some of these conventions are unique to the C shell. Ask your system manager which ones are in common use at your site.

If you prefer to use different keys, you can insert an *stty* command in your startup command file that redefines the functions.

### 2.6.1 Typing Commands

| | |
|---|---|
| DELETE | erases the last character (RUBOUT on some terminals). |
| CTRL U | erases the entire line. |
| CTRL R | retypes the entire line. |
| RETURN | executes a command line. |
| \ | continues a command on the next line. |

### 2.6.2 Replacement Characters

These keystrokes are for terminals that do not provide the corresponding keys.

| | |
|---|---|
| CTRL H | issues a backspace character. |
| CTRL I | issues a horizontal tab. |

### 2.6.3 Program Control

CTRL C aborts a program.

CTRL Z stops (suspends execution of) a program.

### 2.6.4 Input/Output

CTRL S stops terminal output.

CTRL Q resumes terminal output.

CTRL O flushes terminal output (allows a command to complete execution without having to view its output). Type CTRL O again to resume output.

CTRL D signals end-of-file.

If you want to examine the output of a command without having it scroll off the screen, send the output to the program *more*. *More* displays the output, pauses after each complete screenful, and accepts commands that let you control the display. For example, type the following command and use CTRL Q and CTRL S to control the output:

```
% cat /etc/termcap
```

When you have seen enough, type CTRL O to flush the rest away. Now type:

```
% more /etc/termcap
```

Press the question mark key to get a set of instructions. When you have seen enough, type q or CTRL C.

# CHAPTER 3
# USING THE C SHELL

This chapter discusses the use of the C shell as an *interactive* command language interpreter: command syntax, job control, and the history mechanism. The C shell is also a programming language. C shell programs, called *shell scripts*, are described in Chapter 6. A detailed reference document that describes all aspects of the C shell can be found in Chapter 7.

## 3.1 WHAT IS A SHELL?

Most users communicate with the Concentrix system through a program called a *shell*. The name "shell" comes from the conceptual schema shown in Figure 3–1.

A shell is a user program just like any that you might write. It functions primarily as a medium through which other programs are invoked. While it has a set of *built-in* functions that it performs directly, most user commands cause execution of programs that are external to the shell.

The C shell is a powerful shell that was developed at the University of California, Berkeley. It provides a history mechanism, extended job control, and other features useful for software development. Alliant recommends that you use the C shell.

The Bourne shell is the original UNIX shell that was developed at Bell Laboratories. It is described in Chapters 8 and 9. If you prefer the Bourne shell, you can make it your default login shell; use the *chsh* command described in the *Commands and Applications Manual*.

Any interactive program can function as a shell. For some applications, a dedicated interface to the system might be desirable and can be easily arranged for. Each user's entry in the system password file contains the name of a program to be invoked at login. Normally, the program invoked is the C shell or the Bourne shell.

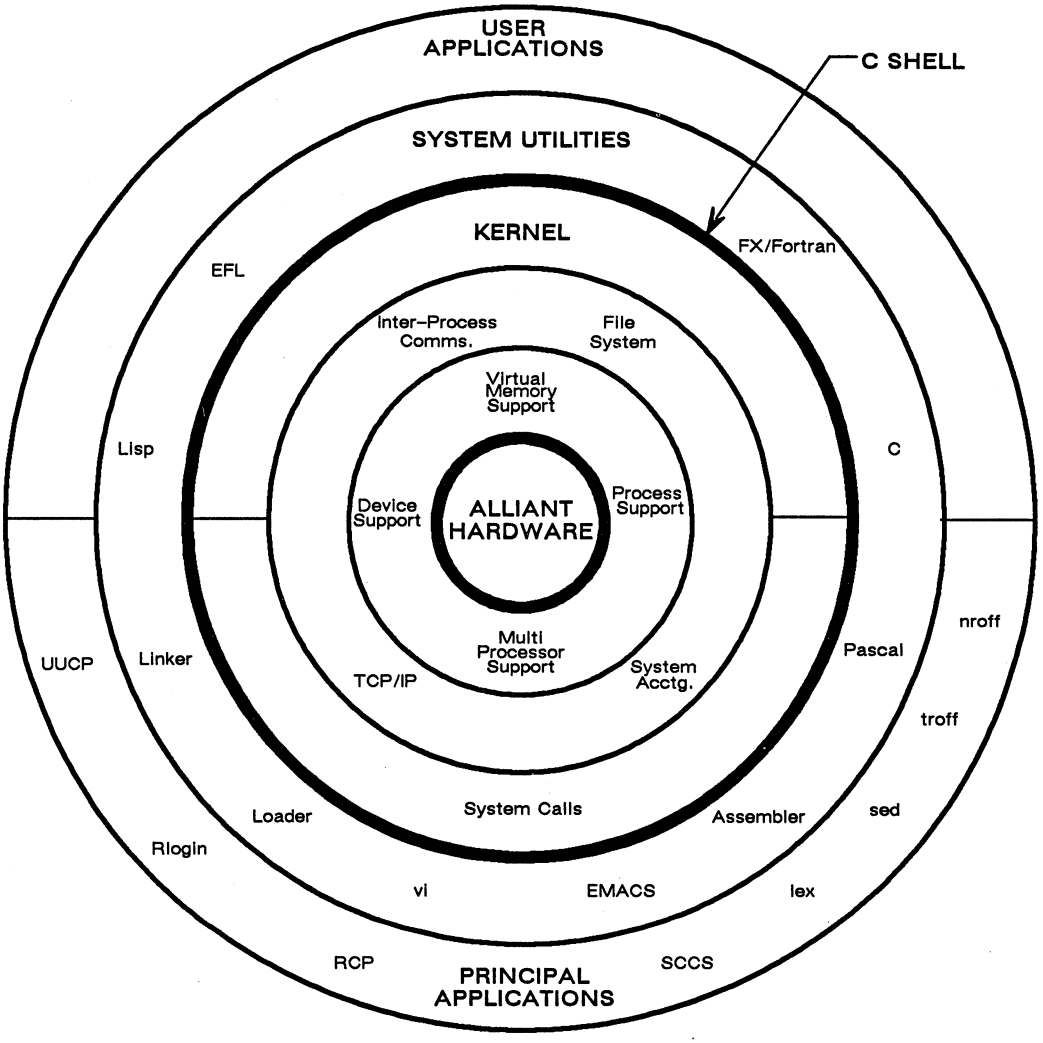



**Figure 3–1: The Concentrix Operating System**

A non-technical user, however, might prefer *Mail* as his default user interface. When that user logs in, he is inside *Mail* and can begin work immediately. If necessary, he can use a temporary escape to gain access to other utilities.

Another example is a guest account; by replacing the login shell with a different user interface, transient system users can be prevented from invoking programs not intended for their use.

## 3.2 COMMANDS

In its simplest form, a C shell command line consists of a command name followed by arguments, all separated by spaces:

*command arg1 arg2 ... argn*

The arguments are accessible to the program invoked by the command. This example (with the C shell percent prompt shown) consists of a command and two arguments:

```
% ls -l myfile
```

The first word, `ls`, names the command to be executed, in this case a program that lists files.

The second word, `-l`, is an argument that specifies an optional capability of the command that you wish to invoke. By convention, most option arguments (options for short) begin with a hyphen (-) to distinguish them from other types of arguments. Some commands also accept options that begin with a plus sign (+). See the preface to the *Commands and Applications Manual* for more information about options.

The third word, `myfile`, is a file name argument. By convention, most arguments that do not begin with a hyphen are interpreted as user-supplied information such as file names, user names, and so forth.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal; the shell executes the command; when execution completes, the shell prompts for a new command.

### 3.2.1 Types of Commands

Some commands are "built-in" or specific to a particular shell. For example, the C shell and the Bourne shell both have commands named *kill, nice, nohup,* and *set*. The exact syntax for each command, however, is different for each shell. All of the built-in commands are listed in Chapter 7. The most commonly-used ones are also in the *Commands and Applications Manual.*

When the shell recognizes a built-in command, it simply executes it and prompts for the next command. Most commands, however, are the names of programs external to the shell. If the shell does not recognize a command as built-in, it searches specific parts of the file system (explained in detail below) for a program that has the same name as the command. If the program is found, it is brought into memory and executed. A utility program can be a binary file or a shell script.

### 3.2.2 Command Execution

When you type a command, the shell:

- splits up the command name and the arguments into separate words.
- performs various substitutions (explained in detail later) on the words.
- executes the command.

A command name can be a full pathname that specifies any program in the file system. For example, this is a valid command:

```
% /usr/jones/project/bin/cleanup
```

If you do not supply a pathname, the shell examines the variable `path`, which contains a sequence of directory names where the shell searches for the command. Using the *set* command (which also displays values), the value of path might appear as:

```
% set path
path    (. /usr/ucb /bin /usr/bin)
```

The information within the parentheses indicates that the variable path contains:

| | |
|---|---|
| . | your current working directory. |
| /usr/ucb | commands developed at Berkeley. |
| /bin | commands developed at Bell Laboratories. |
| /usr/bin | other commands developed at Bell Laboratories. |

You can add any directory to your search path. For example, programs developed locally can be found in the directory `/usr/local`. To add this directory to your path, edit your `.cshrc` or `.login` file and change the `set path` command to:

```
set path=(. /usr/ucb /bin /usr/bin /usr/local/bin)
```

#### 3.2.2.1 The Hash Table

When it starts up, the shell examines each directory in your search path and creates a *hash table* of where commands are found. Thus, if you create a command or change the location of a command, the shell might not be able to find it. To reset the hash table, type:

```
rehash
```

The *rehash* command is not necessary if the new command is in your current directory. The shell always looks in your current directory (placing it in your path variable reduces overhead).

### 3.2.3 Multiple Command Lines

An input line can contain more than one command if you separate the commands with a semicolon. For example, this command runs the program `date`, followed by the program `ls`, followed by the program who:

```
% date; ls -l; who
```

To cause one or more commands to execute in a separate subshell, enclose them in parentheses. For example:

```
% pwd; (cd /bin; pwd); pwd
/usr/jones
/bin
/usr/jones
```

Because the cd (change directory) command executed in a separate subshell, it had no effect on the current shell's working directiory.

### 3.2.4 Command Input/Output

All commands executed by the shell begin with three open files: the *standard input* file, the *standard output* file, and the *diagnostic output* file. By default, all three files are connected to your terminal.

Several symbols allow you to *redirect* input/output. Redirection has no effect on the operation of a command. In fact, the command is "unaware" that any redirection is going on.

< causes the standard input to come from a file. For example:

```
% mail adam eve mary joe <letter
```

You can prepare the file letter, then send it to several people with a single command.

> redirects the standard output to a file. If the file does not exist, the shell creates a new one. If the file exists, it is overwritten, unless the shell variable noclobber is set. For example:

```
% ls >filelist
```

A list of the files in your current working directory is placed in the file filelist.

>! is the same except that the shell variable noclobber is ignored.

>> redirects the standard output to the end of a file. If noclobber is set, the file must already exist.

```
% cat f1 f2 f3 >>filelist
```

The files f1, f2 and f3 are appended to (added to the end of) filelist. The existing contents are unchanged.

>>! is the same except that the shell variable noclobber is ignored.

>& redirects both the standard output and the diagnostic output to a file. If the file does not exist, the shell creates a new one. If the file exists, it is overwritten, unless noclobber is set. For example:

```
% nroff chap1 >& chap1.out
```

Any diagnostic messages that occur in processing chap1 are written to chap1.out.

>&! is the same except that the shell variable noclobber is ignored.

The shell variable noclobber prevents existing files from being destroyed. It also causes an error if you try to append to a nonexisting file. Each output redirection symbol has an alternate version with an exclamation point that overrides noclobber.

> *Note: There must be a space between the exclamation point used to override* noclobber *and the next argument. Otherwise, the exclamation point would invoke the history mechanism (see Section 3.4) with a totally different effect.*

Redirecting input can make an editor like *ed* do things that would normally require special programs on other systems. For example, suppose that you want to see the first and last lines of each of a set of files, `chap1.1` to `chap1.25`. You could laboriously type:

```
% ed
e chap1.1
1p
$p
e chap1.2
1p
$p
e chap1.3
1p
$p
```

and so forth. You can do the job much more easily. Create a file `script` that contains

```
1p
$p
```

Now, program the C shell to perform a loop. This simple program sets the shell variable `i` to each file name in turn, then invokes *ed*:

```
foreach i (chap*)
        ed $i <script
end
```

You can type the program as a command, or store it in a file for later execution.

### 3.2.5 Pipes

A *pipe* is a way to direct the output of one program to the input of another program, so that the two run as a sequence of processes: a *pipeline*.

`|` redirects the standard output of a program to the standard input of the next program. For example:

```
% cat f g h | pr
```

The output from *cat*, which would normally go to the terminal, instead becomes the input to *pr* for formatting. Another example:

```
% who | wc
```

This pipeline tells how many people are logged on (in the first column).

`|&` redirects both the standard and diagnostic output of one program to the input of another.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

The pipe mechanism lets you fabricate quite complicated operations out of programs that already exist. A program that copies its standard input to its standard output (with processing) is called a *filter*. Some commonly-used filters perform character transliteration, selection of lines according to a pattern, sorting, encryption, and decryption.

For example, suppose that you want to know the five largest files in your directory. The *-s* option of *ls* produces an alphabetic list of files with sizes in blocks of 512 characters. The *-n* option of *sort* specifies a numeric sort, rather than an alphabetic sort and the *-r* option specifies reverse (descending) order. The *head* command displays a specified number of lines from the beginning of a file. Thus:

```
% ls -s | sort -n -r | head -5
```

In this command, *sort* and *head* are filters. Another example is the first draft of the *spell* program (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 it goes a long way for such a small effort.

## 3.2.6 Command Substitution

You can cause a command to be executed by the C shell, then replaced in a command line by its own output. Enclosing a command in open single quotes (`) causes this *command substitution* to happen. For example:

```
% set pwd=`pwd`
```

This command line executes the command *pwd* and stores its output (your current working directory) in the variable pwd. Another example:

```
% ex `grep -l TRACE *.c`
```

This command line runs the editor *ex* on output from a *grep* command that searches for files that contain the string TRACE and whose names end in .c.

To suppress command substitution, enclose the string in (close) single quotes or precede each open single quote with a backslash. Enclosing the string in double quotes does not work. For example:

```
% echo ``ls``
ls
% echo "`ls`"
bin desktop mbox
```

Command substitution also occurs when a shell script uses in-line data. See Chapter 6 for details.

## 3.2.7 Defining Your Own Commands

The shell provides an *alias* mechanism that can be used to simplify the commands you type, to provide short names for commands, to supply commonly-used arguments, and

to define new commands in terms of other commands. For example, to create a command `foo` that invokes `date` and who, type:

```
% alias foo 'date;who'
% foo
```

As long as you remain logged in, `foo` will do the same thing. If you create an *alias* that is useful enough to make permanent, add the *alias* command to your `.login` or `.cshrc` file. For example,suppose you would like the command `ls` to always list in long format, that is to always use the `-l` option. Edit your `.cshrc` file and insert:

```
alias ls ls -l
```

or even:

```
alias dir ls -l
```

It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism (see Section 3.4). For example, the following definition executes an *ls* command after each *cd* (change directory) command:

```
alias cd 'cd \!* ; ls '
```

The entire alias definition is enclosed in single quotes to prevent filename substitutions from occurring as it is typed in and to prevent the semicolon from being recognized as a shell metacharacter. The exclamation point is escaped with a backslash to prevent it from being interpreted as a history substitution as you type it in.

The construction `\!*` expands to the entire argument list to the *cd* command (as explained in Section 3.4.3), and does not give an error if there are no arguments. The semicolon separates the commands.

Similarly the following alias defines a command that looks up its first argument in the password file:

```
alias whois 'grep \!^ /etc/passwd'
```

The construction `\!^` expands to the first argument to the *whois* command (as explained in Section 3.4.3).

## 3.3 JOB CONTROL

Job control is perhaps the most powerful feature of the C shell. The ability to stop a task, do something else, and resume the orginal task is fundamental to the way most people work.

### 3.3.1 Jobs

A *job* consists of one or more commands typed together as a pipeline or as a sequence of commands separated by semicolons. For example, some command lines that create jobs (one per line) are:

```
% sort < data
% ls -s | sort -n | head -5
% mail harold
```

As each job is started, it is assigned an identifying number called the *job number* that you can use to refer to that job. The shell assigns job numbers consecutively, beginning with 1. For example, if there are no current jobs, the next number assigned is 1; if there is a job number 2, the next number assigned is 3.

The C shell maintains a job table that contains the command names, arguments and the process numbers of all commands in each job as well as the working directory where the job was started.

Jobs terminate when a program gets an end-of-file from its standard input. For example, the *mail* program terminates when you type CTRL D.

By default, the shell also terminates (logs you out) when it gets an end-of-file. To prevent this from happening, set the `ignoreeof` variable (described in Chapter 7).

## 3.3.2 Foreground and Background Jobs

Each job in the job table has one of the following states:

- running in the foreground
- running in the background
- stopped

The *foreground job* is a job that executes while the shell waits for it to finish. No other commands can be entered in the mean time. Thus, only one job can run in the foreground.

A *background job* is a job that runs at the same time that the foreground job continues to be read and executed by the shell. Several jobs can be *stopped* (active but not running) or running in the background.

This feature allows you to run non-interactive programs such compilers and text processors in the background at the same time that you are running an interactive program in the foreground. You can even log out and leave background jobs running.

To run a job in the background, type an ampersand (&) at the end of a command or pipeline. The C shell starts the job runnng in the background, types its number, as well as the process numbers of all its (top level) commands, and immediately accepts another command from the terminal. For example:

```
% ls -s | sort -n > usage &
[2] 2034 2035
```

The two programs, *ls* and *sort*, start together as a background job. The shell prints the job number in brackets, in this case `[2]`, followed by the unique *process number* of each program in the job. Some of the job control commands accept process numbers as well as job numbers.

The ampersand can be used several times in a line. For example:

```
% cc prog1 & cc prog2 &
```

Both jobs run simultaneously in the background (in this case, competing with each other for resources). The outputs of the commands appear intermingled.

To run several consecutive jobs in the background, enclose them in parentheses. For example:

```
% (cc prog1; cc prog2)&
```

When a background job terminates, the shell reports the job number, the command, and a done message. If the job terminates abnormally the message might say something like `Killed`.

By default, shell waits for the foreground job to terminate before printing the report. If you prefer the shell to report immediately when a background job terminates, (possibly interrupting the output of other jobs), set the `notify` variable (see Chapter 7).

Normally, background jobs can write output to the terminal. You can, however, cause background jobs to stop when they are about to write:

```
% stty tostop
```

This command prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing its output. You can think of it as a "do not disturb" sign. For example:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
```

Some time later, when you have finished editing:

```
q
[1] Stopped (tty output)      wc hugefile
% fg wc
wc hugefile
  13371   30123     302577
% stty -tostop
```

The wc command, which counts the lines, words, and characters in a file, produced one line of output. When it tried to write to the terminal it stopped. Only when you restarted it in the foreground, could it to write to the terminal.

Some interactive programs have long periods in which they require no interaction. If you run such a program in the background, it stops each time it requires input. You can restart it in the foreground, supply the input, stop the program, and restart it in the background.

### 3.3.3 Aborting the Foreground Job

To abort the foreground job, type CTRL C, which sends an INTERRUPT signal to the job. For example:

```
% cat /etc/termcap
```

This command displays a large file. To abort the job, type CTRL C. The *cat* command terminates and the shell prompts for the next command.

Some programs, such as the shell, handle CTRL C. If you press CTRL C while the shell is waiting for input, it just reprompts. Text editors generally abort whatever they are doing but leave you in the editor.

To abort a program that handles CTRL C (other than the shell), you may have to use a command specific to the program. Alternatively, you can type CTRL Z to stop the job, then use the *kill* command to abort it (see below).

If you write or run programs that are not fully debugged, it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing CTRL \. That usually provokes the shell to produce a message like:

```
Quit (Core dumped)
```

The message indicates that a file `core` has been created containing information about the program's state when it terminated. You can examine `core` yourself, or forward information to the maintainer of the program telling him where the file is located.

You can set the abort command to whatever keystroke you like. For example some terminals have an INTERRUPT key that might be more convenient than CTRL C. Long-time UNIX users might prefer DELETE.

## 3.3.4 Aborting a Background Job

Background jobs ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. For example:

```
% kill %2
[2]    Terminated              cc myprog.c
```

If you cannot remember the job number, use the *jobs* command to print the numbers of all your jobs.

## 3.3.5 Stopping Jobs

A feature unique to the C shell is the ability to stop and restart a job. Stopping jobs can be very useful for day-to-day work when you need to temporarily change what you are doing (execute other commands) and then return to the stopped job.

For example, if you are editing a file, you can stop the editor, start a compile running in the background, read your mail, and resume editing. Stopping and restarting an editor is *much faster* than exiting and reentering.

Also, foreground jobs can be stopped and then restarted as background jobs, allowing you to continue other work rather than wait for a job to finish.

When jobs are stopped they merely cease any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports.

#### 3.3.5.1 Stopping the Foreground Job

CTRL Z stops your foreground job and returns control to the C shell. The job remains stopped until you restart it but is otherwise unaffected. You can have several jobs stopped at a time. For example:

```
% cc myprog.c
^Z
Stopped
```

You can execute other commands while the compile job remains stopped.

#### 3.3.5.2 Stopping a Background Job

Use the *stop* command as shown below to stop a background job. The stop command accepts a job number argument:

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)        sort usage
%
```

### 3.3.6 Restarting Jobs

You can restart a stopped job in the foreground or in the background. The *fg* command restarts a stopped job in the foreground.The *bg* command restarts a stopped job in the background.

The shell retypes the command line to remind you which command is being continued, and causes the job to resume execution. Unless any input files in use by the stopped job have been changed in the mean time, the suspension has no effect whatsoever on the execution of the job.

You can change your current working directory between stopping and starting a job without any adverse effects. When you start a job, the shell stores your current working directory as part of the job's environment. In order to prevent you from becoming confused, however, the shell informs you whenever you terminate, stop, or restart a job in the foreground, and your current working directory is not the same as the directory in which the job was started. For example:

```
% ed prog.c
1143
^Z
Stopped
% cd myproject
% fg
ed prog.c (wd: ~)
```

When you restart *ed*, the shell informs you that *ed* was started in your login directory. The purpose is to remind you that *ed* has no knowledge of your new working directory.

```
q
(wd now: ~/myproject)
%
```

When you exit from *ed*, the shell informs you that your current working directory is now `~/myproject`. The purpose is to remind you that your working directory is not the same as that of the job that just terminated.

These messages can be confusing if you use programs that change their own working directories. The shell remembers only the directory in which a job started and assumes that it does not change. If you are in doubt as to a job's working directory, the –l option of the *jobs* command displays the working directory of suspended or background jobs when it is different from the current working directory of the shell.

## 3.3.7 Summary of Job Control Commands

The *current job* is the default argument for the job control commands. When only one job is stopped or running in the background (the usual case) it is the current job. If you stop a foreground job, it becomes the current job. The existing current job (if any) becomes the *previous* job. When the current job terminates, the previous job becomes the current job.

*jobs* types the table of jobs, giving the job number, commands and status (`Stopped` or `Running`) of each backgound or suspended job. The `-l` option types the process numbers as well.

+ indicates the current job.

– indicates the previous job.

The *jobs* command only prints jobs started in the currently executing shell; it knows nothing about background jobs started in other login sessions or within shell files. The *ps* command can be used in this case to find out about background jobs not started in the current shell.

*fg* runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal).

*bg* runs a suspended job in the background. It is usually used after stopping the currently running foreground job with CTRL Z (the STOP signal). The combination of CTRL Z and the *bg* command changes a foreground job into a background job.

*stop* suspends a background job.

*kill* terminates a background or suspended job immediately. In addition to jobs, it can be given process numbers as arguments, as printed by *ps*.

*notify* (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

All job control commands accept an optional argument that identifies a particular job:

*%+* specifies the current job (the default).

*%–* specifies the previous job.

*%n* specifies job number *n*.

*%pref* specifies a job where *pref* is some unique prefix of the command name and arguments

*%?string* specifies a job that contains a unique *string*.

Some of the job control commands also accept process numbers (printed by the *ps* command.)

## 3.4 THE HISTORY LIST

As you work, the C shell saves the commands that you type. The contents of this *history list* are available to you through a concise notation that allows you to:

- correct minor typing errors without tedious retyping.
- reuse any previous command.
- use parts of previous commands to form new commands.
- save and restore a terminal session.

The *history* command displays the contents of the history list. For example:

```
% history
     1   cd misc/letters
     2   mail
     3   write michael
     4   ex write.c
     5   cat oldwrite.c
     6   history
```

The numbers are called *event* numbers because an event can contain several commands. To make the C shell display event numbers as you work, type:

```
% set prompt='\! % '
```

*Note: The examples in this section are shown with event numbers.*

The history list is only as big as you want it to be. The `history` variable controls its size. For example, if you want to save to the 30 most recent events, put this command in your `.login` file:

```
% set history=30
```

### 3.4.1 Correcting Typing Errors

Perhaps the most common use of the history mechanism is fixing typos. Suppose that you type:

```
8 % cp /smith/projects/documents/commands/fobar .
cp: /smith/projects/documents/commands/fobar: No such file or directory
```

Rather than retype the whole line, type:

```
9 % ^fo^foo
cp /smith/projects/documents/commands/foobar .
```

This notation repeats the previous event, replacing the first appearance of one substring with another. It is an abbreviated form of a more general feature that is described later.

### 3.4.2 Reexecuting Events

To repeat a previous event, type an exclamation point followed by the number of the desired event. The exclamation point invokes a *history substitution*. The C shell displays the resulting command. For example:

```
10 % !1
cd misc/letters
```

You can use an absolute event number, as shown above, or a relative event number, counting backward from the current event. For example:

```
11 % !-2
cp /smith/projects/documents/commands/foobar .
```

If the current event number is 11, this substitution gets you event number 9. There is an abbreviation for the previous event: a double exclamation point.

```
% !!
```

is exactly the same as:

```
% !-1
```

You also can use the name (or partial name) of a command. For example:

```
12 % !c
cp /smith/projects/documents/commands/foobar .
```

The shell searched for the most recent event that begin with the specified string, in this case, `c`. Similarly, you can use an arbitrary substring enclosed in question marks. For example:

```
13 % !?michael?
write michael
```

The shell searched for the most recent event that contained the specified string, in this case, `michael`. You can omit the second question mark if it would be the end of the input line.

### 3.4.3 Reusing Parts of Events

You can refer to any word or set of words in an event. The words are numbered from the left, beginning at zero. For example:

```
14 % echo one two three four
one two three four
15 % echo !14:2
echo two
two
```

A colon followed by a number is a *modifier* to a history substitution. A numeric modifer refers to a specific word of an event. A pair of numbers separated by a hyphen indicates a range of words. For example:

```
16 % echo !14:2-4
echo two three four
two three four
```

There are some special designators for commonly used words. A circumflex means word one, which is usually the first argument of the first command. For example:

```
17 % echo !14:^
echo one
one
```

A dollar sign means the last word. For example:

```
18 % echo !14:$
echo four
four
```

An asterisk means word one through the last word (as in ^-$). For example:

```
19 % echo !14:*
echo one two three four
one two three four
```

Other designators are described in Chapter 7.

### 3.4.4 Modifying Events

There are several non-numeric modifers that can be useful when you want to reexecute an event in modified form. Some of them are:

**p** prints the event, rather than executing it (useful for testing your knowledge of the history mechanism).

s/*old*/*new*/
replaces the first occurrence of the *old* string with the *new* string. This is the general form of the typo-correction feature mentioned earlier. You can omit the final slash if it would be the end of the line. For example:

```
20 % !3:s/michael/fred
write fred
```

**g** combined with another modifer, applies the change to each word of the command. For example, combined with the s modifier:

```
21 % echo aaa aaa aaa
aaa aaa aaa
22 % !!:gs/a/A
echo Aaa Aaa Aaa
Aaa Aaa Aaa
```

**h** (head) removes the last component from a pathname. For example:

```
23 % !9:^:h:p
/smith/projects/documents/commands
```

**t** (tail) removes all but the last component from a pathname. For example:

```
24 % !9:^:t:p
foobar
```

Other modifiers are described in Chapter 7.

### 3.5.5 Suppressing History Substitution

To suppress a history substitution, quote the exclamation point with a backslash. For example:

```
25 % echo \!9
!9
```

### 3.5.6 Saving the History List

There is a shell variable `savehist` that, if given a numeric value *n*, causes the C shell to automatically save and restore all or part of your history list. When you log out, the C shell writes *n* events into the file `~/.history`. When you log in, the C shell reads it into your history list so that you can pick up right where you left off. See the *C Shell Reference* for details.

# CHAPTER 4
# THE FILE SYSTEM

This chapter discusses the organization of the Concentrix file system. It covers types of files, pathnames, directories, files, and filename substitution, file protection, and links.

## 4.1 INTRODUCTION

From a user's the point of view, there are three kinds of files: directories, ordinary disk files, and special files.

- **Directories**

  Directories provide the mapping between the names of files and the files themselves, and thus impose a structure on the file system as a whole. Each user has a directory of his own files and can create a hierarchy of subdirectories to contain groups of files.

- **Disk Files**

  An ordinary disk file contains whatever information you write to it. The structure of files is controlled by the programs that use them, not by the system.

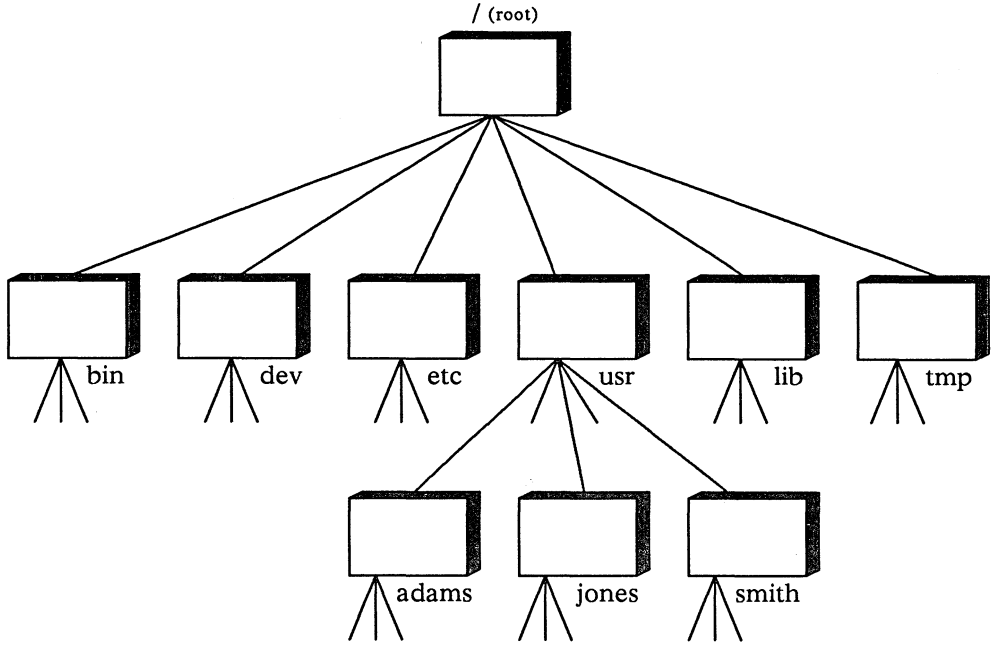
- **Special Files**

  Special files are associated with peripheral devices. Requests to read from or write to a special file results in the activation of the associated device. Special files are described in Chapter 5 and in the *System Reference Manual.*

The system does not impose any association between the name of a file and its type. Such associations exist only by convention.

## 4.2 DIRECTORIES

*Directories* provide the mapping between the names of files and the files themselves, and thus impose a structure on the file system as a whole. The directory structure has the form of a rooted tree, sometimes called a *hierarchy*, as shown below:




Except for the special entries described below, each directory appears as an entry in exactly one other directory, its *parent*. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs; the system controls the contents of directories. Anyone with appropriate permission can read a directory just like any other file.

Each user has a directory of his own. By creating new directories, each user can make his directory the root of a subtree, arranged for his own convenience. The most common example is the use of a directory to contain the source files, object files, makefiles, and so forth, that comprise a program.

The system maintains several directories for its own use. One of these is the root directory, which has the conventional name /. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Other system directories contain all the programs provided for general use; that is, all the *commands*.

Each directory always contains at least two entries. The entry . (sometimes called "dot") refers to the directory itself. Thus a program can read the current directory without knowing its complete path name. The entry .. (sometimes called "dot-dot") refers to the parent of the directory in which it appears. Thus a program can move up one level in the tree without knowing the complete path name.

Unless you specify otherwise, commands and programs use your *current working directory*. When you log in, your working directory is your *login directory*, which usually has your name.

### 4.2.1 Summary of Directory Commands

**mkdir** (make directory) creates a new directory.

**rmdir** (remove directory) deletes a directory.

**cd** (change directory) changes to a specified directory. With no arguments, *cd* returns to your login directory. The shell stores the pathname of its current working directory in the variable cwd.

**pwd** (print working directory) reports the absolute pathname of the working directory.

**pushd** (push directory) is the same as *cd* except that the shell saves the name of the current working directory on the directory stack before changing to the new one. With no arguments, *pushd* swaps your current directory with the top of the stack.

**popd** (pop directory) without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack.

**dirs** (directories) displays the contents of the *directory stack* with a tilde (~) as shorthand for your home directory. It is faster than *pwd*.

There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *Commands and Applications Manual* for details.

## 4.2 DISK FILES

An ordinary disk file contains whatever information you write to it. The structure of files is controlled by the programs that use them, not by the system.

Text files (source programs, documents, and so forth) consist of a sequence of bytes (characters) with lines delimited by newline characters.

Binary program files consist of a sequence of words as they will appear in core memory when the program starts executing.

A few user programs use more structure; for example, the assembler generates, and the loader expects, a binary object file in a particular format.

### 4.2.2 Summary of File Commands

This section demonstrates some of the most commonly-used file manipulation commands.

**ls** lists the names of files in a directory

**ls -l** gives more information. For example:

```
ls -l
-rw-rw-rw-      1 bwk   41 Jul 22 2:56 junk
-rw-rw-rw-      1 bwk   78 Jul 22 2:57 temp
```

From left to right, it shows: the protection codes, the number of hard links, the owner, the number of bytes, the date and time created, and the name.

**cat** is the simplest of all file printing programs. It simply prints on the terminal the contents of all the files named in a list. Thus:

```
cat junk temp
```

catenates (hence the name cat) the files and displays them on your terminal.

**pr** produces formatted printouts of files. As with *cat*, *pr* prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper.

**lpr** prints files on a line printer. It uses the environment variable `PRINTER` by default.

**mv** moves a file from one place to another (changes its directory entry). If you move a file to another one that already exists, the existing file's contents are lost. You also can use it to rename a file.

**cp** makes a *copy* of a file.

**rm** removes a link to a file (see Section 4.7). Files are not removed from the file system until the last hard link is removed.

## 4.4 FILE NAMES

Most filenames consist of up to 14 alphanumeric characters and periods. In fact, all printing characters except slash (/) can appear in filenames. It is inconvenient to use most non-alphabetic characters in filenames because many of them have special meaning to the shell.

A period is often used to separate the *extension* of a file name from the *base* of the name. Thus, these are four related files:

```
prog.c prog.o prog.errs prog.out
```

They share a base (that part of the name that is left when a trailing period and following characters that are not periods are stripped off). The file `prog.c` might be the source for a C program, the file `prog.o` the corresponding object file, the file `prog.errs` the errors resulting from a compilation of the program and the file `prog.out` the output of a run of the program.

Filenames that begin with a period are treated differently by programs such as *ls*. For example, *ls* does not list dot, dot-dot, `.login` or `.cshrc` unless you use the -a option.

## 4.5 PATHNAMES

*Pathnames* consist of a number of *components* separated by slashes. They have the following format:

[/] [directory/] ... filename

/ is a conventional name for the root directory, the top of the file system.

*directory* specifies the directory in which the next component resides.

/ separates components.

*filename* specifies a file that can be an ordinary file, a directory, or a special file.

A pathname, in effect, specifies the *path* of directories to follow to reach a file. Pathnames that begin with slash are *absolute* because they are specified from the absolute top of the entire directory hierarchy of the system (the *root*). For example:

```
/usr/bin/sort
```

The pathname above specifies the file `sort` in the directory `bin`, which is a subdirectory of `usr`, which is a subdirectory of the *root* directory `/`.

Pathnames that begin with a component are *relative* because they are specified from the current working directory. For example:

```
projects/outline
```

The pathname above specifes a file named `outline` in the directory `projects`, which is a subdirectory of the current working directory.

If a pathname contains no slashes at all (one component), the file is contained in the current working directory itself. The pathname is merely the name of the file. For example:

```
outline
```

The pathame above consists of the file `outline` in the current working directory. The null file name refers to the current directory itself.

## 4.6 FILENAME SUBSTITUTION

The system provides several metacharacters (sometimes called "wild card characters") that provide a form of "shorthand" notation for pathname components.

* An asterisk matches any string of characters, including the empty string. An asterisk is expanded by the shell before its command is executed. You can use it by itself, or as part of a file name. For example:

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

is equivalent to:

```
pr chap1*
```

An asterisk can be anywhere in a file name and can occur several times. The names that match are alphabetically sorted and placed in the *argument list* of the command. Thus:

```
rm *junk* *temp*
```

removes all files that contain `junk` or `temp` as any part of their name.

?

The question mark matches any single character in a filename. For example:

```
ls -l chap?.1
```

lists information about the first file of each chapter (`chap1.1`, `chap2.1`, etc.). Another example:

```
echo ? ?? ???
```

echoes a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length are independently sorted.

[]

Single-character numbers and letters within brackets specify a set. For example, [12349] matches any number in the set 1, 2, 3, 4, 9.

A hyphen within a set indicates a range. For example, [0–9] matches any number in the range zero through nine inclusive, and [a–z] matches any character in the range a through z inclusive. For example:

```
pr chap[1-49]*
```

prints only chapters 1 through 4 and chapter 9:

~

The tilde character expands into your home directory. For example, if you are `jones`:

```
~/mbox
```

would, on most systems, expand to:

```
/usr/jones/mbox
```

A tilde followed immediately by a user name expands to the *home* directory of the specified user. For example:

```
~smith
```

would, on most systems, expand to the pathname:

```
/usr/smith
```

On large systems, users can have login directories scattered over many different disk volumes with different prefix directory names. Thus, the tilde notation provides a reliable way of accessing other users' files.

{}

Strings contained in braces are consecutively substituted into the containing characters and the results expanded left to right. The strings are separated by commas. For example:

```
% echo a{b,c,d}e
abe ace ade
```

Brace expansions occur before other filename substitutions, and can be applied recursively (nested). The results of each expanded string are sorted separately, left to right, order being preserved.

The resulting filenames are not required to exist if no other substitution mechanisms are used. Thus, braces can be used to generate arguments that are not filenames, but that have common parts.

A typical use of brace expansion is in "factoring" pathnames. For example:

```
% echo root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

If an argument list contains filename substitution characters that fail to match any existing file names, the shell does not execute the command and prints a diagnostic:

```
No match.
```

None of the filename metacharacters match filenames that begin with a period. That prevents accidental matching of the filenames dot and dot-dot in the working directory (that have special meaning to the system), or other files such as `.cshrc` that are not normally visible.

### 4.6.7 Suppressing Filename Substitution

To *escape* (suppress the expansion of) a string that contains filename metacharacters, enclose it in single quotes. For example:

```
% echo '*'
*
```

The single quote character can be escaped with a preceding backslash (\) character:

```
% echo \'
'
```

These mechanisms can be combined. For example:

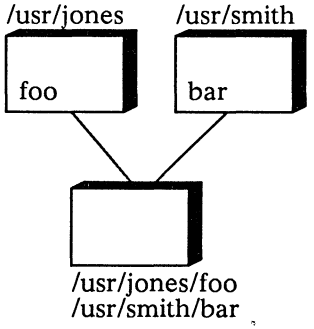
```
% echo \''*'
'*
```

## 4.7 HARD LINKS

A *hard link* is a directory entry for a file. It consists of the name of the file and a pointer to the information on disk that actually describes the file. Thus, each file exists independently of any directory entry.

> *Note: In general, the term "link" can be assumed to mean "hard link" rather than "symbolic link" (discussed in Section 4.8).*

A file can have several links to it. The only restriction is that hard links cannot cross from one file system to another and cannot refer to directories. Thus, a file can appear in several different directories under several different names. For example:




In the diagram, there are two user directories, each of which contains a link to the same file. Either user can manipulate the file, governed by the file's protection code (see Section 4.9). UNIX differs from other operating systems in which linking is permitted in that all links to a file have equal status.

Either user can remove his link without affecting the contents of the file. A file is removed from the file system only when the last link to it is removed.

Some programs may not treat hard links exactly as you might expect. For example, some editors operate by writing a new temporary file that assumes the name of the orginal file when you exit. The original file is either removed or renamed, *along with all of its links.*

## 4.7.1 Summary of Hard Link Commands

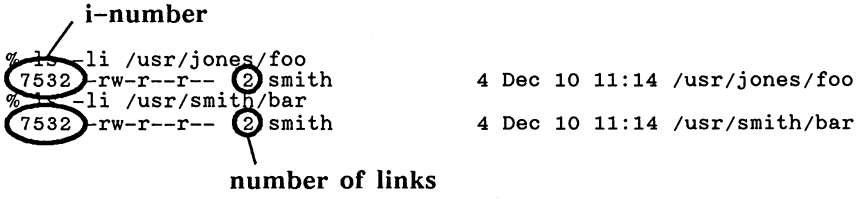
**ln** creates a hard link.

**rm** removes a hard link.

**ls -l** shows the number of hard links to a file.

**ls -i** displays the i-number (identification number) of each file.

For example, using the links in the diagram above:

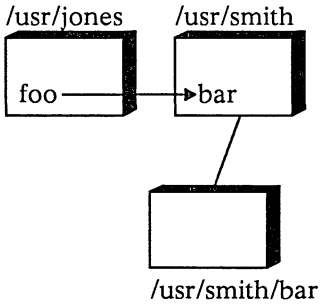



Although the directory entries are different, they clearly point to the same file. If one of the users were to delete the file, the system would simply remove his directory entry and decrement the number of links by one.

## 4.8 SYMBOLIC LINKS

A *symbolic link* is a directory entry that points to another directory entry. The kernel (not the C shell) resolves symbolic links when executing system calls. Thus, all filename substitution occurs before symbolic links are resolved.

Because they can refer to directories and can cross file systems, most users find symbolic links more useful than hard links for the purpose of sharing files. For example:




For most purposes, Jones can manipulate the symbolic link `foo` as if it were a file in his own directory. The exceptions are the commands *ls* and *rm*.

Should the referenced file or directory be removed (all hard links are gone), a symbolic link remains intact but causes an error when used.

### 4.8.1 Summary of Symbolic Link Commands

**ln -s** creates symbolic links. See the *Commands and Applications Manual* for details.

**rm** removes a soft link, not the file or directory that it refers to.

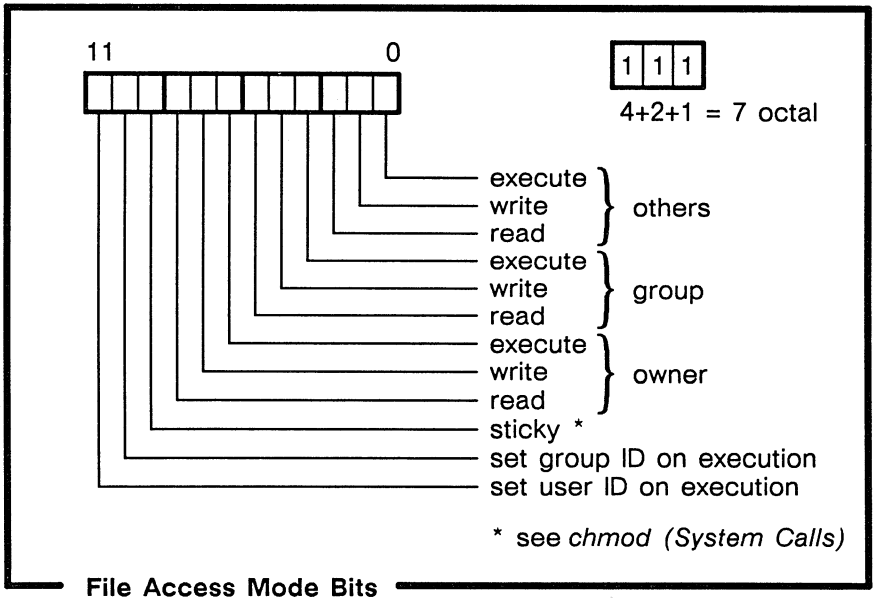
**ls** lists symbolic links in the form:

*linkname -> reference*

**ls -L** lists the file or directory referenced by the link, instead of the link itself.

## 4.9 PROTECTION

Each file is marked with the user identification (user ID) and group identification (group ID) of its owner. In addition, each file has a set of 12 access mode (protection) bits, as shown below:




**File Access Mode Bits**

To obtain an *access code*, convert the binary number formed by the access mode bits to an octal number. The binary-to-octal conversion technique is shown in the top-right corner above. For each three bits, form an octal digit by adding some combination of the numbers 4, 2, and 1. If set, the high-order bit is 4, the middle bit is 2, and the low order bit is 1.

The low-order nine bits specify read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

The high-order three bits specify special permissions for files that are executable programs:

- One bit (the sticky bit) is of concern only to systems programmers and is explained in the *Systems Programming Guide*.
- One bit causes the system to temporarily change the current group ID to that of the file during execution.
- One bit causes the system to temporarily change the current user ID to that of the file during execution.

The set-user-ID and set-group-ID features provide for privileged programs that can use files inaccessible to other users. For example, a program can keep an accounting file that can be neither read nor changed except by the program itself or a particular group of users.

The actual user ID and group ID of the invoker of any program are always available. Thus, programs can take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to restrict access to system administration commands.

Because anyone can set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention.

The system recognizes one particular user ID (the *super-user*) as exempt from the usual constraints on file access. For example, programs can be written to dump and reload the file system without unwanted interference from the protection system.

### 4.9.1 Summary of Protection Commands

**chmod** sets the access mode bits of a file. The syntax of chmod is extensive. See the *Commands and Applications Manual* for details.

**umask** set the file creation mask. A file creation mask is the *logical complement* of the desired file access mode as shown above.

# CHAPTER 5
# PERIPHERAL DEVICES

Peripheral devices are hardware units that perform input and/or output, for example, disks, tapes, communication lines, and printers. This chapter provides fundamental information about the use of peripheral devices on the Concentrix operating system.

## 5.1 SPECIAL FILES

On the Concentrix operating system, accessing a device is very similar to accessing a file. Each peripheral device is associated with at least one *special file*, which behaves just like an ordinary file except that requests to read from or write to a special file activate an associated device. Thus, any program expecting an ordinary file name as a parameter can be passed a special file name. Each type of special file is described in detail in the *System Reference Manual.*

Special files exist for:

- each communication line.
- each disk.
- each tape drive.
- physical and kernel memory.

A directory named /dev is reserved by convention for special files. To list the special files present on your system, type:

```
% ls -l /dev | more
```

Because a special file is not a regular disk file or directory, no size is printed. Instead, the *major* and *minor device* numbers are shown. The major device number specifies the type of device associated with the special file. The minor device number usually specifies one of several devices or a subdevice attached to a specific controller.

## 5.1.1 Special File Names

Special file names have the format:

**/dev/[n][r]xxpu[d]**

**/dev** is the name of the directory in the file system where all special files normally exist.

**n** optionally inhibits the rewinding of a magnetic tape.

**r** optionally specifies the raw device interface (as explained in Section 5.1.2). The default is the block device interface.

*xx* is a two-character string that specifies a device type. The Concentrix operating system supports the following devices:

| | |
|---|---|
| **ds** | hard disk (Qualogy controller, 5.25 inch ST506 interface) |
| **xd** | hard disk (Xylogics controller, SMD interface) |
| **xt** | magnetic tape (Xylogics controller, Pertec interface) |
| **fd** | floppy diskette (5.25 inch, 96 TPI, 80-track) |
| **st** | streaming tape (5.25 inch, QIC02/QIC24, .25 inch tape) |
| **ld** | line printer (Data Products interface) |
| **tty** | terminal |
| **pty** | pseudo-terminal (network) |

*p* is a single hexadecimal digit that specifies a controller. Each controller has a unique number that maps to its Multibus address, allowing controllers to change Multibus location while retaining the same number.

*u* is a single digit that specifies a unit.

*d* is a single alphabetic character whose meaning depends on *xx*. If *xx* is a disk, *d* specifies a partition (as explained in Section 5.2.3). If *xx* is a magnetic tape drive, *d* specifies a density as follows:

| | |
|---|---|
| **l** | 800 BPI/NRZI |
| **m** | 1600 BPI/PE |
| **h** | 6250 BPI/GCR |

## 5.1.2 Interfaces: Character and Block

There are two different interfaces between peripheral devices and the operating system: the *block* interface and the *character* interface.

The block interface transfers data in blocks native to the device and is used primarily for mass-storage devices: disks and magnetic tapes. The block interface allows the kernel to use buffering in order to reduce I/O overhead.

The character interface transfers data in variable-sized blocks (up to a system-defined limit) and is used primarily for terminals, printers, and so forth. Most mass-storage devices also have a character interface (often called the "raw" interface) that allows direct (unbuffered) access to the device. The raw interface is used by system software, for example, bootstrap procedures, *fsck*, and *dump*, and disk copies using *dd*, because it tends to work faster in some cases.

When working with devices, there are cases in which it is important to use the appropriate interface (as specified in the documentation). Be careful not to use the wrong one indiscriminately.

### 5.1.3 Device Protection

Special files are subject to the same protection mechanism as regular files. With improper protection values, important resources such as the current system image in memory or entire contents of disks can be inadvertantly destroyed.

## 5.2 DISKS AND FILE SYSTEMS

Although the root of a file system is always stored on the root device, parts of a file system hierarchy can reside on other devices. In a typical installation, the root directory resides on a small partition (see Section 5.2.3) of one of the disk drives, while user files reside on another partition, which may or may not be on the same disk drive.

Because there is a single root, a user can access the entire file system without having to know anything about the number or types of disks present on the system. The hardware configuration is relevant only to system administrators and those responsible for backups. In contrast, other operating systems require users to specify individual disks.

There is only one exception to the rule of identical treatment of files on different devices: *no hard link can exist between one file system hierarchy and another.* This restriction is enforced in order to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the mountable file system is dismounted. Symbolic links, however, can cross device boundaries.

### 5.2.2 Mounting File Systems

The *mount* command (see *Administration Commands*) accepts two arguments: the name of an existing ordinary file (usually an empty directory at root level), and the name of a block device special file whose associated storage volume (a disk) has the structure of an independent file system containing its own directory hierarchy.

The effect of *mount* is to cause references to the ordinary file to refer instead to the root directory of the file system on the disk. In effect, *mount* replaces a node of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the disk). After the mount, there is virtually no distinction between files in the mounted file system and those in the root file system.

When Concentrix boots, it executes the command *mount -a*, which mounts all of the file systems as specified in the file `/etc/fstab` (described in the *System Reference Manual, File Formats.*)

### 5.2.3 Disk Partitions

A physical disk can be divided into 1 to 26 logical disks called *partitions*. Each partition can be used for a raw data area (such as a paging area) or to contain a file system. Partitions can overlap.

Partitions are named alphabetically: a through z. The names are intended to be mnemonic, for example, the swapping partition is often named s. These are the partition names used by Concentrix:

| | |
|---|---|
| a | alternate track area (Qualogy controller) |
| b | boot area (variable size starting at first cylinder) |
| c | reserved for Alliant |
| d | diagnostic area (last cylinder) |
| e | reserved for Alliant |
| f | used by Alliant for default system configuration |
| g | reserved for user partition |
| h | used by Alliant for default system configuration |
| i | reserved for user partition |
| j | reserved for user partition |
| k | reserved for user partition |
| l | reserved for user partition |
| m | used by Alliant for default system configuration |
| n | reserved for user partition |
| o | reserved for user partition |
| p | paging area (reserved for Alliant) |
| q | reserved for user partition |
| r | used by Alliant for default system configuration |
| s | swap area (150Mb) |
| t | reserved for user partition |
| u | reserved for user partition |
| v | reserved for user partition |
| w | entire disk, except diagnostic cylinder |
| x | reserved for Alliant |
| y | reserved for Alliant |
| z | reserved for Alliant |

A typical disk contains several partitions, not all of which contain file systems. The prototypes for the partitions on specific disks can be found in the file `/etc/disktab`. For any device, the program *diskf* (*Administration Commands*) can display the partition table stored in the disk header (a data structure on the first block of the disk).

## 5.3 MAGNETIC TAPE

Alliant systems support two kinds of magnetic tape drive: 9-track and ¼-inch "streaming" tape. If you use 9-track tape, make sure to specify the proper density as described in Section 5.1.1 You do *not* have to set the density switch on the drive. (If you have difficulty writing a magtape at your specified density, the drive may be improperly configured. Call Alliant Customer Support for assistance.)

To store files on magnetic tape, use the program *tar*, which is described in the *Commands and Applications Manual*. A typical tar command looks something like:

```
% tar cvf /dev/rst00 memos
```

The c argument means "create," which writes to the tape, overwriting any data already there. To read from the tape, you would use x (for "extract") instead of c. The v argument means "verbose," which displays all of the files copied, and the f argument means "file" which allows you to specify a device (special file), in this case, the streaming tape drive. The last argument, memos, is the name of the file to write, which is

normally a directory. (When writing files to tape, make sure to use a relative pathname; otherwise, you can extract the files only to the same absolute pathname from which they were written.)

There is another program *dump* that is used primarily for incremental file system backups. You can, however, use it instead of *tar* if you prefer. It is described in the *Administration Commands Manual.*

# CHAPTER 6

# PROGRAMMING THE C SHELL

This chapter describes how to program the C shell by writing command files called *shell scripts*. Any command that you can use interactively can be executed in a shell script. In addition, the C shell provides some features similar to the C language that are used primarily for writing scripts.

- **Variables**

  The C shell maintains two discrete sets of variables:

  - *Shell variables* are local (can be accessed only within the current C shell).
  - *Environment variables* are global (passed from the current C shell to all child processes).

  The contents of variables can be used in commands through a *substitution mechanism*.

- **Control structures**

  The C shell provides control statements similar to those in the C language: if, if-then-else, switch, foreach, while, and goto.

- **Interrupt handling**

  C shell scripts can include a block of code to which control is transferred when an interrupt is received.

- **In-line data**

  C shell scripts can contain blocks of in-line data for use by commands.

- **Special shell scripts**

  There are special scripts that the C shell automatically runs when you log in, log out, or invoke a subshell.

## 6.1 INVOKING SHELL SCRIPTS

A C shell command script can be executed by using the *csh* command:

csh [*–arg*] *script* [*arg...*]

*–arg* is one or more C shell options as described in the *Commands and Applications Manual*. A few commonly used options are:

- *–v* sets the `verbose` variable so that command input is echoed after history substitution.
- *–v* sets the `echo` variable so that command input is echoed immediately before execution.
- *–n* causes the C shell to read commands but not execute them.

*script* is the name of a C shell script (a file containing commands).

*arg* is one of a sequence of arguments passed to the C shell script through the same mechanisms used to reference other shell variables (see Section 6.3).

For example, assume that the file `tryout` contains the following commands:

```
# this is a C shell script
as source
mv a.out testprog
testprog
```

The *csh* command would invoke a C shell to execute the commands sequentially:

```
% csh tryout
```

The file `source` would be assembled, the resulting program renamed `testprog`, and `testprog` executed. The effect is as if you had typed the contents of the file on the terminal. Giving a C shell script execute permission eliminates the need to type `csh`:

```
% chmod +x tryout
```

Thereafter, to invoke `tryout`, you need only type:

```
% tryout
```

To add `tryout` to the commands in your search path, type:

```
% rehash
```

## 6.2 COMMENTS

The sharp sign character (#) can be used anywhere a C shell script to prefix a comment line.

*Note: All C shell scripts must begin with a comment line.*

Scripts that do not begin with a comment are executed by the Bourne shell, a convention that allows shell scripts for both shells to be used transparently. One of the must common bugs in writing a C shell script is forgetting the intial comment line.

## 6.3 SHELL VARIABLES

Shell variables are arrays of strings. Variable names consist of up to 20 letters (including underscore) and digits, beginning with a letter. The *set* command assigns a string value to a variable:

```
% set flavor=vanilla
```

*Variable substitution* replaces the name of a variable by its value. It is invoked by the dollar sign ($) character:

```
% echo $flavor
vanilla
```

Without the dollar sign, no substitution is performed:

```
% echo flavor
flavor
```

To create an array, assign a discrete value to each element by enclosing a list of values in parentheses. For example:

```
% set flavor=(vanilla chocolate coffee)
% echo $flavor
vanilla chocolate coffee
```

To access individual elements of an array, use index numbers (counting from one) enclosed in brackets. For example:

```
% set $flavor[2]=peach
% echo $flavor
vanilla peach coffee
```

To access a range of elements, separate the index numbers with a hyphen. For example:

```
% set flavor=(vanilla chocolate coffee strawberry)
% echo $flavor[2-4]
chocolate coffee strawberry
```

If omitted, the second index number defaults to the last element. For example:

```
% echo $flavor[3-]
coffee strawberry
```

It is an error to invoke substitution on the name of a variable that has not been set:

```
% echo $xxx
xxx: Undefined variable.
```

To find out if a variable exists, use a question mark between the dollar sign and the name of the variable. This mechanism expands to `1` if the variable exists (has been assigned a value) or to `0` otherwise. For example:

```
% echo $?xxx
0
```

To find out the number of elements in a variable, use a sharp sign between the dollar sign and the variable name. For example:

```
% set flavor=(vanilla chocolate coffee strawberry)
% echo $#flavor
4
```

To suppress variable substitution, enclose the string in single quotes or precede the dollar sign with a backslash. Double quotes do not work. For example:

```
% echo '$flavor'
$flavor
% echo \$flavor
$flavor
% echo "$flavor"
vanilla chocolate coffee strawberry
```

### 6.3.1 Variable Substitution Modifiers

Variable substitution provides some of the same modifiers available in history substitution. (A full list is provided in Chapter 7). Some of them are:

**h** (head) removes a trailing pathname component, leaving the head.

**r** (root) removes a trailing extension component, leaving the root name.

**e** (extension) removes all but the extension component.

**t** (tail) removes all leading pathname components, leaving the tail.

**g** (global) applies the change globally, prefixing another modifier.

For example:

```
% set name=/mnt/foo.bar
% echo $name:r
/mnt/foo
% echo $name:e
bar
```

The `r` modifier strips off the trailing extension and the `e` modifier strips off everything but the extension.

### 6.3.2 Reading the Standard Input

Variable substitution includes a special notation that is useful for writing shell scripts that read commands from the terminal or act as a filter, reading lines from an input file.

$< is replaced by the next line of input read from the shell's standard input (not the script it is reading).

For example, this script would write out the specified prompt without a newline, then read the answer into the variable a:

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

The variable a would contain a null value if either a blank line or an end-of-file was typed.

### 6.3.3 Special Variables

Some shell variables directly affect the behavior of the shell and/or are predefined by the shell. Some of the special variables (echo, ignoreeof, and so forth) are boolean; the shell only recognizes if they are set or not. These descriptions are very brief. Refer to Chapter 7 for details.

**argv** contains command line arguments (see Section 6.3.3.1).

**cdpath** specifies the search path for commands that change the current directory.

**cwd** contains the full pathname of the current directory.

**echo** echoes each command and its arguments just before execution.

**histchars** changes the characters used in history substitution.

**history** controls the size of the history list.

**home** contains your home directory.

**ignoreeof** prevents exit on end-of-file from input devices that are terminals.

**mail** specifies the files where the shell checks for mail.

**noclobber** restricts output redirection to prevent accidental destruction of files and appendages to nonexistant files.

**noglob** suppresses filename substitution.

**nonomatch** suppresses errors from nonmatching filename substitutions.

**notify** notifies you asynchronously of job completions.

**path** specifies the search path for command execution and corresponds to the environment variable PATH (see Section 6.4).

**prompt** specifies the string printed before each command.

**savehist** controls the number of history list entries of that are saved at logout.

**shell** contains the name of the file in which the shell resides.

**status** contains the status returned by the last command.

**term** specifies the type of terminal in use and corresponds to the environment variable TERM (see Section 6.4).

**time** controls automatic timing of commands.

**user** specifies the current user name and corresponds to the environment variable USER (see Section 6.4).

**verbose** prints the words of each command after history substitution.

`Argv`, `cwd`, `home`, `path`, `prompt`, `shell` and `status` are predefined by the shell. `Cwd` and `status` are updated continuously; the others are set only at initialization.

#### 6.3.3.1 Special Substitutions on the Variable Argv

The special variable `argv` contains the command line arguments passed to a shell script. To help you parse the contents of `argv`, some special notation is provided:

$*n* is shorthand for `$argv[`*n*`]` but does not yield an error if *n* is out of range as the longer form does.

`$*` is shorthand for `$argv`.

You can set the special variable `noglob` to prevent filename substitution of the elements of `argv`. This is recommended if the arguments to a shell script are filenames that have already been expanded or if the arguments contain filename substitution metacharacters.

## 6.3.4 Using Variables in Calculations

Shell variables can be used in numeric calculations. Strings of digits are treated as integers (the null string is treated as zero). Strings containing non-numeric characters are invalid. The second and subsequent words of multiword values are ignored.

The @ command evaluates a numeric expression and assigns the result (in the form of a numeric string) to the specified variable. For example:

```
% @ x = 2
% @ y = 3
% @ z = $x * $y
% echo $z
6
```

All the arithmetic operators of the C language are available with the same precedence. (See Chapter 7 for a complete list.) In particular, == and != compare strings and && and || perform boolean and/or operations.

The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (like asterisk, question mark, or brackets) and test whether the string on the left matches the pattern on the right.

## 6.3.5 File Expressions

The C shell includes a mechanism that provides information about files. Use an operand of the following form in a numeric expression:

*-c filename*

where *c* is one or more of the following characters and *filename* is the name of a file.

| | |
|---|---|
| **d** | directory |
| **e** | existence |
| **f** | plain file |
| **o** | ownership |
| **r** | read access |
| **w** | write access |
| **x** | execute access |
| **z** | zero size |

The expression returns true (1) or false (0). For example, the following commands test whether the file `myfile` exists:

```
% @ temp = -e myfile
% echo $temp
1
```

### 6.3.6 Command Expressions

The C shell includes a mechanism that tests whether a command terminates normally. Use an operand of the following form in a numeric expression:

```
{ command }
```

The expression returns `1` (true) if the command succeeds (exits normally with exit status zero), or `0` (false) if the command fails (terminates abnormally or with exit status non-zero.) For example:

```
% @ temp = { echo }
% echo $temp
1
```

For more detailed information about the execution status of a command, examine the shell variable `status`. Because it is set by every command, you must save the value of `status` in order to use it in any but the immediately subsequent command.

### 6.3.7 Summary of Variable Commands

| | |
|---|---|
| **set** | shows the value of all variables currently defined in the shell. |
| **set** *name=word* | assigns the value *word* to the variable *name*. |
| **set** *name=(word...)* | assigns a multiword value to the variable *name*. |
| **set** *name[n]=word* | assigns the value *word* to element *n* of variable *name*. |
| **unset** *name* | deletes the variable *name*. |
| @ | shows the value of all variables currently defined in the shell. |
| @ *name=expr* | assigns the result of the expression *expr* to the variable *name*. |
| @ *name[n]=expr* | assigns the result of the expression *expr* to element *n* of the variable *name*. |

## 6.4 ENVIRONMENT VARIABLES

Environment variables are maintained independently from other shell variables. They are automatically passed by the C shell to each subshell and are available to user programs via the system call *execve* (*System Reference Manual)*. By convention, are environment variables are upper-case and environment strings have the form *name=value*. The following commands operate on environment variables:

| | |
|---|---|
| **setenv** | creates or changes environment variables. |
| **unsetenv** | removes enviroment variables. |
| **printenv** | lists environment variables. |

The following are the default environment variables:

| | |
|---|---|
| **EXINIT** | is a list of startup commands read by *ex*, *edit*, and *vi*. |
| **HOME** | is your login directory, set by *login* from the password file. |

**PATH** is the sequence of directory prefixes that *sh*, *time*, *nice*, etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colon. The *login* program sets the following value:

```
PATH=:/usr/ucb:/bin:/usr/bin
```

**PRINTER** is the name of the default printer to be used by *lpr*, *lpq*, and *lprm*.

**SHELL** is the file name of the your login shell.

**TERM** is the kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*, that can exploit special terminal capabilities. See *termcap* (*System Reference Manual: File Formats)* for details.

**TERMCAP** is the string describing the terminal in `TERM`, or the name of the termcap file.

**USER** is your login name.

The environment variables, `USER`, `TERM`, and `PATH` have a special correspondence to the shell variables `user`, `term`, and `path`. Changing one of them automatically changes its corresponding shell variable. Similarly, changing of the the shell variables automatically changes the corresponding environment variable.

## 6.5 CONTROL STRUCTURES

The C shell provides control structures similar to those in the C language.

### 6.5.1 The If Statement

The if statement provides a simple branching mechanism. It has the form:

```
if ( expression) command
```

*expression* is an numeric, filename, or command expression as described in Section 6.3. If *expression* returns true (non-zero), the *command* is executed.

*command* cannot be: a pipeline, a background command, commands separated by semicolons, or another control command.

You also can write the if statement this way if you prefer:

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

The backslash must *immediately* precede the end-of-line.

## 6.5.2 The If-Then-Else Statement

The if-then-else statement provides a multi-way branching mechanism. It has the following general form.

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

*expression* is an numeric, filename, or command expression as described in Section 6.3.

*command* is any valid command.

The shell evaluates *expressions* until it finds one that returns true (non-zero) and executes its block of *commands*. If control reaches the else clause, its block of *commands* are executed.

The else if clauses and the else clause are optional; you can have several else if clauses but only one else clause.

The placement of the keywords if an if-then-else statement is *not* flexible. The following two formats are *not* valid:

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

and

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

## 6.5.3 The Switch Statement

The switch statement is similar to the case structure in C: a dispatch-table mechanism. It has the general form:

```
switch ( word )

case str1:
        command
          :
        breaksw
   :
case strn:
        command
          :
        breaksw
default
        command
          :
        breaksw
endsw
```

*word* is the string on which control is dispatched.

*str1*
*strn* is a string that is matched against *word*.

*command* is any valid command.

The shell searches for the first case string that matches the dispatch *word* and executes the commands in its clause. If none match, the shell executes the commands in the default clause, if present. Refer to the *C Shell Reference* for details.

> *Note: a very common mistake in shell scripts is to use* break *rather than* breaksw *in* switch *statements.*

If a case string matches the dispatch *word* but has no command block, control passes to the next command block. Thus, you can have a construct like:

```
case str1:
case str2:
case str3:
        command
          :
        breaksw
```

### 6.5.4 The Foreach Statement

The foreach statement provides a simple iterated loop mechanism based on a list of string values. It has the form:

```
foreach name ( wordlist )
        command
          :
end
```

*name* is the name of a variable that acts as a loop index.

*wordlist* is a list of strings that are successively assigned to *name*.

*command* is any valid command.

The loop iterates once for each value in the *wordlist*, assigning the value to the loop index variable *name*, which can be accessed by variable substitution.

You can branch to the end of a foreach loop. The continue statement causes the remaining commands on the line to execute, then transfers control to the end statement, beginning another iteration. For example, consider the script `temp1`:

```
#
foreach i (a b c)
        echo $i
        if ($i == b) continue
        echo test failed
end

% temp1
a
test failed
b
c
test failed
```

You also can branch out of a foreach loop. The break statement causes the remaining commands on the line to execute, then transfers control to the statement after the end statement, terminating the loop. For example, consider the script `temp2`:

```
#
foreach i (a b c)
        echo $i
        if ($i == b) break
        echo test failed
end

% temp2
a
test failed
b
```

#### 6.5.4.1 Foreach Loops at the Terminal

It is occasionally useful to use the foreach statement at the terminal to aid in performing a number of similar commands. For example, to search several user directories for the file `foobar`, you could type:

```
% find /usr/jones -name foobar -print
% find /usr/smith -name foobar -print
% find /usr/adams -name foobar -print
% find /usr/rogers -name foobar -print
```

These commands are very similar; thus, you can use foreach to do the same thing:

```
% foreach i (jones smith adams rogers)
? find /usr/$i -name foobar -print
? end
```

The shell prompts for input with a question mark when reading the body of the loop.

Very useful with loops are variables that contain lists of filenames or other words. For example:

```
% set list=(`ls`)
% foreach i ($list)
        :
        :
```

The *set* command uses command substitution (see Chapter 3) to assign the names of all the files in the current directory to the variable `list`. The foreach loop iterates over the value of `list` to perform any chosen function.

### 6.5.5 The While Statement

The while statement provides a simple test-and-loop structure. It has the form:

```
while ( expression )
        command
            :
            :
end
```

*expression* is an numeric, filename, or command expression (see Section 6.3).

*command* is any valid command.

The loop iterates as long as the *expression* returns true (non-zero). If the *expression* is false upon reaching the while statement, the loop does not iterate at all.

You can branch to the end of a while loop. The `continue` statement causes the remaining commands on the line to execute, then transfers control to the end statement, beginning another test and iteration. For example, consider the script `temp3`:

```
#
@ i = 0
while ($i <= 5)
        @ i += 1
        echo $i
        if ($i <= 3) continue
        echo test failed
end

% temp3
1
2
3
4
test failed
5
test failed
6
test failed
```

You also can branch out of a while loop. The `break` statement causes the remaining commands on the line to execute, then transfers control to the statement after the `end` statement, terminating the loop. For example, consider the script `temp4`:

```
#
@ i = 0
while ($i <= 5)
        @ i += 1
        echo $i
        if ($i > 3) break
end

% temp4
1
2
3
4
```

### 6.5.6 The Goto Statement

The C shell also provides the `goto` statement, with labels looking as they do in C. It has the form:

```
name:
        command
          :
        goto name
```

## 6.6 HANDLING INTERRUPTS

You can program a shell script to retain control when an interrupt is received. For example, a script that creates temporary files might want to handle interrupts in order to clean up after itself.

To enable interrupt handling, use the onintr statement, which has the form:

```
onintr label
```

*label* is an existing label in the shell script. If you omit *label*, interrupt handling is disabled.

When an interrupt is received, the shell transfers control to the command following the specified label. Interrupt handling remains in effect; thus, the subsequent commands (the *interrupt handler*) are responsible for exiting from the shell script. For example:

```
# idemo
onintr ihand
while (1)
        echo foobar
end
ihand:
echo signal received
exit

% idemo
foobar
foobar
foobar
^Csignal received
%
```

The *exit* command (built in to the C shell) exits from the shell script. The default exit status is zero. To exit with a status of one, for example:

```
exit(1)
```

## 6.7 INCLUDING DATA IN SHELL SCRIPTS

By default, commands that run from shell scripts receive the standard input of the shell that is running the script. This allows shell scripts to participate in pipelines, but requires extra notation for commands that are to take inline data.

For example, consider the script temp:

```
#
cat << 'EOF'
aaa
$bbb
ccc
'EOF'

% temp
aaa
$bbb
ccc
```

The notation << ´EOF´ means that the standard input for the *cat* command is to come from the text in the shell script file up to the next line consisting of exactly ´EOF´.

Because the 'EOF' is enclosed in single quotes, the shell does not perform variable substitution or command substitution on the data. In general, if any part of the word following the angle brackets is quoted, substitutions are not performed. For example, consider the script `temp2`:

```
#
cat << EOF
aaa
$bbb
ccc
EOF

% temp
bbb: Undefined variable.
```

## 6.8 SPECIAL SHELL SCRIPTS

Your home directory can contain several shell scripts that are executed automatically by the C shell:

`.login` executes each time you log in.

`.cshrc` executes each time a new C shell starts up.

`.logout` executes each time you log out.

When you log in, the system starts up a C shell called a *login* shell that begins by executing commands from the file `.cshrc`. All C shells that you start during your terminal session also execute this file. If `.cshrc` does not exist, the shell does not complain about its absence.

> *Note: If you place a large number of commands in* .cshrc, *shells will tend to start slowly. Try to limit the number of aliases you have to a reasonable number (10 or 15). As many as 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.*

A login shell then reads and executes `.login`. A typical `.login` file might look something like this. (The lines are shown numbered for reference only.)

```
1.      set ignoreeof
2.      set mail=(0 /usr/spool/mail/jones)
3.      alias ts 'set noglob;eval `tset -s -q`';
4.      ts; stty intr ^C kill ^U crt
5.      set time=15 history=10
6.      msgs -f
```

1. Set the shell variable `ignoreeof` so that typing CTRL D does not log you off the system. (Use the *logout* command to log out.)
2. Set the shell variable `mail` to the pathname of your mailbox. This causes the shell to watch for incoming mail. Every five minutes, the shell looks for this file and tells you if more mail has arrived there. An alternative is:

```
biff y
```

The *biff* command notifies you immediately when mail arrives, and shows you the first few lines of the new message.

3. Create an alias `ts` that uses the *tset* command to set up the modes of your terminal.
4. Executes `ts` and uses the *stty* command to change the interrupt character to CTRL C and the line kill character to CTRL U.
5. Set the shell variable `time` to `15`, causing the shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time, and the variable `history` to 10, causing the shell to remember your last ten commands in its history list, (see Chapter 3).
6. Run the *msgs* program to display any system messages that you have not seen before; the `-f` option prevents *msgs* from typing anything if there are no new messages.

# CHAPTER 7
# C SHELL REFERENCE

The C shell is a command language interpreter incorporating a history mechanism, job control facilities, and a C-like syntax. This chapter provides detailed descriptions of the various aspects of the C shell.

## 7.1 OPERATION

The C shell begins operation by executing commands from the file `.cshrc` in the home directory of the invoker. A login C shell also executes commands from the file `.login` in the home directory. When a login shell terminates, it executes commands from the file `.logout` in the users home directory.

### 7.1.1 Commands

A simple command is a sequence of words consisting of up to 1024 characters. The first word specifies the command to be executed. The subsequent words specify arguments to the command. Argument lists consists of up to 10240 characters. The number of arguments to a command that involves filename substitution (see Section 7.7) is limited to 1706 (one sixth the number of characters allowed in an argument list.)

A simple command or a sequence of simple commands separated by a vertical bar (|) forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines can be separated by semicolons (;) and are then executed sequentially. A sequence of pipelines can be executed without immediately waiting for it to terminate by following it with an ampersand (&).

Any of the above can be placed in parentheses to form a simple command (which can be a component of a pipeline, etc.) It is also possible to separate pipelines with a double vertical bar (||) or a double ampersand (&&) indicating, as in the C language, that the

second is to be executed only if the first fails or succeeds respectively. (See Section 7.9.2.)

In the normal case, the shell repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

## 7.1.2 Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions:

- The following characters form separate words:

  | | |
  |---|---|
  | & | ampersand |
  | \| | vertical bar |
  | ; | semicolon |
  | < | left angle bracket |
  | > | right angle bracket |
  | ( | left parenthesis |
  | ) | right parenthesis |

- Doubled characters form single words:

  | | |
  |---|---|
  | && | ampersand |
  | \|\| | vertical bar |
  | << | left angle bracket |
  | >> | right angle bracket |

These parser metacharacters can be made part of other words (prevented their special meaning) by preceding them with backslash (\). A newline preceded by a backslash is equivalent to a blank.

Strings enclosed in matched pairs of single quotes, open single quotes or double quotes, form parts of a word. Metacharacters in quoted strings, including blanks and tabs, do not form separate words. These quotations have semantics described in Section 7.3. Within pairs of single quotes or double quotes, a newline preceded by a backslash gives a true newline character.

When the shell's input is not a terminal, the sharp sign (#) introduces a comment that continues to the end of the input line. It is prevented this special meaning when preceded by backslash and in quotations using single quote, open single quote, and double quote.

## 7.1.3 Substitutions

The C shell performs various input transformations in the following order. Each transformation is described in its own section:

1. **History Substitution**

   History substitution places words from previous commands as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

2. **Quotation**

   Enclosing a string in single quotes and prevents all substitutions except history substitutions. Enclosing a string in double quotes prevents alias subsititutions and filename substitutions.

3. **Alias Substitution**

   Alias substitution automatically replaces commands with other commands, providing a form of user-defined "shorthand" notation.

4. **Variable Substitution**

   Variable substitution replaces variable names with values from a table.

5. **Command Substitution**

   Command substitution replaces a command with its own output.

6. **Filename Substitution**

   Filename substitution replaces a word with a list of file names, selected according to a specified pattern.

Command and filename substitutions are applied selectively to the arguments of built-in commands. Portions of expressions that are not evaluated are not subjected to these substitutions. For commands that are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

### 7.1.4 Command Execution

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

When a command to be executed is found to not be a built-in command, the shell attempts to execute the command via *execve (System Reference Manual: System Calls)*.

Each word in the variable *path* names a directory from which the shell attempts to execute the command. If it is given neither a –c nor a –t option, the shell hashes the names in these directories into an internal table so that it only tries an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a –c or –t argument, and in any case for each directory component of *path* that does not begin with a slash, the shell concatenates with the given command name to form a path name of a file that it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus `(cd ; pwd) ; pwd` leaves you in your current directory while `cd ; pwd` leaves you in the *home* directory. Parenthesized commands are most often used to prevent *cd* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, it is assumed to be a file containing shell commands and a new shell is spawned to read it.

If there is an *alias* for *shell*, the words of the alias are prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (`$shell`). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

## 7.1.5 Command Input/Output

The standard input and standard output of a command can be redirected with the following syntax:

| | |
|---|---|
| < name | opens file *name* (first variable, command and filename expanded) as the standard input. |
| << word | reads the shell input up to a line that is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. A quoting backslash, double quote, single quote or open single quote in *word* prevents variable and command substitution from being performed on the intervening lines, allowing backslash to quote dollar sign, backslash and open single quote. Commands that are substituted have all blanks, tabs, and newlines preserved, except for the final newline, which is dropped. The resultant text is placed in an anonymous temporary file that is given to the command as standard input. |
| > name<br>>! name<br>>& name<br>>&! name | opens the file *name* as standard output. If the file does not exist, it is created; if the file exists, its is truncated, its previous contents being lost.<br>If the variable *noclobber* is set, the file must not exist or be a character special file (e.g. a terminal or `/dev/null`) or an error results. This helps prevent accidental destruction of files. In this case the exclamation point forms can be used and suppress this check.<br>The forms involving ampersand route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way that redirected input filenames are. |
| >> name<br>>>& name<br>>>! name<br>>>&! name | Uses file *name* as standard output like right angle bracket but places output at the end of the file. If the variable *noclobber* is set, it is an error for the file not to exist unless one of the exclamation point forms is given. Otherwise similar to right angle bracket. |

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. Note that the default standard input for a command run detached is not modified to be the empty file `/dev/null`; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, the process will block and the user will be notified (see Section 7.8.)

Diagnostic output can be directed through a pipe with the standard output. Simply use the form `|&` rather than just `|`.

### 7.1.6 Status Reporting

The C shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked (no further progress is possible), but only just before it prints a prompt so that it does not otherwise disturb your work. If, however, you set the shell variable *notify*, the shell notifies you immediately of changes of status in background jobs. There is also a shell command *notify* that marks a single process so that its status changes are immediately reported. By default *notify* marks the current process; simply type `notify` after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you are warned: `You have stopped jobs.` You can use the *jobs* command to see what they are. If you do that or immediately try to exit again, the shell does not warn you a second time; the suspended jobs are terminated.

## 7.2 HISTORY SUBSTITUTION

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

History substitutions begin with an exclamation point (!) and can begin anywhere in the input stream (but cannot be nested). History substitutions also occur when an input line begins with circumflex (as described later.) Any input line that contains history substitution is echoed on the terminal after substitution but before execution.

(Exclamation point can be preceded by a backslash (\) to prevent its special meaning; for convenience, exclamation point is passed unchanged when it is followed by a blank, tab, newline, equal sign or left parenthesis.)

Commands from the terminal that consist of one or more words are saved on the history list. The size of the list is controlled by the *history* variable; the previous command is always retained, regardless of size of the list.

Each *event* (command line) is numbered sequentially from 1 as shown in the following output from the *history* command:

```
 9      write michael
10      ex write.c
11      cat oldwrite.c
12      diff *write.c
```

Event numbers can be made part of the shell prompt by placing an exclamation point in the prompt string. You can refer to events in several ways. For example, given that the current event is 13:

| | |
|---|---|
| !*n* | absolute event number as in `!11`. |
| !-*n* | relative event number as in `!-2` for event 11. |
| !*string* | a prefix of a command as in `!d` for event 12 or `!wri` for event 9, |
| !?*string*? | a substring of the command string as in `!?mic?` for event 9. |
| !! | refers to the previous command. |

By themselves, these forms simply reintroduce the words of the specified events, each separated by a single blank.

To select specific words from an event, follow the event specification with a colon (:) and a designator for the desired words. The words of a input line are numbered from 0, the first word (usually a command) is 0, the second word (usually the first argument) is 1, and so forth. The basic word designators are:

| | |
|---|---|
| *n* | specifies the *n*'th word from the left, numbered from 0. |
| ^ | specifies the second word (usually the first argument) . |
| $ | specifies the last word. |
| % | specifies the word matched by the immediately preceding substring *(?string?)* search. |
| *x–y* | specifies a range of words, *x* to *y*. You can use designators in a range. |
| *–y* | is equivalent to 0–*y*. |
| * | is equivalent to ^–$, or nothing if there was only one word in the event. |
| *x** | is equivalent to *x*–$. |
| *x–* | is similar to *x** but omits the last word ($) . |

The colon separating the event specification from the word designator can be omitted if the argument selector begins with a circumflex, dollar sign, asterisk, hyphen, or per cent sign. After the optional word designator, you can place a sequence of modifiers, each preceded by a colon. The following modifiers are defined:

| | |
|---|---|
| h | removes a trailing pathname component, leaving the head. |
| r | removes a trailing extension (.xxx) component, leaving the root name. |
| e | removes all but the extension (.xxx) component. |
| s/*old*/*new*/ | substitutes *new* for *old*. |
| t | removes all leading pathname components, leaving the tail. |
| & | repeats the previous substitution. |
| g | applies the change globally, prefixing the above, as in g&. |
| p | prints the new command but does not execute it. |
| q | quotes the substituted words, preventing further substitutions. |
| x | is similar to q, but breaks into words at blanks, tabs and newlines. |

## 7.2.1 Notes on History Substitution

- Unless preceded by a g the modification is applied only to the first modifiable word. With substitions, it is an error for no word to be applicable.
- The *old* string in a substitution is a not regular expression in the sense of the editors, but rather a string.
- Any character can be used as a substitution delimiter in place of slash; a backslash quotes the delimiter into the *old* and *new* strings.
- An ampersand in the *new* string is replaced by the *old* string. A backslash quotes ampersand also.

- A null *old* uses the previous string either from an *old* or from a contextual scan string as in `!?`*string*`?`.
- The trailing delimiter in the substitution can be omitted if a newline follows immediately as can the trailing question mark in a contextual scan.
- A history reference can be given without an event specification, as in `!$`. In this case the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus `!?foo?^ !$` gives the first and last arguments from the command matching `?foo?`.
- A circumflex (^) as the first non-blank character of an input line is a special abbreviation of `!:s^`. This provides a convenient shorthand for substitutions on the text of the previous line. For example, `^lb^lib` equivalent to `!:s^lb^lib`.
- A history substitution can be surrounded with braces if necessary to insulate it from the characters that follow. Thus, after `ls -ld ~paul` you can type `!{l}a` to get `ls -ld ~paula`, while `!la` would look for a command starting with `la`.

## 7.3 QUOTATION

Enclosing a string in single quotes and prevents all substitutions except history substitutions. Enclosing a string in double quotes prevents alias subsititutions and filename substitutions.

In both cases the resulting text becomes (all or part of) a single word. Only in one special case (see Section 7.6) does a double quoted string yield parts of more than one word; single quoted strings never do.

The sharp sign (#) introduces a comment except when preceded by backslash and in quotations using single quote, open single quote, and double quote.

## 7.4 ALIAS SUBSTITUTION

After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, the text that is the alias for that command is reread (with the history mechanism available) as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, the argument list is left unchanged.

For example, if the alias for `ls` is `ls -l`, the command `ls /usr` maps to `ls -l /usr`, the argument list being undisturbed. Similarly, if the alias for `lookup` is `grep !^ /etc/passwd`, the command `lookup bill` maps to `grep bill /etc/passwd`.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error. To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

Aliases can introduce parser metasyntax. For example: `alias print 'pr \!* | lpr'` makes a command that *pr's* its arguments to the line printer.

## 7.5 VARIABLE SUBSTITUTION

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed, keyed by dollar sign characters. Substitution can be prevented by preceding the dollar sign with a backslash except within double quotes, where it always occurs, and within single quotes where it never occurs. Strings enclosed in open single quotes are interpreted later (see Section 7.6); thus dollar sign substitution does not occur there until later, if at all. A dollar sign is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable substitution, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in double quotes or given the :q modifier, the results of variable substitution can eventually be command and filename substituted. Within double quotes, a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the :q modifier is applied to a substitution, the variable expands to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable that is not set.

`$name`
`${name}`
: are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters that would otherwise be part of it. If *name* is not a shell variable, but is set in the environment, that value is returned (but colon modifiers and the other forms given below are not available in this case).

`$name[selector]`
`${name[selector]}`
: can be used to select only some of the words from the value of *name*. The selector is subjected to dollar sign substitution and can consist of a single number or two numbers separated by a hyphen. The first word of a variables value is numbered `1`. If the first number of a range is omitted it defaults to `1`. If the last member of a range is omitted it defaults to `$#`*name*. The selector asterisk selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

`$#name`
`${#name}`
: gives the number of words in the variable. This is useful for later use in a selector.

`$0`
: substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

| | |
|---|---|
| $number<br>${number} | is equivalent to $argv[number]. |
| $* | is equivalent to $argv[*]. |

The following substitutions can not be modified:

| | |
|---|---|
| $?name<br>${?name} | substitutes the string `1` if name is set, `0` if it is not. |
| $?0 | substitutes `1` if the current input filename is known, `0` if it is not. |
| $$ | substitute the (decimal) process number of the (parent) shell. |
| $< | substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script. |

### 7.5.1 Variable Substitution Modifiers

The following modifiers can be applied to the substitutions above except where noted. If braces appear in the command form, the modifiers must appear within the braces. Only one colon modifier is allowed on each variable substitution.

| | |
|---|---|
| h | removes a trailing pathname component, leaving the head. |
| r | removes a trailing extension (.xxx) component, leaving the root name. |
| e | removes all but the extension (.xxx) component. |
| t | removes all leading pathname components, leaving the tail. |
| g | applies the change globally, prefixing another modifier. |
| q | quotes the substituted words, preventing further substitutions. |
| x | is similar to q, but breaks into words at blanks, tabs and newlines. |

## 7.6 COMMAND SUBSTITUTION

Command substitution is indicated by a command enclosed in open single quotes (‘). The command is executed and the output replaces the original string.

Normally, the command output is broken into separate words at blanks, tabs and newlines, with null words being discarded. Within double quotes, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. It is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line. Command substitutions can substitute no more characters than are allowed in an argument list.

## 7.7 FILENAME SUBSTITUTION

If a word contains asterisk, question mark, left bracket, or left brace, or begins with tilde, that word is a candidate for filename substitution, also known as "globbing". This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names that match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters asterisk, question mark, and left bracket imply pattern matching, tilde and left brace being more akin to abbreviations.

In matching filenames, a period at the beginning of a filename or immediately following a slash, as well as slash, must be matched explicitly. Asterisk matches any string of characters, including the null string. Question mark matches any single character. Characters enclosed in brackets match any one of themselves. Within brackets, a pair of characters separated by a hyphen matches any character lexically between the two.

A tilde at the beginning of a filename is used to refer to home directories. Standing alone, it expands to the invoker's home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits andp hyphens, the shell searches for a user with that name and substitutes their home directory; thus `~ken` might expand to `/usr/ken` and `~ken/chmach` to `/usr/ken/chmach`. If the tilde is followed by a character other than a letter or slash or appears not at the beginning of a word, it is left undisturbed.

The metanotation `a{b,c,d}e` is a shorthand for `abe ace ade`. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct can be nested. Thus `~source/s1/{oldls,ls}.c` expands to `/usr/source/s1/oldls.c /usr/source/s1/ls.c` whether or not these files exist without any chance of error if the home directory for `source` is `/usr/source`. Similarly `../{memo,*box}` might expand to `../memo ../box ../mbox`. (Note that `memo` was not sorted with the results of matching `*box`.) As a special case `{`, `}` and `{}` are passed undisturbed.

## 7.8 JOB CONTROL

> *Note: To use its job control facilities, users of the C shell must automatically use the new tty driver fully described in tty* (System Reference Manual: Special Files). *The new tty driver allows generation of interrupt characters from the keyboard to tell jobs to stop. See stty for details on setting options in the new tty driver.*

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with ampersand, the shell prints a line that looks like:

```
[1] 1234
```

indicating that the job that was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you can press CTRL Z which sends a STOP signal to the current job. The shell will then normally indicate that the job has been "Stopped", and print another prompt.

You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A CTRL Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

> *Note: When a command is restarted from a stop, the shell prints the directory it started in if different from the current directory; this can be misleading (i.e. wrong) as the job may have changed directories internally.*

There is another special key, CTRL Y, that does not generate a STOP signal until a program attempts to *read* it *(System Reference Manual: System Calls)*. This can usefully be typed ahead when you have prepared some commands for a job that you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command `stty tostop`. If you set this tty option, background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The per cent sign (%) introduces a job name. If you wish to refer to job number 1, you can name it as `%1`. Just naming a job brings it to the foreground; thus `%1` is a synonym for `fg %1`, bringing job 1 back into the foreground. Similarly, `%1 &` resumes job 1 in the background.

Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus `%ex` would normally restart a suspended *ex* job, if there were only one suspended job whose name began with the string `ex`. It is also possible to say `%?string`, which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a plust sign (+) and the previous job with a hpyhen (-). The abbreviation `%+` refers to the current job and `%-` refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), `%%` is also a synonym for the current job.

### 7.8.1 Signal Handling

The shell normally ignores *quit* signals. Jobs running detached (either by & or the *bg* or %... & commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values that the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file `.logout`.

## 7.9 BUILT-IN COMMANDS

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

> *Note: Shell built-in commands are not stoppable or restartable. Command sequences of the form* `a ; b ; c` *are also not handled gracefully when stopping is attempted. If you suspend* `b`*, the shell immediately executes* `c`*. This is especially noticeable if this expansion results from an alias. It suffices to place the sequence of commands in parentheses to force it to a subshell, i.e.* `( a ; b ; c )`.

### 7.9.1 Flow of Control

The shell contains a number of commands that can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if-then-else* form of the *if* statement require that the major keywords appear in a single simple command on an input line as shown below.

> *Note: Commands within loops, prompted for by question mark, are not placed in the history list.*

If the shell's input is not seekable, the shell buffers input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

### 7.9.2 Expressions

A number of the built-in commands take expressions in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, *exit*, *if*, and *while* commands. The following operators operate on numbers except where indicated. They are shown in descending order of precedence:

| | |
|---|---|
| ( ) | grouping |
| ! | negation |
| % | remainder |
| / | division |
| * | multiplication |
| - | subtraction |
| + | addition |
| >> | shift right |
| << | shift left |
| > | greater than |
| < | less than |
| >= | greater than or equal to |

| | |
|---|---|
| <= | less than or equal to |
| !~ | not equal to (pattern string) |
| =~ | equal to (pattern string) |
| != | not equal to (string) |
| == | equal to (string) |
| & | bitwise and |
| ^ | bitwise exclusive or |
| \| | bitwise inclusive or |
| && | logical and |
| \|\| | logical or |

The following groups have the same level of precedence: (== != =~ !~), (<= >= < >), (<< >>), (+ -), (* / %).

The operators =~ and !~ are like != and == except that the right hand side is a *pattern* (containing filename substitution characters) against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings that begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The result of all expressions are strings that represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions that are syntactically significant to the parser (ampersand, vertical bar, angle brackets, and parentheses) they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in braces and file enquiries of the form *-l name* where *l* is one of:

| | |
|---|---|
| r | read access |
| w | write access |
| x | execute access |
| e | existence |
| o | ownership |
| z | zero size |
| f | plain file |
| d | directory |

The specified *name* is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, all enquiries return false, i.e. 0. Command executions succeed, returning true, i.e. 1, if the command exits with status 0, otherwise they fail, returning false, i.e. 0. If more detailed status information is required, the command should be executed outside of an expression and the variable *status* examined.

## 7.9.3 Summary of Built-in Commands

%job

Brings the specified job into the foreground.

%job &

Continues the specified job in the background.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains <, >, & or |, at least this part of the expression must be placed within parentheses. The third form assigns the value of *expr* to the *index'th* argument of *name*. Both *name* and its *index'th* component must already exist.

The operators *=, +=, and so forth, are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix ++ and -- operators increment and decrement *name* respectively, i.e. @ i++.

**alias**
**alias** name
**alias** name wordlist

The first form prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command and filename substituted. *Name* is not allowed to be *alias* or *unalias*.

**alloc**

Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

**bg**
**bg** %job ...

Puts the current or specified jobs into the background, continuing them if they were stopped.

**break**

Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

**breaksw**

Causes a break from a *switch*, resuming after the *endsw*.

**case** label:

A label in a *switch* statement as discussed below.

**cd**
**cd** name
**chdir**
**chdir** name

Change the shells working directory to directory *name*. If no argument is given, change to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with /, ./ or ../), each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with /, this is tried to see if it is a directory.

> *Note: Symbolic links fool the shell. In particular,* **dirs** *and* **cd** name *do not work properly once you've crossed through a symbolic link.*

**continue**

Continue execution of the nearest enclosing *while* or *foreach*. The rest of the commands on the current line are executed.

**default:**

Labels the default case in a *switch* statement. The default should come after all *case* labels.

**dirs**

Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

> *Note: Symbolic links fool the shell. In particular,* **dirs** *and* **cd** name *do not work properly once you've crossed through a symbolic link.*

**echo** wordlist
**echo** **–n** wordlist

The specified words are written to the shells standard output, separated by spaces, and terminated with a newline unless the –n option is specified.

**else**
**end**
**endif**
**endsw**

See the descriptions of the *foreach*, *if*, *switch*, and *while* statements below.

**eval** arg ...

(As in *sh*.) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands

generated as the result of command or variable substitution, since parsing occurs before these substitutions.

**exec** command

The specified command is executed in place of the current shell.

**exit**
**exit(expr)**

The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

**fg**
**fg** %job ...

Brings the current or specified jobs into the foreground, continuing them if they were stopped.

**foreach** name (wordlist)
...
**end**

The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end* must appear alone on separate lines.)

The built-in command *continue* can be used to continue the loop prematurely and the built-in command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with a question mark before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

**glob** wordlist

Like *echo* but no backslash escapes are recognized and words are delimited by null characters in the output. Useful for programs that wish to use the shell to filename expand a list of words.

**goto** word

The specified *word* is filename and command expanded to yield a string of the form `label`. The shell rewinds its input as much as possible and searches for a line of the form `label:` possibly preceded by blanks or tabs. Execution continues after the specified line.

**hashstat**

Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding *exec*'s). An *exec* is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component that does not begin with a slash.

**history**
**history** *n*
**history -r** *n*
**history -h** *n*

Displays the history event list; if *n* is given only the *n* most recent events are printed. The -r option reverses the order of printout to be most recent first rather than oldest first. The -h option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourceing using the -h option to *source*.

**if** (expr) command

If the specified expression evaluates true, the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is not executed (this is a bug).

**if** (expr) **then**
...
**else if** (expr2) **then**
...
**else**
...
**endif**

If the specified *expr* is true, the commands to the first *else* are executed; else if *expr2* is true, the commands to the second else are executed, etc. Any number of *else-if* pairs are possible; only one *endif* is needed. The *else* part is likewise optional. (The words *else* and *endif* must appear at the beginning of input lines; the *if* must appear alone on its input line or after an *else*.)

**jobs**
**jobs -l**

Lists the active jobs; given the -l options lists process id's in addition to the normal information.

**kill** %job
**kill** -sig %job ...
**kill** pid
**kill** -sig pid ...
**kill -l**

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in `/usr/include/signal.h`, stripped of the prefix `SIG`). The signal names are listed by `kill -l`. There is no default, saying just `kill` does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), the job or process will be sent a CONT (continue) signal as well.

**limit**
**limit** *resource*
**limit** *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, the current limit is printed; if no *resource* is given, all limitations are given.

Resources controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file that can be created), *datasize* (the maximum growth of the data+stack region via *sbrk(System Calls)* beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), and *coredumpsize* (the size of the largest core dump that will be created).

The *maximum-use* can be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is `k` or `kilobytes` (1024 bytes); a scale factor of `m` or `megabytes` can also be used. For *cputime* the default scaling is `seconds`, while `m` for minutes or `h` for hours, or a time of the form `mm:ss` giving minutes and seconds can be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

**login**

Terminates a login shell, replacing it with an instance of /bin/login. This is one way to log off, included for compatibility with *sh*.

**logout**

Terminates a login shell. Especially useful if *ignoreeof* is set.

**nice**
**nice** +number
**nice** command
**nice** +number command

The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user can specify negative niceness by using `nice` –number `...`. Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

**nohup**
**nohup** command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with ampersand are effectively *nohup'ed*.

**notify**
**notify** %job ...

Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

**onintr**
**onintr** –
**onintr** label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. The second form `onintr –` causes all interrupts to be ignored. The final form causes the shell to execute a `goto label` when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

**popd**
**popd** +n

Pops the directory stack, returning to the new top directory. With a argument +n discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

**pushd**
**pushd** name
**pushd** +n

With no arguments, *pushd* exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* changes to the new directory (ala *cd*) and pushes the old current working directory (as in *csw*) onto the directory stack. With a numeric argument, rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

**rehash**

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

**repeat** count command

The specified *command* that is subject to the same restrictions as the *command* in the one line *if* statement above, is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

**set**
**set** name
**set** name=word
**set** name[index]=word
**set** name=(wordlist)

The first form of the command shows the value of all shell variables. Variables that have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index'th* component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments can be repeated to set multiple values in a single set command. Note however, that variable substitution happens for all arguments before any setting occurs.

**setenv** name value

Sets the value of environment variable *name* to be *value*, a single string. The most commonly used environment variable USER, TERM, and PATH are automatically imported to and exported from the *csh* variables *user*, *term*, and *path*; there is no need to use *setenv* for these.

**shift**
**shift** variable

The members of *argv* are shifted to the left, discarding *argv*[*1*]. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

**source** name
**source** –h name

The shell reads commands from *name*. *Source* commands can be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Normally input during *source* commands is not placed on the history list; the –h option causes the commands to be placed in the history list without being executed.

**stop**
**stop** %job ...

Stops the current or specified job that is executing in the background.

**suspend**

Causes the shell to stop in its tracks, much as if it had been sent a stop signal with ^Z. This is most often used to stop shells started by *su*.

```
switch (string)
case str1:
  ...
  breaksw
...
default:
  ...
  breaksw
endsw
```

Each case label is successively matched, against the specified *string*, which is first command and filename expanded. The file metacharacters `*`, `?` and `[...]` can be used in the case labels, which are variable expanded. If none of the labels match before a "default" label is found, the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise control can fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

**time**
**time** command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the *time* variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

**umask**
**umask** value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

**unalias** pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by `unalias *`. It is not an error for nothing to be *unaliased*.

**unhash**

Use of the internal hash table to speed location of executed programs is disabled.

**unlimit** *resource*
**unlimit**

Removes the limitation on *resource*. If no *resource* is specified, all *resource* limitations are removed.

**unset** pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by `unset *`; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

**unsetenv** pattern

Removes all variables whose name match the specified pattern from the environment. See also the *setenv* command above and *printenv*.

**wait**

All background jobs are waited for. It the shell is interactive, an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

**while** (expr)
...
**end**

While the specified expression evaluates non-zero, the commands between the *while* and the matching end are evaluated. *Break* and *continue* can be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

## 7.10 VARIABLES

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of its value are referred to in special ways.

Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter.

The values of variables can be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle that causes command input to be echoed. The setting of *verbose* results from the -v command line option.

Other operations treat variables numerically. The @ command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

## 7.10.1 Special Variables

The following variables have special meaning to the shell. Of these, *argv*, *cwd*, *home*, *path*, *prompt*, *shell* and *status* are always set by the shell. Except for *cwd* and *status*, setting occurs only at initialization; variables will not then be modified unless done explicitly by the user.

The shell copies the environment variable USER into the variable *user*, TERM into *term*, and HOME into *home*, and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is handled the same way; it is not necessary to worry about its setting other than in the file `.cshrc` as inferior *csh* processes will import the definition of *path* from the environment, and re-export it if you then change it.

**argv** set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. `$1` is replaced by `$argv[1]`, etc.

**cdpath** gives a list of alternate directories searched to find subdirectories in *chdir* commands.

**cwd** the full pathname of the current directory.

**echo** set when the –x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-built-in commands all substitutions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively.

**histchars** can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing exclamation point. The second character of its value replaces vertical bar in quick substitutions.

**history** can be given a numeric value to control the size of the history list. Any command that has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

**home** the home directory of the invoker, initialized from the environment. The filename substitution of tilde refers to this variable.

**ignoreeof** if set, the shell ignores end-of-file from input devices that are terminals. This prevents shells from accidentally being killed by CTRL D.

**mail** the files where the shell checks for mail. This is done after each command completion that will result in a prompt, if a specified interval has elapsed. The shell prints `You have new mail.` If the file exists with an access time not greater than its modify time.

If the first word of the value of *mail* is numeric it specifies a different mail checking interval, in seconds, than the default, which is ten minutes.

If multiple mail files are specified, the shell prints `New mail in` *name* when there is mail in the file *name*.

**noclobber** As described in Section 7.1.5, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that >> redirections refer to existing files.

**noglob** if set, filename substitution is inhibited. This is most useful in shell scripts that are not dealing with filenames, or after a list of filenames has been obtained and further substitutions are not desirable.

**nonomatch** if set, it is not an error for a filename substitution to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. `echo [` still gives an error.

**notify** if set, the shell notifies asynchronously of job completions. The default is to rather present job completions just before printing a prompt.

**path** each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable, only full path names will execute. The usual search path is the current directory, `/bin` and `/usr/bin`, but this can vary from system to system. For the super-user the default search path is `/etc`, `/bin` and `/usr/bin`. A shell that is given neither the –c nor the –t option will normally hash the contents of the directories in the *path* variable after reading `.cshrc`, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.

**prompt** the string that is printed before each command is read from an interactive terminal input. If an exclamatin point appears in the string it will be replaced by the current event number unless a preceding backslash is given. The default is % for normal users and # for the super-user.

**savehist** is given a numeric value to control the number of entries of the history list that are saved in `~/.history` when the user logs out. Any command that has been referenced in this many events will be saved. During start up the shell sources `~/.history` into the history list enabling history to be saved across logins. Too large values of *savehist* will slow down the shell during start up.

**shell** the file in which the shell resides. This is used in forking shells to interpret files that have execute bits set, but that are not executable by the system. (See Section 7.1.4.) Initialized to the (system-dependent) home of the shell.

**status** the status returned by the last command. If it terminated abnormally, 0200 is added to the status. Built-in commands that fail return exit status `1`, all other built-in commands set status `0`.

**time** controls automatic timing of commands. If set, then any command that takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage that is the ratio of user plus system times to real time to be printed when it terminates.

**verbose** set by the –v command line option, causes the words of each command to be printed after history substitution.

# CHAPTER 8
# USING THE BOURNE SHELL

## 8.1 INTRODUCTION

A shell is both a command language and a programming language that provides an interface to the operating system. This chapter describes, with examples, the Bourne shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with Concentrix is an advantage when reading this section. Section 8.2 describes those features of the shell primarily intended for use within shell procedures. These include the control–flow primitives and string–valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell.

### 8.1.1 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* to print status information, size and the creation date for each file.

### 8.1.2 Background Commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command can be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing & is an operator that 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 can be obtained using the *ps* command.

### 8.1.3 Input/Output Redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output can be sent to a file by writing, for example,

```
ls -l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist, the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls*. Output can be appended to a file using the notation

```
ls -l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command can be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* 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 this could be used:

```
wc -l <file
```

### 8.1.4 Pipelines and Filters

The standard output of one command can be connected to the standard input of another by writing the 'pipe' operator, indicated by |, as in,

```
ls -l | wc
```

Two 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 system call and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* 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*, 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*. For example,

```
who | sort
```

prints an alphabetically sorted list of logged in users.

A pipeline can consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

### 8.1.5 File Name Generation

Many commands accept arguments that are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names 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*.

```
/usr/fred/test/?
```

matches all names in the directory /usr/fred/test that consist of a single character. If no file name is found that matches the pattern, 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 can 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. (*echo* is a 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 file name must be explicitly matched.

```
echo *
```

therefore echoes all file names in the current directory not beginning with '.'.

```
echo .*
```

echoes all those file names that begin with '.'. This avoids inadvertent matching of the names '.' and '..' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '.' and '..'.)

## 8.1.6 Quoting

Characters that have a special meaning to the shell, such as < > * ? | &, are called metacharacters. A complete list of metacharacters is given in Section 8.5. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

echoes a single ?, and

```
echo \\
```

echoes a single \. To allow long strings to be continued over more than one line the sequence \newline is ignored.

Backslash is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters can be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

echoes

```
xx****xx
```

The quoted string cannot contain a single quote but can contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to Section 8.3.4.

## 8.1.7 Prompting

When the shell is used from a terminal, it issues a prompt before reading a command. By default this prompt is '$ '. It can be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. 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, an interrupt (CTRL C) returns the shell to read another command. This prompt can be changed by saying, for example,

```
PS2=more
```

### 8.1.8 The Shell and Login

Following *login*, the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file .profile, it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

### 8.1.9 Summary

- ls
  Print the names of files in the current directory.
- ls >file
  Put the output from *ls* into *file*.
- ls | wc –l
  Print the number of files in the current directory.
- ls | grep old
  Print those file names containing the string *old*.
- ls | grep old | wc –l
  Print the number of files whose name contains the string *old*.
- cc pgm.c &
  Run *cc* in the background.

## 8.2 SHELL PROCEDURES

The shell can be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments can 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

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

Files have three independent attributes, *read*, *write* and *execute*. The command *chmod* can be used to make a file executable. For example,

```
chmod +x wg
```

ensures that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run 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 -ms $*
```

which simply prepends some arguments to those already given.

## 8.2.1 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* that searches the file /usr/lib/telnos that 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*.

```
telfredbert
```

prints those lines containing *fred* followed by those for *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 semicolon. Furthermore, reserved words like do and done are only recognized following a newline or semicolon. *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, 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 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* can 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.

## 8.2.2 Control Flow – Case

A multiple way branch is provided for by the case notation. For example,

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

is an *append* command. When called with one argument as

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2, 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.

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

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

Another example of the use of the case construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* 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 ?.

## 8.2.3 Including Data in Shell Procedures

The shell procedure *tel* in Section 8.2.1 uses the file /usr/lib/telnos to supply the data for *grep*. An alternative is to include this data within the shell procedure 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*. The string ! is arbitrary, the data being terminated by a line that consists of the string following <<.

Parameters are substituted in the data before it is made available to *grep* 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 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* prints a ? if there are no occurrences of the string $1.) Substitution within in-line data can be prevented entirely by quoting the terminating string, for example,

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

The in-line data is presented without modification to *grep*. If parameter substitution is not required in in-line data, the latter form is more efficient.

## 8.2.4 Shell Variables

The shell provides string–valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables can be given values by writing, for example,

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

which assigns values to the variables user, box and acct. A variable can be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with $; for example,

```
echo $user
```

echoes *fred*.

Variables can be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

moves the file *pgm* 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* to the file /tmp/psa, whereas,

```
ps a >$tmpa
```

would cause the value of the variable tmpa to be substituted.

Except for $? the following are set initially by the shell. $? is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non–zero exit status is returned. 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 the *append* command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. 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* command. The current directory is used to resolve file name 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.

```
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 the user's login profile.

$PATH A list of directories that contain 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 (see Section 8.3.4)*.

## 8.2.5 The Test Command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test -f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see the command *test* 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
```

## 8.2.6 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 whose actions 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, *command-list2* is executed; otherwise, the loop terminates. For example,

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

is equivalent to

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

*shift* is a shell command that renames the positional parameters $2, $3, ... as $1, $2, ... and loses $1.

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

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

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

## 8.2.7 Control Flow – If

Also available is a general conditional branch of the form,

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

that tests the value returned by the last simple command following if.

The if command can 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 Section 8.2.10.

A multiple test if command of the form

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

can be written using an extension of the if notation as,

```
if ...
then      ...
elif           ...
then      ...
elif           ...
...
fi
```

The following example is the *touch* command that changes the 'last modified' time for a list of files. The command can be used in conjunction with the program *make* to force recompilation of a list of files.

```
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 subsequent files to be created if they do not already exist. Otherwise, if the file does 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 thus causing the last modified date to be updated.

The sequence

```
if command1
then      command2
fi
```

can 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.

## 8.2.8 Command Grouping

Commands can be grouped in two ways,

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

and

```
( command-list )
```

In the first *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.

### 8.2.9 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 to help isolate syntax errors. It can be invoked without modifying the procedure by saying

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

where *proc* is the name of the shell procedure. This flag can be used in conjunction with the –n flag that prevents execution of subsequent commands. (Note that saying *set –n* at a terminal renders the terminal useless until an end–of–file is typed.) The command

```
set -x
```

produces an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags can be turned off by saying

```
set -
```

and the current setting of the shell flags is available as $–.

## 8.3 KEYWORD PARAMETERS

Shell variables can 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 can also be used to set positional parameters from within a procedure. For example,

```
set - *
```

sets $1 to the first file name in the current directory, $2 to the next, and so on. Note that the first argument, –, ensures correct treatment when the first file name begins with a –.

### 8.3.1 Parameter Transmission

When a shell procedure is invoked both positional and keyword parameters can 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 cannot modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant can be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are invalid.

### 8.3.2 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 can 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 can be assigned a default value using the notation

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

which substitutes the same string as

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

and if d were not previously set, it is set to the string '.'. (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 that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables user, acct or bin are not set, the shell abandons execution of the procedure.

## 8.3.3 Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* 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 taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \. For example,

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

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including in-line data) 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,* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

prints the string *main*. Its use is illustrated by the following fragment from a *cc* command.

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

that 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 in time order, most recent first.

```
set `date`; echo $6 $2 $3, $4
```

prints, for example, "1977 Nov 1, 23:59:59".

## 8.3.4 Evaluation and Quoting

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

Commands are parsed initially according to the grammar given in Section 8.4. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. $user
- 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.

- blank interpretation

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

  ```
  echo ''
  ```

  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 set to the null string.

- file name generation

  Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of file names is generated to replace the word. Each such file name 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.

As well as the quoting mechanisms described earlier using \ and '...' a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and can 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.

```
echo "$@"
```

passes the positional parameters, unevaluated, to *echo* and is equivalent to

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

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

| | *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 |

**Figure 8–1: Quoting Mechanisms**

In cases where more than one evaluation of a string is required the built–in command *eval* can 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 since there is no interpretation of metacharacters, such as |, following substitution.

## 8.3.5 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 the system call *gtty*)). A shell invoked with the –i flag is also interactive.

Execution of a command (see also 8.3.7) may fail for any of the following reasons.

- Input output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 8–2 below for a complete list of signals.
- The command terminates normally but returns a non–zero 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.

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

The shell flag –e causes the shell to terminate if any error is detected.

| | |
|---|---|
| 1 | hangup |
| 2 | interrupt |
| 3* | quit |
| 4* | invalid instruction |
| 5* | trace trap |
| 6* | IOT instruction |
| 7* | EMT instruction |
| 8* | floating point exception |
| 9 | kill (cannot be caught or ignored) |
| 10* | bus error |
| 11* | segmentation violation |
| 12* | bad argument to system call |
| 13 | write on a pipe with no one to read it |
| 14 | alarm clock |
| 15 | software termination (from the command *kill*) |

**Figure 8–2: Signals**

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit 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.

## 8.3.6 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), and if this signal is received, executes the commands

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

*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.

Signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without 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 (see 8.3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 8–3). The cleanup action is to remove the file junk$$.

```
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
```

**Figure 8–3: The Touch Command**

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, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure can, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

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

which causes *hangup*, *interrupt*, *quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps can be reset by saying

```
trap 2 3
```

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

```
trap
```

The procedure *scan* (Figure 8–4) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

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

**Figure 8–4: The Scan Command**

*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 non–zero exit status if either an end–of–file is read or an interrupt is received.

## 8.3.7 Command Execution

To run a command (other than a built–in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built–in command *exec* is used in the rare cases when no fork is required and simply replaces the 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 file name 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.

> *word* The standard output (file descriptor 1) is sent to the file *word* that is created if it does not already exist.

>> *word* The standard output is sent to file *word*. If the file exists, 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 not including a line consisting only of *word*. If *word* is quoted, no interpretation of the data occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to quote the characters \ $ ' and the first character of *word*. In the latter case \newline is ignored (c.f. quoted strings).

>& *digit* The file descriptor *digit* is duplicated using the system call *dup* 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 can be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

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

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

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

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

The environment for a command run in the background such as

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

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

```
ed file &
```

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

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

## 8.3.8 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, 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 *gtty*), this shell is *interactive*. In this case TERMINATE

is ignored (so that kill 0 does not kill an interactive shell) and INTERRUPT is caught and ignored (so that wait is interruptable). In all cases QUIT is ignored by the shell.

## 8.4 GRAMMAR

*item*: *word*
*input–output*
*name = value*

*simple–command*: *item*
*simple–command item*

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

*pipeline*: *command*
*pipeline* | *command*

*andor*: *pipeline*
*andor* && *pipeline*
*andor* || *pipeline*

*command–list*: *andor*
*command–list* ;
*command–list* &
*command–list* ; *andor*
*command–list* & *andor*

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

*file*: *word*
& *digit*
& –

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

*pattern*: *word*
*pattern* | *word*

*else–part*: **elif** *command–list* **then** *command–list else–part*
**else** *command–list*
*empty*

*empty*:

*word*: a sequence of non–blank characters

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

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

## 8.5 METACHARACTERS AND RESERVED WORDS

a) syntactic

| | |
|---|---|
| \| | pipe symbol |
| && | 'andf' symbol |
| \|\| | 'orf' symbol |
| ; | command separator |
| ;; | case delimiter |
| & | background commands |
| ( ) | command grouping |
| < | input redirection |
| << | input from in-line data |
| > | 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 ' |
| "..." | quote the enclosed characters except for $ ` \ " |

e) reserved words

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

# CHAPTER 9
# BOURNE SHELL REFERENCE

The Bourne Shell is a command programming language that executes commands read from a terminal or a file. This chapter provides reference information about the Bourne Shell.

## 9.1 COMMANDS

A *simple-command* is a sequence of non blank *words* separated by blanks (a blank is a tab or a space). The first word specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see the system call *execve*). The *value* of a simple-command is its exit status if it terminates normally or 200+*status* if it terminates abnormally (see the system call *sigvec* for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by |. The standard output of each command but the last is connected by a *pipe* to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A *list* is a sequence of one or more *pipelines* separated by ;, &, && or || and optionally terminated by ; or &. ; and & have equal precedence, which is lower than that of && and ||, && and || also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding *pipeline* to be executed without waiting for it to finish. The symbol && (||) causes the *list* following to be executed only if the preceding *pipeline* returns a zero (non zero) value. Newlines can appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

**for** *name* [**in** *word* ...] **do** *list* **done**
Each time a for command is executed, *name* is set to the next word in the for word list. If in *word* ... is omitted, in "$@" is assumed. Execution ends when there are no more words in the list.

**case** *word* **in** [*pattern* [ | *pattern* ] ... ) *list* ;;] ... **esac**
A case command executes the *list* associated with the first pattern that matches *word*. The form of the patterns is the same as that used for file name generation.

**if** *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following if is executed and, if it returns zero, the *list* following then is executed. Otherwise, the *list* following elif is executed and, if its value is zero, the *list* following then is executed. Failing that the else *list* is executed.

**while** *list* [**do** *list*] **done**
A while command repeatedly executes the while *list* and if its value is zero executes the do *list*; otherwise the loop terminates. The value returned by a while command is that of the last executed command in the do *list*. until can be used in place of while to negate the loop termination test.

( *list* ) Execute *list* in a subshell.

{ *list* } *list* is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

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

## 9.2 COMMAND SUBSTITUTION

The standard output from a command enclosed in a pair of back quotes (``) can be used as part or all of a word; trailing newlines are removed.

## 9.3 PARAMETER SUBSTITUTION

The character $ is used to introduce substitutable parameters. Positional parameters can be assigned values by set. Variables can be set by writing

*name=value* [ *name=value* ] ...

${*parameter*}
A *parameter* is a sequence of letters, digits or underscores (a *name*), a digit, or any of the characters * @ # ? - $ !. The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter is a digit, it is a positional parameter. If parameter* is * or @, all the positional parameters, starting with $1, are substituted separated by spaces. $0 is set from argument zero when the shell is invoked.

${*parameter-word*}
If *parameter* is set, substitute its value; otherwise substitute *word*.

${*parameter=word*}
If *parameter* is not set, set it to *word*; the value of the parameter is then substituted. Positional parameters can not be assigned to in this way.

${*parameter?word*}
If *parameter* is set, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, a standard message is printed.

${*parameter+word*}
If *parameter* is set, substitute *word*; otherwise substitute nothing.

In the above *word* is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d-'pwd'} only executes *pwd* if *d* is unset.)

The following *parameters* are automatically set by the shell.

| | |
|---|---|
| # | The number of positional parameters in decimal. |
| - | Options supplied to the shell on invocation or by set. |
| ? | The value returned by the last executed command in decimal. |
| $ | The process number of this shell. |
| ! | The process number of the last background command invoked. |

The following *parameters* are used but not set by the shell.

| | |
|---|---|
| **HOME** | The default argument (home directory) for the cd command. |
| **PATH** | The search path for commands (see execution). MAIL If this variable is set to the name of a mail file, the shell informs the user of the arrival of mail in the specified file. |
| **PS1** | Primary prompt string, by default '$ '. |
| **PS2** | Secondary prompt string, by default '> '. |
| **IFS** | Internal field separators, normally space, tab, and newline. |

## 9.4 BLANK INTERPRETATION

After parameter and command substitution, any results of substitution are scanned for internal field separator characters (those found in $IFS) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

## 9.5 FILE NAME GENERATION

Following substitution, each command word is scanned for the characters *, ? and [. If one of these characters appears, the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found

that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following a /, and the character /, must be matched explicitly.

* Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the characters enclosed. A pair of characters separated by – matches any character lexically between the pair.

## 9.6 QUOTING

The following characters have a special meaning to the shell and cause termination of a word unless quoted.

**; & ( ) | < > newline space tab**

A character can be *quoted* by preceding it with a \. \newline is ignored. All characters enclosed between a pair of quote marks (''), except a single quote, are quoted. Inside double quotes ("") parameter and command substitution occurs and \ quotes the characters \ ' " and $.

"**$***" is equivalent to "**$1 $2 ...**" whereas
"$@" is equivalent to "**$1**" "**$2**" ...

## 9.7 PROMPTING

When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt ($PS2) is issued.

## 9.8 INPUT/OUTPUT

Before a command is executed its input and output can be redirected using a special notation interpreted by the shell. The following can appear anywhere in a simplecommand or can precede or follow a *command* and are not passed on to the invoked command. Substitution occurs before *word* or *digit* is used.

*<word* Use file *word* as standard input (file descriptor 0).

*>word* Use file *word* as standard output (file descriptor 1). If the file does not exist, it is created; otherwise it is truncated to zero length.

*>>word* Use file *word* as standard output. If the file exists, output is appended (by seeking to the end); otherwise the file is created.

*<<word* The shell input is read up to a line the same as *word*, or end of file. The resulting document becomes the standard input. If any character of *word* is quoted, no interpretation is placed upon the characters of the document;

otherwise, parameter and command substitution occurs, \newline is ignored, and \ is used to quote the characters \ $ ' and the first character of *word*.

<&*digit* The standard input is duplicated from file descriptor *digit*; see the system call *dup*. Similarly for the standard output using >.

<&- The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit, the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example,

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

creates file descriptor 2 to be a duplicate of file descriptor 1.

If a command is followed by &, the default standard input for the command is the empty file (/dev/null). Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input output specifications.

## 9.9 ENVIRONMENT

The environment is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list; see the system call *execve*. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a *parameter* for each name found, giving it the corresponding value. Executed commands inherit the same environment. If the user modifies the values of these *parameters* or creates new ones, none of these affects the environment unless the export command is used to bind the shell's *parameter* to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in export commands.

The environment for any *simple-command* can be augmented by prefixing it with one or more assignments to *parameters*. Thus these two lines are equivalent

```
TERM=450 cmd args
(export TERM; TERM=450; cmd args)
```

If the -k flag is set, *all* keyword arguments are placed in the environment, even if the occur after the command name. The following prints 'a=b c' and 'c': echo a=b c set -k echo a=b c

## 9.10 SIGNALS

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the shell from its parent. (But see also trap.)

## 9.11 EXECUTION

Each time a command is executed the above substitutions are carried out. Except for the 'special commands' listed below a new process is created and an attempt is made to execute the command via the system call *execve*.

The shell parameter $PATH defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (:). The default path is :/bin:/usr/bin. If the command name contains a /, the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an *a.out* file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

## 9.12 SPECIAL COMMANDS

The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. *file* Read and execute commands from *file* and return. The search path $PATH is used to find the directory containing *file*.

**break** [*n*]
Exit from the enclosing for or while loop, if any. If *n* is specified, break *n* levels.

**continue** [*n*]
Resume the next iteration of the enclosing for or while loop. If *n* is specified, resume at the *n*-th enclosing loop.

**cd** [*arg*]
Change the current directory to *arg*. The shell parameter $HOME is the default *arg*.

**eval** [*arg ...*]
The arguments are read as input to the shell and the resulting command(s) executed.

**exec** [*arg ...*]
The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments can appear and if no other arguments are given cause the shell input output to be modified.

**exit** [*n*]
Causes a non interactive shell to exit with the exit status specified by *n*. If *n* is omitted, the exit status is that of the last command executed. (An end of file also exits from the shell.)

**export** [*name ...*]
The given names are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of exportable names is printed.

**login** [*arg ...*]
Equivalent to 'exec login arg ...'.

**read** *name ...*
One line is read from the standard input; successive words of the input are assigned to the variables *name* in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

**readonly** [*name ...*]
The given names are marked readonly and the values of the these names can not be changed by subsequent assignment. If no arguments are given, a list of all readonly names is printed.

**set** [**-eknptuvx** [*arg ...*]]

| | |
|---|---|
| **-e** | If non-interactive, exit immediately if a command fails. |
| **-k** | All keyword arguments are placed in the environment for a command, not just those that precede the command name. |
| **-n** | Read commands but do not execute them. |
| **-t** | Exit after reading and executing one command. |
| **-u** | Treat unset variables as an error when substituting. |
| **-v** | Print shell input lines as they are read. |
| **-x** | Print commands and their arguments as they are executed. |
| **-** | Turn off the -x and -v options. |

These flags can also be used upon invocation of the shell. The current set of flags can be found in $-.

Remaining arguments are positional parameters and are assigned, in order, to $1, $2, etc. If no arguments are given, the values of all names are printed.

**shift** The positional parameters from $2... are renamed $1...

**times** Print the accumulated user and system times for processes run from the shell.

**trap** [*arg*] [*n*] ...
*Arg* is a command to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If *arg* is absent, all trap(s) *n* are reset to their original values. If *arg* is the null string, this signal is ignored by the shell and by invoked commands. If *n* is 0, the command *arg* is executed on exit from the shell, otherwise upon receipt of signal *n* as numbered in the system call *sigvec*. *Trap* with no arguments prints a list of commands associated with each signal number.

**umask** [ *nnn* ]
The user file creation mask is set to the octal value *nnn* (see the system call *umask*). If *nnn* is omitted, the current value of the mask is printed.

**wait** [*n*]
Wait for the specified process and report its termination status. If *n* is not given, all currently active child processes are waited for. The return code from this command is that of the process waited for.

## 9.13 DIAGNOSTICS

Errors detected by the shell, such as syntax errors cause the shell to return a non zero exit status. If the shell is being used non interactively, execution of the shell file is

abandoned. Otherwise, the shell returns the exit status of the last command executed (see also exit).

## 9.14 NOTES

If << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document. A garbage file /tmp/sh* is created, and the shell complains about not being able to find the file by another name.

# CHAPTER 10
# USING MAIL

*Mail* is a program that allows users of computer systems to communicate by sending and receiving messages. The *Mail* program serves as a computer "Post Office." It:

- collects and delivers messages, including those traveling over networks such as the ARPANET, UUCP, and Berkeley network.
- notifies you when new messages arrive.
- provides a set of commands for manipulating messages and sending mail.
- offers simple editing capabilities to ease the composition of outgoing messages.
- provides the ability to define "mailing lists" (convenient names for groups of users).

## 10.1 SETTING UP

If you are a C shell user, you must inform the C shell of the location of your system mailbox so that *Mail* can notify you when messages arrive. Insert the following line in your `.cshrc` file:

```
set mail=(0 /usr/spool/mail/name)
```

where *name* is your own login name.

## 10.2 READING MESSAGES

When a message arrives that is addressed to you, *Mail* stores it in a file called your *system mailbox*. If you are logged in, *Mail* notifies you by printing on your terminal:

```
You have new mail.
```

Otherwise, the next time you log in, *Mail* prints:

```
You have mail.
```

To read your incoming messages, enter the command:

```
% Mail
```

*Mail* displays the name of your system mailbox and a list of *message headers,* then prompts with an ampersand (&) to indicate that it is ready to accept commands. For example, if two new messages had arrived, *Mail* might print:

```
% Mail
"/usr/spool/mail/jones": 2 messages 2 new
>N  1 root      Wed Sep 21 09:21 "Tuition fees"
 N  2 smith     Tue Sep 20 22:55 "Project status"
&
```

Each message header contains the following information:

- The status of the message. N indicates a new message, that is, one that arrived since you last read your mail. U indicates an old, but unread message. Nothing (a blank space) indicates a message that has been read.
- The message number.
- The name of the sender.
- The date and time of arrival.
- The subject of the message.

To read a message, type the `t` (for **t**ype) command followed by the message number. For example:

```
& t1
```

As with postal mail, most of what you receive can be read once and discarded. To delete a specific message, type the `d` (for **d**elete) command followed by the message number. For example:

```
& d1
```

Deleting a message is not permanent until you leave *Mail.* If you make a mistake, you can retrieve deleted messages with the `u` (for **u**ndelete) command. For example:

```
& u1
```

To display a list of the messages remaining in your system mailbox, type the `h` (for **h**eaders) command. *Mail* prints a list of message headers just as it did when you first started it up. For example:

```
& h
>   1 root      Wed Sep 21 09:21 "Tuition fees"
 N  2 smith     Tue Sep 20 22:55 "Project status"
&
```

When you are finished, type the `q` (for **q**uit) command. Messages that you have read but not deleted are stored in the file `mbox` in your login directory. Messages that you

have *not* read remain in your system mailbox. *Mail* prints a message summarizing what you have done and returns you to the shell. For example:

```
& q
Saved 1 message in mbox.
Held 1 message in /usr/spool/mail/jones
%
```

### 10.2.1 Some Shortcuts

When *Mail* displays a list of message headers, the left angle bracket (>) indicates the *current message*, which is the message that *Mail* commands operate on by default. For example: if the angle bracket is pointing to message 3, you can read that message by simply typing:

```
& t
```

To read the next message (advance the angle bracket and type), use the n (for next) command. For example:

```
& n
```

Because it is the default command, it is not necessary to type n. You can simply press RETURN. Thus, you can read through all your messages by pressing RETURN*. When you have read the last message, *Mail* prints:

```
At EOF
&
```

A convenient way to delete the message that you just read and type out the next one is:

```
& dt
```

(which stands for delete-type).

### 10.2.2 Reading Old Messages

To read old messsages (those stored in mbox when you previously quit *Mail*), type:

```
% Mail -f
```

All of the *Mail* commands described in the previous section work here as well.

## 10.3 SENDING MESSAGES

To send a message from within the *Mail* program, type the m (for mail) command followed by the *name* of the recipient (of which there can be several). For example:

```
& m smith jones adams
```

---

* When you first invoke *Mail*, you can press RETURN to read the first new message as long as you have not entered any other *Mail* commands. Otherwise, RETURN prints the second new message.

*Mail* does not issue another prompt; it simply enters message input mode (as described in Section 10.5) waits for you to type your message. When you are finished, press CTRL D (hold down the CTRL key and type D) at the beginning of a line. *Mail* sends the message and prints another ampersand prompt. For example:

```
& m smith jones adams
This is to confirm our meeting next Friday at 4.
CTRL D
&
```

While you are entering a message, you may decide that you do not wish to send it after all. To abort the message, type CTRL C. *Mail* prints:

```
(Interrupt -- one more to kill letter)
```

Type CTRL C again. *Mail* aborts the message saves it in the file `~/dead.letter`. Once you have sent mail to someone, there is no way to undo the act, so be careful.

## 10.3.1 Replying to Messages

If, after reading a message, you wish to immediately send a reply, type the `r` (for reply) command. *Mail* begins a message addressed to the user who sent you the message. For example:

```
& r1
To: root
Subject: Re: Tuition fees
```

and waits for you to type in your reply, followed by a CTRL D, for example:

```
Thanks for the reminder
CTRL D
&
```

The `r` command copies the subject header from the original message so that correspondence about a particular matter tends to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found is also used. For example, if the letter had a `To:` header containing several recipients, *Mail* would arrange to send your reply to the same people as well.

Similarly, if the original message contained a `Cc:` (carbon copies to) field, *Mail* would send your reply to those users as well. *Mail*, however, does not normally send the message to *you*, even if you appear in the `To:` or `Cc:` field.

The `r` command is thus useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation. To reply *only* to the person who sent a message, use the `R` command instead..

## 10.3.2 Sending Messages From the Shell

There is somewhat more convenient way to send messages. At shell command level, enter the *Mail* command and supply the name of the recipients. For example:

```
% Mail smith jones adams
This is to confirm our meeting next Friday at 4.
CTRL D
%
```

*Mail* waits for you to type the message and returns you to the shell.

### 10.3.3 Sending Prepared Messages

Most people prefer to use a text editor to prepare their messages before sending them. For example, suppose that the file memo contains the message you want to send. At shell command level, type:

```
% Mail smith jones adams <memo
```

The left angle bracket instructs the shell to use the contents of the file memo as the input to *Mail*.

## 10.4 MANIPULATING MESSAGES

Several *Mail* commands accept a list of messages as an argument. A *message list* consists of a list of message numbers and names, separated by spaces or tabs.

### 10.4.1 Message Numbers

A message number is a decimal number that directly specifies a message, a range, or one of the following special characters:

| | |
|---|---|
| \| | the first relevant message |
| . | the current message |
| $ | the last relevant message |

In this context, "relevant" means, for most commands "not deleted." For the *undelete* command, it means "deleted."

A range of message numbers consists of two message numbers separated by a dash. For example, to print the first four messages:

```
& t 1-4
```

To print all the messages from the current message to the last message:

```
& t .-$
```

### 10.4.2 Message Names

A message name is a string representing a user name. *Mail* scans the sender field of each message for names that match any of the specified message names.

If a message list consists entirely of user names, every message sent by one those users that is "relevant" (as described above) is selected. For example, to print every message sent to you by root:

```
t root
```

As a shorthand notation, you can specify an asterisk to get every "relevant" (same sense) message. For example, to print all undeleted messages:

```
t *
```

To delete all undeleted messages:

```
d *
```

To undelete all deleted messages:

```
u *
```

To search for the presence of a word in subject lines, preface the word with a slash character. For example, to print the headers of all messages that contain the word "PASCAL":

```
from /pascal
```

Subject searching ignores upper/lower case differences.

## 10.5 MESSAGE INPUT MODE

It is often useful to be able to invoke a text editor on a partial message, print a message, execute a shell command, or do some other auxiliary function. *Mail* provides these capabilities through *tilde escapes*: you type a tilde* (~) at the beginning of a line, followed by a single character that indicates the function to be performed. (To send a message that contains a line beginning with a tilde, use a double tilde.)

A full list of tilde escapes can be found in Chapter 11. Some of the commonly-used ones are:

~p prints a line of dashes, the recipients of your message, and the text of the message so far. Type a single CTRL C to abort the output of ~p or any other tilde escape without killing your letter.

~e copies your message into a temporary file and invokes a text editor on it. After editing the message to your satisfaction, write it out and quit the editor. You can continue typing text CTRL D to end the message. A standard text editor is provided by *Mail*. You can override this default by setting the valued option *EDITOR* to something else.

~v invokes a screen editor as an alternative to the standard text editor. The default is *vi* editor. You can set the valued option *VISUAL* to the path name of a different editor.

~r followed by a filename causes the named file to be appended to your current message. If the read is successful, the number of lines and characters appended to your message is printed, after which you can continue appending text. The

* On some terminals, the tilde character is difficult to type. If that is the case, see the escape option in Chapter 11.

filename can contain shell metacharacters that are expanded according to the conventions of your shell.

~w followed by a filename writes the text of your message to the specified file. *Mail* prints out the number of lines and characters written to the file, after which you can continue appending text to your message. Shell metacharacters can be used in the filename.

~f followed by a message number reads the specified message into the current message. This is the usual way to forward a message. You can name any non-deleted message, or list of messages.

~m is the same as ~f except that it reads the specified message into the current message, shifted right by one tab stop. This is useful for inserting comments into a forwarded letter.

~s followed by a string of text, replaces any previous subject with the specified text.

~! followed by a shell command, executes the specified command and returns you to mailing mode without altering the text of your message.

~| followed by a shell command, pipes the body of your message through the specified filter.

~: followed by a *Mail* command, executes the specified command. This is especially useful for retyping the message you are replying to, using, for example: `~:t`. It is also useful for setting options and modifying aliases.

~? prints out a brief summary of the available tilde escapes.

## 10.6 ORGANIZING MAIL

*Mail* can store messages in and read from *folders* (collections of messages other than your system mailbox). You can have as many folders as you like. All of the commands that you can use on your system mailbox are also applicable to folders.

### 10.6.1 Your System Mailbox

The mail system accepts your incoming messages and stores them in your system mailbox until you have time to read and/or dispose of them. Messages that you have not read are retained there automatically. Messages that you have read are stored in the default folder `~/mbox` (see Section 10.6.2) automatically. To retain a read message in your system mailbox, use the *hold* command (synonym: *preserve*).

### 10.6.2 The Default Folder ~/mbox

- When you enter the *quit* command, *Mail* automatically saves messages that you have read in the folder `~/mbox`. Thus, for convenience, `~/mbox` is the *default* folder. When reading your incoming mail, you can save an unread message in `~/mbox`; use the *mbox* command (which can be abbreviated to *mb*). To switch to the default folder:

```
& folder mbox
```

*Mail* closes your system mailbox, switches to the default folder, and prints a summary of its contents. To invoke *Mail* on the default folder:

```
% Mail -f
```

## 10.6.3 The Folder Directory

For convenience, all of your folders (other than `~/mbox`) can be kept in a single directory of your choosing. To tell *Mail* the name of your folder directory, insert the following line in your `.mailrc` file:

```
set folder=name
```

where *name* is the name of your folder directory. If *name* does not begin with a slash, *Mail* assumes that it is to be found in your home directory. For example, if your login name is `jones` and your folder directory name is `letters`, *Mail* would look for your folders in `/usr/jones/letters`.

## 10.6.4 Using Folders

To start *Mail* reading one of your folders, can use the *-f* option and precede the folder name with a plus sign. For example, this command causes *Mail* to read your `classwork` folder without looking at your system mailbox:

```
% Mail -f +classwork
```

Anywhere a file name is expected in a *Mail* command, you can use a folder name, preceded with a plus sign. For example, if you are in your system mailbox, you can put a message into the folder `classwork` with the *save* command:

```
save +classwork
```

If the `classwork` folder does not yet exist, it is created. By default, messages saved with the *save* command are automatically removed from your system mailbox.

To make a copy of a message in a folder without removing that message from your system mailbox, use the *copy* command, which is identical in all other respects to the *save* command. For example, this command copies the current message into the `classwork` folder and leaves a copy in your system mailbox.

```
copy +classwork
```

The *folder* command can be used to direct *Mail* to the contents of a different folder. For example, this command directs *Mail* to read the contents of the `project` folder:

```
folder +project
```

To inquire which folder you are currently in:

```
folder
```

To list your current set of folders:

```
folders
```

## 10.7 CUSTOMIZING MAIL

When it is invoked, *Mail* looks for and reads commands from two files:

| | |
|---|---|
| `/usr/lib/Mail.rc` | is maintained by the system administrator and contains *set* commands that are applicable to all users of the system. |
| `~/.mailrc` | contains user-specified *Mail* commands. |

Use your `.mailrc` file to define *aliases* and to set *options* as described below. For example, a typical `.mailrc` file might look like:

```
set append ask askcc keep quiet
set crt=24 folder=letters SHELL=/bin/csh
alias project sam sally steve susan
```

### 10.7.1 Aliases

An alias is simply a symbolic name that stands for one or more real user names. *Mail* sent to an alias is sent to the list of real users associated with it. For example, this *alias* command allows you to send mail to four different people via a single name: `project`.

```
alias project sam sally steve susan
```

Another use of *alias* is to provide a convenient name for someone whose user name is inconvenient or hard to remember. For example:

```
alias martha mh_smith
```

With this alias, you can send mail to `martha` without having to remember how to spell her actual login name.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail*. These aliases are stored in the file `/usr/lib/aliases` and are accessible to all users of the system. The lines in `/usr/lib/aliases` are of the form:

```
alias: name<1>, name<2>, name<3>
```

where *alias* is the mailing list name and the *name<i>* are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing `/usr/lib/aliases` since the delivery system uses an indexed file created by *newaliases*.

### 10.7.2 Options

*Mail* includes several options that allow you to tailor it to your liking. A complete list of the *Mail* options appears in Chapter 11.

The *set* command controls the options. Like shell variables, there are two kinds of *Mail* options, *binary* and *valued*. Binary options are either on or off. For example, if the `ask` option is on, *Mail* prompts you for a subject header each time you send a message. To set the *ask* option:

```
set ask
```

Another binary option is *hold*. By default, *Mail* moves read messages from your system mailbox to the file `~/mbox` when you leave *Mail*. If you want *Mail* to keep your letters in your system mailbox instead, set the *hold* option.

Valued options have string values. For example, the *SHELL* option tells *Mail* which shell you like to use:

```
set SHELL=/bin/csh
```

No spaces are allowed in the arguments to the *set* command.

Another important valued option is *crt*. If you use a fast video terminal, long messages may fly by too quickly for you to read them. This command sends any message longer than 24 lines through the paging program *more*:

```
set crt=24
```

*More* prints a screenful of information, then types `--MORE--`. Type a space to see the next screenful. (See the *Commands and Applications Manual* for details.)

## 10.8 SENDING MAIL OVER A NETWORK

This section describes how to send mail to people on other machines.

When you use the *reply* command to respond to a letter, there is a problem of figuring out the names of the users in the `To:` and `Cc:` lists *relative to the current machine*. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. Thus, when you *reply* to remote mail, the names in the `To:` and `Cc:` lists may change somewhat.

### 10.8.1 Arpanet

If your machine is directly (or sometimes, even, indirectly) connected to the Arpanet, you can send messages using a name of the form:

*name@host*

where *name* is the login name of the person you're trying to reach and *host* is the name of the machine where he logs in on the Arpanet.

### 10.8.2 Uucp

To send a message by *uucp* (the Bell Laboratories supplied network that communicates over telephone lines), you must know the list of machines through which your message must travel. If the recipient's machine is directly connected to yours:

*host!name*

where *host* is the name of the recipient's machine and *name* is his login name. If your message must go through an intermediate machine first, you must use the syntax:

*intermediate!host!name*

The map of all the systems in the *uucp* network is not known anywhere. See your system administrator for information about the machines available from your site.

### 10.8.3 Berknet

To send a message to a recipient on the Berkeley network (Berknet):

*host:name*

where *host* is the recipient's machine name and *name* is his login name. You need not know the names of the intermediate machines.

## 10.9 SENDING MAIL TO FILES AND PROGRAMS

Along with user names and aliases, you can send messages directly to files or to programs, using special conventions.

/ a recipient name that contains a slash character is the path name of a file into which to send the message. If the file already exists, the message is appended.

+ a recipient name that begins with a plus sign expands into the full path name of the folder name in your folder directory. If the folder already exists, the message is appended.

| a recipient name that begins with a vertical bar is a program.

The ability to send mail to files can be used for a variety of purposes, such as maintaining a journal or keeping a record of mail sent to a certain group of users. For example, with the following alias, all mail sent to `project` would be saved in the file `/usr/project/mail_record` as well as being sent to the members of the project:

```
alias project sam sally steve susan /usr/project/mail_record
```

The record file can be examined using `Mail -f`.

To specify a file in your current directory (one for which a slash would not usually be needed), precede the slash with a period. For example, to send mail to the file `memo` in your current directory:

```
% Mail ./memo
```

It is sometimes useful to send mail directly to a program. An alias can be set up to reference a vertical bar prefaced name. For example, the following command allows you to send mail to the `msgs` program:

```
alias msgs "| msgs -s"
```

The shell treats a vertical bar specially so it must be quoted. Also, the recipient program must be presented as a single argument to mail. It is recommended that you enclose the entire name with double quotes.

## 10.10 MESSAGE FORMAT

Messages begin with a *from* line, which consists of the word `From` followed by a user name, followed by anything, followed by a date in the format returned by the *ctime* library routine described in the *System Reference Manual.* For example:

```
Tue Dec 1 10:58:23 1981
```

The *ctime* date can be optionally followed by a single space and a time zone indication, in the form of three capital letters, such as EDT.

Following the *from* line are zero or more *header field* lines. Each header field line has the form:

*name*: *information*

*Name* can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are:

- article–id
- bcc
- cc
- from
- reply–to
- sender
- subject
- to

Other header fields are also significant to other systems. See, for example, the current Arpanet message standard for much more on this topic.

A header field can be continued onto following lines by making the first character on the following line a space or tab character. If any headers are present, they must be followed by a blank line.

The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line can be longer than 512 characters. Some network transport protocols enforce limits to the lengths of messages.

If binary data must be passed through the mail system, it is suggested that this data be encoded in a system that encodes six bits into a printable character. For example, one could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Thus, you can send a 16-bit binary number as three characters. Pack the characters into lines, preferably about 70 characters long as long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted. The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

# CHAPTER 11
# MAIL REFERENCE

## 11.1 COMMAND LINE OPTIONS

**-d** enables debugging information. Not of general interest.

**-v** is used when invoking sendmail to verify transmission(see option *verbose*).

**-i** ignores tty interrupt signals. Useful on noisy phone lines that generate spurious CTRL C or DELETE characters.

**-n** inhibits reading of `/usr/lib/Mail.rc`.

**-N** suppresses the initial printing of headers.

**-s** *string* specifies the subject of the message being composed. If *string* contains blanks, you must enclose it in quotes.

**-f** *file* shows the messages in the specified *file* instead of your system mailbox. The default *file* is `~/mbox`.

**-u** *name* reads a specified user's mail; equivalent to: `-f /usr/spool/`*name*.

The following command line flags are also recognized, but are intended for use by programs invoking *Mail* and not for people. The *-h* and *-r* options are for network mail forwarding and are not used in practice since mail forwarding is now handled separately.

**-T** *file* arranges to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. *-T* is for the *readnews* program. Do *not* use it for reading your mail.

**-h** *number* passes on hop count information. *Mail* takes the number, increments it, and passes it with *-h* to the mail delivery system. Only has effect when sending mail and is used for network mail forwarding.

**-r** *name* used for network mail forwarding: interprets *name* as the sender of the message. The *name* and *-r* are simply sent along to the mail delivery system. Also, *Mail* waits for the message to be sent and returns the exit status. Also restricts formatting of message.

## 11.2 COMMANDS

Each command is typed on a line by itself and can take arguments. The **bold** letters are all that is required to execute a command. For commands that take message lists as arguments, the default is the next message forward that satisfies the command's requirements. If there are none, the search proceeds backwards. If none is found, *Mail* prints `No applicable messages` and aborts the command.

| | |
|---|---|
| *–n* | goes to the *n*th previous message and prints it out (default *n* is 1). |
| ? | prints a brief summary of *Mail* commands. |
| ! | executes a shell command. |
| = | prints the current message number. |
| # | does nothing at all. |
| **a**lias | (also **g**roup) with no arguments, prints out all currently-defined aliases; with one argument, prints out that alias; with more than one argument, creates an new alias or changes an on old alias. |
| **a**lternates | informs *Mail* that the listed addresses are really you (useful if you have accounts on several machines). When you reply to messages, *Mail* does not send a copy of the message to any of the addresses listed. If given with no argument, the current set of alternate names is displayed. |
| **c**hdir | (also **cd**) changes your working directory to that specified (default is your login directory). |
| **c**lobber | clobbers part of the stack (for debugging purposes only). |
| **c**opy | does the same thing as save, except that it does not mark the messages for deletion when you quit. |
| **c**ore | writes a core dump to disk (for debugging purposes only). |
| **d**elete | marks a list of messages as deleted. Deleted messages are not saved in `mbox`, nor are they be available for most other commands. |
| **d**iscard | (also **i**gnore) prevents the specified header fields from begin printed on your terminal when you print a message. You can use it to suppress certain machine- generated header fields. The Type and Print commands can be used to print a message in its entirety, including ignored fields. With no arguments, ignore lists the current set of ignored fields. |
| **dp** | (also **dt**) deletes the current message and prints the next message. |
| **e**cho | expands file names like the shell command *echo*. |
| **e**dit | invokes the text editor on each of a list of messages in turn. For each message, *Mail* creates a temporary file `Messagen`, where *n* is the message number. When finished editing, write out your buffer and exit. *Mail* reads the file back in and removes it. |
| **e**lse | marks the end of the *then* clause of an *if* statement and the beginning of the clause to take effect if the condition of the *if* statement is false. |
| **e**ndif | marks the end of an *if* statement. |
| **e**xit | (also **x**) returns immediately to the shell without modifying your system mailbox, `mbox` file, or edit file in –f. |
| **f**ile | is the same as folder. |
| **f**older | switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it writes out |

changes (such as deletions) you have made in the current file and read in the new file. Some special conventions are recognized for the name:

| | |
|---|---|
| # | previous file read |
| % | your system mailbox |
| %name | *name*'s system mailbox |
| & | your ~/mbox file |
| +folder | a file in your folder directory |

**folders** lists the names of the folders in your folder directory.

**from** takes a list of messages and prints their message headers.

**group** (also **alias**) with no arguments, prints out all currently-defined aliases; with one argument, prints out that alias; with more than one argument, creates an new alias or changes an on old alias.

**headers** lists the current range of headers, which is an 18 message group. If a plus sign argument is given, the next 18 message group is printed, and if a minus sign argument is given, the previous 18 message group is printed.

**help** is a synonym for ?.

**hold** (also **preserve**) marks each specified message to be saved in your system mailbox instead of in mbox. It does not override the delete command.

**if** commands in your .mailrc file can be executed conditionally with the *if* command. The only allowed conditions are *receive* and *send*. For example:

```
if receive
        commands...
endif
```

An *else* form is also available:

```
if send
        commands...
else
        commands...
endif
```

**ignore** (also **discard**) prevents the specified header fields from begin printed on your terminal when you print a message. You can use it to suppress certain machine- generated header fields. The Type and Print commands can be used to print a message in its entirety, including ignored fields. With no arguments, ignore lists the current set of ignored fields.

**list** lists the vaild *Mail* commands.

**local** list other names for the specifid host.

**mail** sends mail to a specified list of login names and distribution group names.

**mbox** (also **touch**) sends a list of messages to mbox in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

**next** goes to the next message in sequence and types it. With an argument list, it types the next matching message.

**preserve** is a synonym for hold.

**print** (also **type**) takes a message list and types out each message on your terminal.

**Print** (also **Type**) is like print but also prints out ignored header fields. See also ignore.

| | |
|---|---|
| **quit** | terminates *Mail*, saves all undeleted, unsaved messages in your `mbox` file, preserves all messages marked with hold or preserve or never referenced in your system mailbox, and removes all other messages from your system mailbox. If new mail has arrived during the session, the message `You have new mail` is given. If given while editing a mailbox file with the –f flag, the edit file is rewritten. *Mail* returns to the shell unless the rewrite of edit file fails, in which case you can escape with the exit command. |
| **reply** | (also **respond**) takes a message list and sends mail to the sender and all recipients of the specified message. The default message must not be deleted. |
| **Reply** | (also **Respond**) reply to originator, not to other recipients of the original message. |
| **save** | takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on your terminal. |
| **set** | with no arguments, prints all variable values. Otherwise, it sets options. Arguments are of the form "option=value" or "option." |
| **shell** | invokes an interactive version of the shell. |
| **size** | takes a message list and prints out the size in characters of each message. |
| **source** | reads *Mail* commands from a file. |
| **top** | takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable *toplines* and defaults to five. |
| **touch** | (also **mbox**) sends a list of messages to `mbox` in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set. |
| **type** | (also **print**) takes a message list and types out each message on your terminal. |
| **Type** | (also **Print**) is like type but also prints out ignored header fields. See also ignore. |
| **unalias** | takes a list of names defined by alias commands and discards them. |
| **undelete** | takes a message list and marks each one as *not* being deleted. |
| **unset** | takes a list of option names and discards their remembered values; the inverse of set. |
| **version** | identifies the current version of *Mail*. |
| **visual** | takes a message list and invokes the display editor on each message. |
| **write** | is synonym for save. |
| **xit** | is a synonym for exit. |
| **z** | presents message headers in windowfuls as described under the headers command. You can move forward to the next window with the z command. Also, you can move to the previous window by using z–. |

## 11.3 TILDE ESCAPES

Tilde escapes are used to perform special functions when composing messages and are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option escape.

| | |
|---|---|
| **~c** *name ...* | adds the given *name*s to the list of carbon copy recipients. |
| **~d** | reads the file `~/dead.letter` into the message. |
| **~e** | invokes the text editor on the message collected so far. After the editing session is finished, you can continue appending text to the message. |
| **~f** *messages* | reads the named *messages* into the message being sent. If no messages are specified, it reads in the current message. |
| **~h** | edits the message header fields by typing each one in turn and allowing you to append text to the end or modify the field by using the current terminal erase and kill characters. |
| **~m** *messages* | reads the named *messages* into the message being sent, shifted right one tab. If no messages are specified, it reads the current message. |
| **~p** | prints out the message collected so far, prefaced by the message header fields. |
| **~q** | aborts the message being sent, copying the message to `~/dead.letter` if save is set. |
| **~r** *filename* | reads the named file into the message. |
| **~s** *string* | causes the named *string* to become the current subject field. |
| **~t** *name ...* | adds the given *name*s to the direct recipient list. |
| **~v** | invokes the editor defined by the VISUAL option (usually a screen editor) on the message collected so far. After you quit the editor, you can resume appending text to the end of your message. |
| **~w** *filename* | writes the message onto the named file. |
| **~!***command* | executes the indicated shell *command*, then returns to the message. |
| **~\|***command* | pipes the message through the *command* as a filter. If the command gives no output or terminates abnormally, it retains the original text of the message. The command *fmt* is often used as *command* to rejustify the message. |
| **~:***command* | executes the indicated *Mail* command, then returns to the message. |
| **~~***string* | inserts *string* into the message prefaced by a tilde. If you have redefined the escape character, double that character in order to send it. |

## 11.4 MAIL OPTIONS

Options are controlled via the set and unset commands. Options can be either binary, in which case it is only significant to see whether they are set or not, or string, in which case the actual value is of interest. The binary options include the following:

| | |
|---|---|
| **append** | causes messages saved in `mbox` to be appended rather than prepended. |
| **ask** | prompts for the subject of each message you send. If you respond with a newline, no subject field is sent. |

**askcc** prompts for additional carbon copy recipients at the end of each message. Responding with a newline indicates that there are no additional recipients.

**autoprint** causes the delete command to behave like the dp command, thus, after deleting a message, the next one is typed automatically.

**debug** is the same as specifying –d on the command line. It outputs all sorts of information useful for debugging *Mail*.

**dot** causes *Mail* to interpret a period alone on a line as the terminator of a message you are sending.

**hold** holds messages in the system mailbox by default.

**ignore** causes interrupt (CTRL C) signals from your terminal to be ignored and echoed as @'s (equivalent to the –i command line option).

**ignoreeof** refuses to accept a CTRL D as the end of a message (use with the dot option). *Ignoreeof* also applies to *Mail* command mode.

**keep** truncates your system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox.

**keepsave** retains all saved messages. By default, *Mail* discards all saved messages when you *quit*.

**metoo** prevents the default removal of the sender from a group of recipients.

**noheader** suppresses the printing of the version and headers when *Mail* is first invoked (same as using *–N* on the command line).

**nosave** prevents, when you abort a message, copying the partial letter to the file `~/dead.letter`.

**quiet** suppresses the printing of the version when first invoked as well as printing the for example `Message 4:` from the *type* command.

**verbose** is the same as using the –v flag on the command line. When *Mail* runs in verbose mode, the actual delivery of messages is displayed on your terminal.

The following options have string values:

**crt** specifies how long a message must be before *more* is used to read it.

**EDITOR** specifies the pathname of the text editor to use in the edit command and ~e escape. If not defined, a default editor is used.

**escape** specifies the character to use in the place of tilde to denote escapes.

**folder** specifies the name of the directory to use for storing folders of messages. If the name begins with a slash, *Mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

**record** specifies the pathname of the file used to record all outgoing mail. If not defined, outgoing mail is not so saved.

**screen** specifies how many message headers you want printed; also used for scrolling with the *z* command.

**sendmail** specifies the full pathname of an alternate delivery system. Note: most people should use the default delivery system.

**SHELL** is the pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.

**toplines** specifies the number of lines of a message to be printed out by the top command; the default is five.

**VISUAL** specifies the pathname of the text editor to use in the visual command and ~v escape.

# CHAPTER 12

# DISPLAY EDITING WITH VI

- Version 3.5

*Vi* (pronounced *vee-eye*) is a display-oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file that you are editing. Changes that you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like *d* for delete and *c* for change, are combined with the motion commands to form operations such as delete word or change paragraph, in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the *vi* command set easy to remember and to use.

*Vi* works on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. While it is advantageous to have an intelligent terminal that can locally insert and delete lines and characters from the display, *vi* functions quite well on dumb terminals over slow phone lines. It makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and "glass tty's" using a one line editing window; thus *vi's* command set is available on all terminals. The full command set of the more traditional, line oriented editor *ex* is available within *vi*; it is quite simple to switch between the two modes of editing.

## 12.1 GETTING STARTED

This document provides a quick introduction to *vi*. Run *vi* on a file you are familiar with while you are reading this. The first part of this document (Sections 12.1 through 12.5) describes the basics of using *vi*. Some topics of special interest are presented in Section

12.6, and some details of how *vi* functions are saved for Section 12.7 to avoid cluttering the presentation here. There is also a short section that gives for each character the special meanings that it has in *vi*.

### 12.1.1 Specifying Terminal Type

Before you can start *vi* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be 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 act4 | 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'. In this case you can use one of the following commands to tell the system the type of your terminal. If you are using the C shell:

```
% setenv TERM 2621
```

If you are using the Bourne shell:

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use csh) would be

```
setenv TERM `tset - -d mime`
```

or for your *.profile* file (if you use sh)

```
TERM=`tset - -d mime`
```

*Tset* knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

### 12.1.2 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, giving the command

```
% vi name
```

replacing *name* with the name of the copy file you just created. The screen clears and the text of your file appears on the screen. If something else happens, one of the following may apply:

If you gave the system an incorrect terminal type code, *vi* may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case press the keys :q (colon and the q key) and then press RETURN. This gets you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

If *vi* just printed an error diagnostic, you may typed the wrong file name. Get out of *vi* following the procedure above and try again, this time spelling the file name correctly.

If *vi* doesn't seem to respond to the commands that you type here, try sending it an interrupt by pressing CTRL C, then pressing the :q command again followed by RETURN.

### 12.1.3 The Buffer

*Vi* does not directly modify the file that you are editing. Rather, it makes a copy of the file, in a place called the *buffer*, and remembers the file's name. You do not affect the contents of the file unless and until you write the changes you make back into the original file.

### 12.1.4 Notational Conventions

In our examples, input that must be typed as shown is presented in `this font`. Text that you replace with appropriate input is given in *italics*. A combination such as CTRL C means that you hold down the CTRL key and type C (upper or lower case).

### 12.1.5 Arrow Keys

The *vi* command set is independent of the terminal you are using. On most terminals, cursor positioning keys also work within *vi*. 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. As shown later, *h* moves back to the left (like CTRL h, which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and / moves to the right.

(Particular note for the HP2621: on this terminal the function keys must be *shifted* to send to the machine, otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.)

### 12.1.6 Special Characters: ESCAPE, RETURN, and CTRL C

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESCAPE or ALT. It should be near the upper left corner of your terminal. Try pressing this key a few times. *Vi* rings the bell to indicate that it is in a quiescent state. (On smart terminals where it is possible, *vi* quietly flashes the screen rather than ringing the bell.) Partially formed commands are cancelled by ESCAPE, and when you insert text in the file you end the text insertion with ESCAPE. This key is a fairly harmless one to press, so you can just press it if you don't know what is going on until *vi* rings the bell.

The 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.

CTRL C generates an interrupt, telling *vi* to stop what it is doing. It is a forceful way of making *vi* listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try pressing the '/' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor is 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 pressing CTRL C ; try this now. (Backspacing over the '/' also cancels the search.) From now on, CTRL C is referred to as "sending an interrupt."

> *Note: On some systems, this interruptibility comes at a price: you cannot type ahead when* vi *is computing with the cursor on the bottom line.*

*Vi* often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, *vi* is performing a computation, such as a computing 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 *vi* by sending an interrupt.

### 12.1.7 Getting Out of the Editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command ZZ. This writes the contents of *vi*'s buffer back into the file you are editing, if you made any changes, and then quit from *vi*. You can also end an editor session by giving the command* :q!; this is a dangerous but occasionally essential command that ends the editing session and discards all your changes. You need to know about this command in case you change *vi*'s 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.

## 12.2 MOVING AROUND IN THE FILE

### 12.2.1 Scrolling and Paging

*Vi* has a number of commands for moving around in the file. The most useful of these is CTRL D, which scrolls down in the file. The D thus stands for down. Many editor com-

* All commands that read from the last display can be terminated with ESCAPE or RETURN.

mands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is CTRL U. Many dumb terminals can't scroll up at all, in which case pressing CTRL 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, press CTRL E to expose one more line at the bottom of the screen, leaving the cursor where it is.The command CTRL Y (which is hopelessly non-mnemonic, but next to CTRL 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 CTRL F and CTRL 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 rather than CTRL D and CTRL U if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, pressing CTRL F to move forward a page leaves you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

## 12.2.2 Searching, Goto, and Previous Context

Another way to position yourself in the file is by searching for a string. Type the character / followed by a string of characters terminated by RETURN. *Vi* positions the cursor at the next occurrence of this string. Try pressing n to then go to the next occurrence of this string. The character ? searches backwards from where you are, and is otherwise like /.

These searches normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command :se nowrapscan RETURN, or more briefly :se nows RETURN.

If the search string is not present in the file, *vi* prints a diagnostic on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a ^. To match only at the end of a line, end the search string with a $. Thus /^search RETURN searches for the word 'search' at the beginning of a line, and /last$ RETURN searches for the word 'last' at the end of a line.

Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex* and *ed*. If you don't wish to learn about this yet, you can disable this more general facility by doing :se nomagic RETURN; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

The command G, when preceded by a number positions the cursor at that line in the file. Thus 1G moves the cursor to the first line of the file. If you give G no count, it 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, *vi* places 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 past the end of the file.

You can find out the state of the file you are editing by typing a CTRL G. *Vi* shows 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 that you are. 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 to where you were. If you accidentally press n or any command that moves you far away from a context of interest, you can quickly get back by pressing ``.

## 12.2.3 Moving Around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (four or five keys with arrows going in each direction) try them and convince yourself that they work. If you don't have working arrow keys, you can always use *h*, *j*, *k*, and *l*. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

Press 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 up and down lines in the file. Notice that if you go off the bottom or top with these keys, the screen scrolls down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the + key.

*Vi* also has commands to take you to the top, middle and bottom of the screen. H takes you to the top (home) line on the screen. Try preceding it with a number as in 3H. This takes you to the third line on the screen.

Many *vi* commands take preceding numbers and do interesting things with them. Try M, which takes you to the middle line on the screen, and L, which takes you to the last line on the screen. L also takes counts, thus 5L takes you to the fifth line from the bottom.

## 12.2.4 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 pressing the w key. This advances the cursor to the next word on the line. Try pressing the b key to back up words in the line. Also try the e key that 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 right one character and the BACKSPACE (backspace or CTRL H) key that moves left one character. The key h works as CTRL H does and is useful if you don't have a BACKSPACE key. (Also, as noted just above, l moves to the right.)

If the line had punctuation in it you may have noticed that that the w and b keys stopped at each group of punctuation. You can also go back and forwards words without stopping at punctuation by using W and B rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case 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 pressing w.

### 12.2.5 Summary

| | |
|---|---|
| SPACE | advance the cursor one position |
| CTRL B | backwards to previous page |
| CTRL D | scrolls down in the file |
| CTRL E | exposes another line at the bottom |
| CTRL F | forward to next page |
| CTRL G | tell what is going on |
| CTRL H | backspace the cursor |
| CTRL N | next line, same column |
| CTRL P | previous line, same column |
| CTRL U | scrolls up in the file |
| CTRL Y | exposes another line at the top |
| + | next line, at the beginning |
| – | previous line, at the beginning |
| / | scan for a following string forwards |
| ? | scan backwards |
| B | back a word, ignoring punctuation |
| G | go to specified line, last default |
| H | home screen line |
| M | middle screen line |
| L | last screen line |
| W | forward a word, ignoring punctuation |
| b | back a word |
| e | end of current word |
| n | scan for next instance of / or ? pattern |
| w | word after this word |

### 12.2.6 View

If you want to use *vi* to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This sets the *readonly* option that prevents you from accidently overwriting the file.

## 12.3 MAKING SIMPLE CHANGES

### 12.3.1 Inserting

One of the most useful commands is the i (insert) command. After you type i, everything you type until you press ESCAPE is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it seems, for a minute, that some of the characters in your line have been overwritten, but they reappear when you press ESCAPE.

Now try finding a word that can, but does not, end in an 's'. Position yourself at this word and type e (move to end of word), then a for append and then 's ESCAPE' to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

Try inserting and appending a few times to make sure you understand how this works; i placing text to the left of the cursor, a to the right.

It is often the case that 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 command o to create a new line after the line you are on, or the command O to create a new line before the line you are on. After you create a new line in this way, text you type up to an ESCAPE is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

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, press a RETURN at the middle of your input. A new line is created for text, and you can continue to type. If you are on a slow and dumb terminal, *vi* may choose to wait to redraw the tail of the screen, and let you type over the existing screen lines. This avoids the lengthy delay that would occur if *vi* attempted to keep the tail of the screen always up to date. The tail of the screen is fixed up, and the missing lines reappear, when you press ESCAPE.

While you are inserting new text, you can use the characters you normally use at the system command level (usually CTRL H or #) to backspace over the last character that you typed, and the character that you use to kill input lines (usually @, CTRL X, or CTRL U) to erase the input you have typed on the current line.+ The character CTRL W erases a whole word and leaves you after the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion the characters you backspace over are not erased; the cursor moves backwards, 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 when you press ESCAPE; if you want to get rid of them immediately, press an ESCAPE and then a again.

Notice also that you can't erase characters that you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just press ESCAPE 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.

## 12.3.2 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 (press the BACKSPACE key or CTRL H or even just h) or SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed, press 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 (except it's not as messy).

---

* In fact, the backspace (CTRL H) character always works to erase the last input character here, regardless of what your erace character is.

If the character is incorrect, you can replace it with the correct character by giving the command r*c*, where *c* is replaced by the correct character. Finally if the character that is incorrect should be replaced by more than one character, give the command s which substitutes a string of characters, ending with ESCAPE, for it. 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.

## 12.3.3 More Corrections: 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. Try the command *dw* to delete a word. Try pressing . a few times. Notice that this repeats the effect of the dw. The command . repeats the last command that made a change. You can remember it by analogy with an ellipsis '...'.

Now try db. This deletes a word backwards, namely the preceding word. Try dSPACE. This deletes a single character, and is equivalent to the x command.

Another very useful operator is *c* or change. The command *cw* thus changes the text of a single word. You follow it by the replacement text ending with an ESCAPE. Find a word that you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

## 12.3.4 Operating on Lines

It is often the case that you want to operate on lines. Find a line that you want to delete, and type dd, the *d* operator twice. This deletes the line. If you are on a dumb terminal, *vi* may just 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.

Try repeating the *c* operator twice; this changes a whole line, erasing its previous contents and replacing them with text you type up to an ESCAPE. (The command S is a convenient synonym for for cc, by analogy with s. Think of S as a substitute on lines, while s is a substitute on characters.)

You can delete or change more than one line by preceding the *dd* or *cc* with a count, i.e. 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. Try some commands like this now. Notice that *vi* lets you know when you change a large number of lines so that you can see the extent of the change. It also always tells you when a change you make affects text that you cannot see.

> *Note: One subtle point here involves using the / search after a d. This normally deletes characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as /pat/+0, a line address.*

### 12.3.5 Undoing

Now suppose that the last change that 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 you regret a change or make a change incorrectly, *vi* provides a *u* (undo) command to reverse the last change that you made. Try this a few times, and give it twice in a row to notice that an *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 delete, even if undo will not bring it back; see the section on recovering lost text below.

### 12.3.6 Summary

| | |
|---|---|
| SPACE | advance the cursor one position |
| CTRL H | backspace the cursor |
| CTRL W | erase a word during an insert |
| erase | your erase (usually CTRL H or #), erases a character during an insert |
| kill | your kill (usually @, CTRL X, or CTRL U), kills the insert on this line |
| . | repeats the changing command |
| O | opens and inputs new lines, above the current |
| U | undoes the changes you made to the current line |
| a | appends text after the cursor |
| c | changes the object you specify to the following text |
| d | deletes the object you specify |
| i | inserts text before the cursor |
| o | opens and inputs new lines, below the current |
| u | undoes the last change |

## 12.4 MOVING ABOUT: REARRANGING AND DUPLICATING TEXT

### 12.4.1 Low Level Character Motions

Now 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 this character. This command finds the next *x* character to the right of the cursor in the current line. Try then pressing 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 SPACE s. There is also a F command, which is like f, but searches backward. The ; command repeats F also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try df*x* for some *x* now and notice that the *x* character is deleted. Undo this with u and then try dt*x*; the t here stands for to, i.e. 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, a ^ moves the cursor to the first non-white position on the line, and a $ moves it to the end of the line. Thus $a appends new text at the end of the current line.

Your file may have tab (CTRL I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions. (This is settable by a command of the form :se ts=*x* RETURN, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within *vi*.) 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 nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character reveals that the two characters are, like the spaces representing a tab character, a single character.

*Vi* 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 CTRL V before the control character. The CTRL V quotes the following character, causing it to be inserted directly into the file.

## 12.4.2 Higher Level Text Objects

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations ( and ) move to the beginning of the previous and next sentences respectively. Thus the command d) deletes the rest of the current sentence; likewise d( deletes the previous sentence if you are at the beginning of the current sentence, or 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 '?' which is followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and ''' characters can appear after the '.', '!' or '?' before the spaces or end of line.

The operations { and } move over paragraphs and the operations [[ and ]] move over sections.+

> *Note: The [[ and ]] operations require the operation character to be doubled because they can move the cursor far 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 press 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. 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.

You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See Section 12.6.2 for details. The '.bp' directive is also considered to start a paragraph.

Sections in *vi* begin after each macro in the *sections* option, normally '.NH', '.SH', '.H' and '.HU', and each line with a formfeed CTRL L in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The 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 a small count to then see each successive section heading in a small window.

## 12.4.3 Rearranging and Duplicating Text

*Vi* has a single unnamed buffer where the last deleted or changed away 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 operator *y* yanks a copy of the object that follows into the unnamed buffer. If preceded by a buffer name, "*x*y, where *x* here 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*; p 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 that partially spans more than one line, then when you put the text back, it is placed after the cursor (or before if you use P). If the yanked text forms whole lines, they are put back as whole lines, without changing the current line. In this case, the put acts much like a o or O command.

Try the command YP. 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 also makes a copy of the current line, and places 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 the these lines and do a "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*RETURN where *name* is the name of the other file you want to edit. If you have made changes, you have to write back the contents of the current editor buffer (or discard them) before *vi* lets 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, use a named buffer.

## 12.4.4 Summary

| | |
|---|---|
| ^ | first non–white on line |
| $ | end of line |
| ) | forward sentence |

| | |
|---|---|
| } | forward paragraph |
| ]] | forward section |
| ( | backward sentence |
| { | backward paragraph |
| [[ | backward section |
| f*x* | find *x* forward in line |
| p | put text back, after cursor or below current line |
| y | yank operator, for copies and moves |
| t*x* | up to *x* forward, for operators |
| F*x* | f backward in line |
| P | put text back, before cursor or above current line |
| T*x* | t backward in line |

## 12.5 HIGH LEVEL COMMANDS

### 12.5.1 Writing, Quitting, Editing New Files

So far we have seen how to enter *vi* and to write out our file using either ZZ or :w[RETURN]. The first exits from *vi*, (writing if changes were made), the second writes and stays in *vi*.

If you have changed *vi*'s copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, then you can give the command :q![RETURN] to quit from *vi* without writing the changes. You can also reedit the same file (starting over) by giving the command :e![RETURN]. Use these commands rarely, and with caution, as 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 *vi* by giving the command :e *name*[RETURN]. If you have not written out your file before you try to do this, *vi* tells you this, and delays editing the other file. You can then give the command :w[RETURN] to save your work and then the :e *name*[RETURN] command again, or carefully give the command :e! *name*[RETURN], which edits the other file discarding the changes you have made to the current file. To have *vi* automatically save changes, include *set autowrite* in your EX-INIT, and use :n instead of :e.

### 12.5.2 Escaping to a Shell

You can get to a shell to execute a single command by giving a *vi* command of the form :!*cmd*[RETURN]. The system runs the single command *cmd* and, when the command finishes, *vi* asks you to press a [RETURN] to continue. When you have finished looking at the output on the screen, press [RETURN] and *vi* clears and redraws the screen. You can then continue editing. You can also give another : command when it asks you for a [RETURN]; in this case the screen is not redrawn.

If you wish to execute more than one command in the shell, give the command :sh[RETURN]. This gives you a new shell, and when you finish with the shell, ending it by typing a [CTRL]D, *vi* clears the screen and continues.

On systems that support it, CTRL Z suspends *vi* and returns to the (top level) shell. When *vi* is resumed, the screen is redrawn.

### 12.5.3 Marking and Returning

The command '' returned 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*, and pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and press 'a. The cursor returns 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* moves 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 ''.

### 12.5.4 Adjusting the Screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than *vi* wrote output to your terminal, you can press a CTRL L, the ASCII form–feed character, to 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 can get rid of these lines by typing CTRL R to cause *vi* to retype the screen, closing up these holes.

Finally, 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. 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.

## 12.6 SPECIAL TOPICS

### 12.6.1 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 do not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how *vi* optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how *vi* erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force *vi* to use this mode even on faster terminals by giving the command :se slowRETURN. If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by :se noslowRETURN.

*Vi* can simulate an intelligent terminal on a dumb one. Try giving the command :se redraw[RETURN]. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command

```
:se noredraw[RETURN].
```

*Vi* also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. *Vi* can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try *vi* on an intelligent terminal to see how this works.

You can control the size of the window that is redrawn each time the screen is cleared by giving window sizes as argument to the commands that cause large screen motions:

```
: / ? [[ ]] ` ´
```

Thus if you are searching for a particular instance of a common string in a file you can precede the first search command by a small number, say 3, and *vi* draws three line windows around each instance of the string that it locates.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a 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 five line window.

> *Note: The command 5z. has an entirely different effect, placing line 5 in the center of a new window.*

If *vi* is redrawing or otherwise updating large portions of the display, you can interrupt this updating by pressing [CTRL]C as usual. If you do this you may partially confuse *vi* about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by pressing a [CTRL]L; or move or search again, ignoring the current state of the display.

See Section 12.7.8 on *open* mode for another way to use the *vi* command set on slow terminals.

## 12.6.2 Options, Set, and Editor Startup Files

*Vi* has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

| Name | Default | Description |
|---|---|---|
| autoindent | noai | Supply indentation automatically |
| autowrite | noaw | Automatic write before :n, :ta, [CTRL]^, ! |
| ignorecase | noic | Ignore case in searching |
| lisp | nolisp | ( { ) } commands deal with S–expressions |
| list | nolist | Tabs print as [CTRL]I; end of lines marked $ |
| 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 [CTRL]D and [CTRL]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 options, string options, and toggle options. 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 noopt
```

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 RETURN.

You can get a list of all options that you have changed by the command :set RETURN, or the value of a single option by the command :set *opt*? RETURN. A list of all possible options and their values is generated by :set all RETURN. Set can be abbreviated se. Multiple options can be placed on one line, e.g. :se ai aw nu RETURN.

Options set by the set command only last while you stay in *vi*. It is common to want to have certain options set whenever you use *vi*. This can be accomplished by creating a list of *ex* commands that are to be 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. Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

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

which sets the options *autoindent*, *autowrite*, *terse*, (the *set* command), makes @ delete a line, (the first *map*), and makes # delete a character, (the second *map*). (See Section 12.6.9 for a description of the map command.) Place this string in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

```
setenv EXINIT 'set ai aw terse|map @ dd|map # x'
```

If you use the Bourne shell, put these lines in the file *.profile* in your home directory:

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

Of course, the particulars of the line would depend on which options you wanted to set.

## 12.6.3 Recovering Lost Lines

If you delete a number of lines and then regret that they were deleted, you can recover the lost lines. *Vi* saves the last nine deleted blocks of text in a set of numbered registers 1–9. You can get the *n*'th previous deleted text back in your file 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, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, press *u* to undo this and then . (period) to repeat the put command. In general the . command repeats the last change you made. 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, shows you all the deleted text that has been saved for you. 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.

## 12.6.4 Recovering Lost Files

If the system crashes, you can recover the work you were doing to within a few changes. You normally receive mail when you next login giving you the name of the file that has been saved for you. 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 recovers your work to a point near where you left off.

> *Note: In rare cases, some of the lines of the file may be lost.* Vi *gives you the numbers of these lines and the text of the lines is replaced by the string 'LOST'. These lines almost always are among the last few that you changed. You can either choose to discard the changes that you made (if they are easy to remake) or to replace the few lost lines by hand.*

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, *vi* gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation "*vi -r*" does not always list all saved files, but they can be recovered even if they are not listed.

## 12.6.5 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 giving the command :se wm=10RETURN. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.

If *vi* 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. *Vi* 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.

## 12.6.6 Features for Editing Programs

*Vi* has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. *Vi* has a *autoindent* facility for helping you generate correctly indented programs.

To enable this facility you can give the command :se ai[RETURN]. Now try opening a new line with o and type some characters on the line after a few tabs. If you now start another line, notice that *vi* 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 [CTRL]D key to backtab over the supplied indentation.

Each time you type [CTRL]D you back up one position, normally to an 8 column boundary. This amount is settable; *vi* has an option called *shiftwidth* that you can set to change this value. Try giving the command :se sw=4[RETURN] 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.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and press %. This shows you the matching parenthesis. This works also for braces { and }, and brackets [ and ].

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 at a time. When ]] is used with an operator it stops after a line that starts with }; this is sometimes useful with y]].

## 12.6.7 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 with a pretty–printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command !}sort[RETURN]. This says to sort the next paragraph of material, and the blank line ends a paragraph.

## 12.6.8 Commands for Editing LISP

If you are editing a LISP program, set the option *lisp* by doing :se lisp[RETURN]. This changes the ( and ) commands to move backward and forward over s–expressions. The { and } commands are like ( 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[RETURN] and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens

only if the matching '(' is on the screen, and the cursor stays there for at most one second.

*Vi* 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 realigns 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.

## 12.6.9 Macros

*Vi* has a parameterless macro facility that lets you set it up so that when you press a single keystroke, *vi* acts as though you had press some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a) Ones where you put the macro body in a buffer register, say *x*. You can then type @x to invoke the macro. The @ can be followed by another @ to repeat the last macro.

b) You can use the *map* command from *vi* (typically in your *EXINIT*) with a command of the form:

```
:map lhs rhsRETURN
```

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* waits for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs,* escape them with a CTRL V. (It may be necessary to double the CTRL V if the map command is given inside *vi*, rather than in *ex*.) Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the q key write and exit *vi*, you can give the command

```
:map q :wqCTRL vCTRL vRETURN RETURN
```

which means that whenever you type q, it is as though you had typed the four characters :wqRETURN. A CTRL V is needed because without it the RETURN would end the : command, rather than becoming part of the *map* definition. There are two CTRL V's because from within *vi*, two CTRL V's must be typed to get one. The first RETURN 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", this maps the particular function key instead of the two-character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" means function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way:

```
:map CTRL vCTRL vCTRL I #
```

to use tab, for example. (This won't 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, if it made any changes.

Placing a '!' after the word *map* causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for CTRL T to be the same as 4 spaces in input mode, you can type:

```
:map CTRL T CTRL Vbbbb
```

where *b* is a blank. The CTRL V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

## 12.7 WORD ABBREVIATIONS

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* (:ab and *:una*) and have the same syntax as *:map*. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word 'eecs' to always 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.

### 12.7.1 Abbreviations

*Vi* has a number of short commands that abbreviate longer commands. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

## 12.8 OPERATIONAL DETAILS

### 12.8.1 Line Representation in the Display

*Vi* folds long logical lines onto many physical lines in the display. Commands that advance lines advance logical lines and skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be 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.)

*Vi* only puts full lines on the display; if there is not enough room on the display to fit a logical line, *vi* leaves the physical line empty, placing only an @ on the line as a place

holder. When you delete lines on a dumb terminal, *vi* often just clears the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the CTRL R command.

If you wish, you can have *vi* place line numbers before each line on the display. Give the command :se nu RETURN to enable this, and the command :se nonu RETURN to turn it off. You can have tabs represented as CTRL I and the ends of lines indicated with '$' by giving the command :se list RETURN; :se nolist RETURN turns this off.

Finally, 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.

## 12.8.2 Counts

Most *vi* commands use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

```
new window size     : / ? [[ ]] ` ´
scroll amount       CTRL D CTRL U
line/column number  z G |
repeat effect       most of the rest
```

*Vi* maintains a notion of the current default window size. On terminals that run at speeds greater than 1200 baud, *vi* uses the full terminal screen. On terminals that are slower than 1200 baud (most dialup lines are in this group) *vi* uses eight lines as the default window size. At 1200 baud the default is 16 lines.

This size is the size used when *vi* clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands that take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, 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 expands if you move off the top with a – or similar command or off the bottom with a command such as RETURN or CTRL D. The window reverts to the last specified size the next time it is cleared and refilled (but not by a CTRL L which just redraws the screen as it is).

The scroll commands CTRL D and CTRL 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+----ESCAPE inserts a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands that ignore any counts (such as CTRL R), the rest of *vi* commands use a count to indicate a simple repetition of their effect. Thus 5w advances five words on the current line, while 5RETURN 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 delete first one and then three words. You can then delete two more words with 2..

## 12.8.3 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! | reedit, discarding changes |
| :e + *name* | edit, starting at end |
| :e +*n* | edit, starting at line *n* |
| :e # | edit alternate file |
| :w *name* | write file *name* |
| :w! *name* | overwrite file *name* |
| :*x*,*y*w *name* | write lines *x* through *y* to *name* |
| :r *name* | read file *name* 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 RETURN or ESCAPE. The most basic commands are :w and :e. A normal editing session on a single file ends 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 *vi*'s copy of a file, but do not wish to write them back, you must give an ! after the command you would otherwise use; this forces *vi* 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* can be any editor command not containing a space, usefully a scan like +/*pat* or +?*pat*. In forming new names to the e command, you can use the character % that is replaced by the current file name, or the character # that is replaced by the alternate file name. The alternate file name 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 haven't written 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 out the lines that bound the range to be written using CTRL G, 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 here.

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 !*cmd* instead of a file name.

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 file names, or a pattern to be expanded as you would have given it on the initial *vi* command.

If you are editing large programs, you may 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 quickly find a function whose name you give. If the :ta command requires *vi* to switch files, 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.

## 12.8.4 More About Searching for Strings

When you are searching for strings in the file with / and ?, *vi* 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 wish 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 don't give a line offset, *vi* affects characters up to the match place, rather than whole lines; thus use "+0" to affect to the line that matches.

You can have *vi* ignore the case of words in the searches it does by giving the command :se ic[RETURN]. The command :se noic[RETURN] turns this off.

Strings given to searches can actually be regular expressions. If you do not want or need this facility, put

```
set nomagic
```

in your EXINIT. In this case, only the characters ^ and $ are special in patterns. The character \ is also then special (as it is most everywhere in the system), and can be used to get at the an extended pattern matching facility. It is also necessary to use a \ before a / in a forward scan or a ? in a backward scan, in any case. 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, the . [ and * primitives are given with a preceding \.

## 12.8.5 More About 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.

| | |
|---|---|
| [CTRL]H | deletes the last input character |
| [CTRL]W | deletes the last input word, defined as by b |
| erase | your erase character, same as [CTRL]H |
| kill | your kill character, deletes the input on this line |
| \ | escapes a following [CTRL]H and your erase and kill |
| [ESCAPE] | ends an insertion |
| [CTRL]C | interrupts an insertion, terminating it abnormally |

| | |
|---|---|
| RETURN | starts a new line |
| CTRL D | backtabs over *autoindent* |
| 0 CTRL D | kills all the *autoindent* |
| ^ CTRL D | same as 0 CTRL D, but restores indent next line |
| CTRL V | quotes the next non–printing character into the file |

The most usual way of making corrections to input is by typing CTRL H to correct a single character, or by typing one or more CTRL W's to back over incorrect words. If you use # as your erase character in the normal system, it works like CTRL H.

Your system kill character, normally @, CTRL X or CTRL U, erases 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 press ESCAPE 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 (say # or @), 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 CTRL V. The CTRL V echoes as a ^ character on which the cursor rests. This indicates that *vi* expects you to type a control character. In fact you can type any character and it is inserted into the file at that point.

> *Note: This is not quite true.* Vi *does not allow the NULL (*CTRL @*) character to appear in files. Also the linefeed (*LINE FEED *or* CTRL J*) character is used by* vi *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* vi *to echo the ^ before you type the character. In fact,* vi *treats a following letter as a request for the corresponding control character. This is the only way to type* CTRL S *or* CTRL Q*, since the system normally uses them to suspend and resume output and never gives them to* vi *to process.*

If you are using *autoindent* you can backtab over the indent that it supplies by typing a CTRL D. This backs up to a *shiftwidth* boundary. This only works 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 CTRL D. *Vi* moves the cursor to the left margin for one line, and restores the previous indent on the next. You can also type a 0 followed immediately by a CTRL D if you wish to kill all the indent and not have it come back on the next line.

## 12.8.6 Uppercase-only Terminals

If your terminal has only upper case, 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 lower case, and you can type upper case letters by preceding them with a \. The characters { ~ } | ‘ 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 does not echo until you type another key.

### 12.8.7 Vi and Ex

*Vi* is actually one mode of editing within the editor *ex*. When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command Q. All of the : commands that were introduced above are available in *ex*. Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the : and follow them with a RETURN.

In rare instances, an internal error can occur in *vi*. In this case you get a diagnostic and are left in the command mode of *ex*. You can then 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.

There are a number of things that you can do more easily in *ex* than in *vi*. Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out a lot more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

### 12.8.8 Open Mode: Vi on Hardcopy Terminals and "Glass tty's"

If you are on a hardcopy 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, *vi* tells 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, *vi* uses a single line window into the file, and 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*: *z* and CTRL R. The *z* command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the CTRL *R* command retypes the current line. On such terminals, *vi* 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, *vi* types a number of \'s to show you the characters that are deleted. *Vi* 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.

## 12.9 CHARACTER FUNCTIONS

This section gives the uses *vi* makes of each character. The characters are presented in their order in the ASCII character set: Control characters come first, then most special characters, then the digits, upper and then lower case characters.

For each character we tell a meaning it has as a command and any meaning it has during an insert. If it has only meaning as a command, only this is discussed. Section

numbers in parentheses indicate where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

CTRL @ 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 were inserted the mechanism is not available. A CTRL @ cannot be part of the file due to the implementation (12.7.5f).

CTRL A Unused.

CTRL B Backward window. A count specifies repetition. Two lines of continuity are kept if possible (12.2.1,12.6.1, 12.7.2).

CTRL C Unused.

CTRL D As a command, scrolls down a half–window of text. A count gives the number of (logical) lines to scroll, and is remembered for future CTRL D and CTRL U commands (12.2.1, 12.7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (12.6.6, 12.7.5); this white space cannot be backspaced over.

CTRL E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.

CTRL F Forward window. A count specifies repetition. Two lines of continuity are kept if possible (12.2.1, 12.6.1, 12.7.2).

CTRL G Equivalent to :f RETURN, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.

CTRL H (BACKSPACE) Same as *left arrow*. (See *h*). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (12.3.1, 12.7.5).

CTRL 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 which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (12.4.1, 12.6.6).

CTRL J (LINE FEED) Same as *down arrow* (see *j*).

CTRL K Unused.

CTRL L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than *vi* scramble the screen, or after output is stopped by an interrupt (12.5.4, 12.7.2f).

CTRL M (RETURN) Advances to the next line, at the first non–white position in the line. Given a count, it advances that many lines (12.2.3). During an insert, a RETURN causes the insert to continue onto another line (12.3.1).

CTRL N Same as *down arrow* (see *j*).

CTRL O Unused.

CTRL P Same as *up arrow* (see *k*).

CTRL Q Not a command character. In input mode, CTRL *Q* quotes the next character, the same as CTRL *V*, except that some teletype drivers eat the CTRL *Q* so that *vi* never sees it.

CTRL R — Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (12.5.4, 12.7.2, 12.7.8).

CTRL S — Unused. Some teletype drivers use CTRL *S* to suspend output until CTRL *Q* is typed.

CTRL T — Not a command character. During an insert, with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.

CTRL U — Scrolls the screen up, inverting CTRL D which scrolls down. Counts work as they do for CTRL D, and the previous scroll amount is common to both. On a dumb terminal, CTRL U often necessitates clearing and redrawing the screen further back in the file (12.2.1, 12.7.2).

CTRL V — Not a command character. In input mode, quotes the next character so that it is possible to insert non–printing and special characters into the file (12.4.2, 12.7.5).

CTRL W — Not a command character. During an insert, backs up as b would in command mode; the deleted characters remain on the display (see CTRL H) (12.7.5).

CTRL X — Unused.

CTRL Y — Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to CTRL U which scrolls up a bunch.)

CTRL Z — If supported by the system, stops *vi*, exiting to the top level shell. Same as :stop RETURN. Otherwise, unused.

CTRL [ — (ESCAPE) 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 ?); ends insertions of new text into the buffer. If an ESCAPE is given when quiescent in command state, *vi* rings the bell or flashes the screen. You can thus press ESCAPE if you don't know what is happening till *vi* rings the bell. If you don't know if you are in insert mode you can type ESCAPE a, and then material to be input; the material is inserted correctly whether or not you were in insert mode when you started (12.1.5, 12.3.1, 12.7.5).

CTRL \ — Unused.

CTRL ] — Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a RETURN. Mnemonically, this command is "go right to" (12.7.3).

CTRL ^ — Equivalent to :e #RETURN, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (12.7.3). (You have to do a :w before CTRL ^ will work in this case. If you do not wish to write the file, do :e! #RETURN instead.)

CTRL _ — Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

SPACE — Same as *right arrow* (see *l*).

! — 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 RETURN. 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*RETURN reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the com-

mand !*%grind*RETURN,* given at the beginning of a function, runs the text of the function through the LISP grinder (12.6.7, 12.7.3). To read a file or the output of a command into the buffer use :r (12.7.3). To simply execute a command use :! (12.7.3).

(Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.)

" Precedes a named buffer specification. There are named buffers 1–9 used for saving deleted text and named buffers a–z into which you can place text (12.4.3,12. 6.3)

# The macro character which, when followed by a number, substitutes for a function key on terminals without function keys (12.6.9). In input mode, if this is your erase character, it deletes 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.

$ Moves to the end of the current line. If you :se listRETURN, the end of each line is shown by printing a $ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus 2$ advances to the end of the following line.

% Moves to the parenthesis or brace { } which balances the parenthesis or brace at the current cursor position.

& A synonym for :&RETURN, by analogy with the *ex* & command.

' When followed by a ' 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, returns to the line which was marked with this letter with a m command, at the first non–white character in the line. (12.2.2, 12.5.3). 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.

( 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 ? which is followed by either the end of a line or by two spaces. Any number of closing ) ] " and ' characters may appear after the . ! or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[ below). A count advances that many sentences (12.4.2,12. 6.8).

) Advances to the beginning of a sentence. A count repeats the effect. See ( above for the definition of a sentence (12.4.2, 12.6.8).

* Unused.

+ Same as RETURN when used as a command.

, Reverse of the last f F t or T command, looking the other way in the current line. Especially useful after pressing 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 this is not possible. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center (12.2.3).

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then press . to delete more and more words/lines. Given a count, it passes it on to the

command being repeated. Thus after a 2dw, 3. deletes three words (12.3.3, 12.6.3, 12.7.2, 12.7.4).

Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an returns to command state without ever searching. The search begins when you press RETURN 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 then be terminated with a CTRL C, 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 a closing / and then an offset *+n* or *–n*.

To include the character / in the search string, you must escape it with a preceding \. A ^ 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 Section 12.7.4; unless you set nomagic in your *.exrc* file you have to preceed the characters . [ * and ~ in the search pattern with a \ to get them to work as you would naively expect (12.1.5, 12.2, 12.6.1, 12.7.2, 12.7.4).

0 Moves to the first character on the current line. Also used, in forming numbers, after an initial 1–9.

1–9 Used to form numeric arguments to commands (12.2.3, 12.7.2).

: 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 an RETURN, and the command then executed. You can return to where you were by pressing CTRL C if you press : accidentally (see primarily 12.6.2 and 12.7.3).

; Repeats the last single character find which used f F t or T. A count iterates the basic scan (12.4.1).

< An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus 3<< shifts three lines (12.6.6, 12.7.2).

= Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set (12.6.8).

> An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object (12.6.6, 12.7.2).

? Scans backwards, the opposite of /. See the / description above for details on scanning (12.2.2, 12.6.1, 12.7.4).

@ A macro character (12.6.9). 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 (12.3.1, 12.3.4, 12.7.5).

A Appends at the end of line, a synonym for $a (12.7.2).

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 (12.2.4).

C — Changes the rest of the text on the current line; a synonym for c$.

D — Deletes the rest of the text on the current line; a synonym for d$.

E — Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.

F — Finds a single following character, backwards in the current line. A count repeats this search that many times (12.4.1).

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 (12.7.2).

H — *Home arrow*. Homes the cursor to the top line on the screen. If a count is given, the cursor is moved to the count'th line on the screen. In any 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 (12.2.3, 12.3.2).

I — Inserts at the beginning of a line; a synonym for ^i.

J — Joins together lines, 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 two (12.6.5, 12.7.1f).

K — Unused.

L — Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with L (12.2.3).

M — Moves the cursor to the middle line on the screen, at the first non-white position on the line (12.2.3).

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 there up to an ESCAPE. 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 (12.3.1).

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. 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 (12.6.3).

Q — Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; *vi* supplies the : as a prompt (12.7.7).

R — Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESCAPE.

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.

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 (12.4.1).

U — Restores the current line to its state before you started changing it (12.3.5).

V — Unused.

W — Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect (12.2.4).

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 very useful synonym for yy. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (12.7.4).

ZZ — Exits *vi*. (Same as :x[RETURN].) If any changes have been made, the buffer is written out to the current file. Then *vi* quits.

[[ — Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed [CTRL]L. Lines beginning with { also stop [[; this makes it useful for looking backwards, 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. (12.4.2, 12.6.1, 12.6.6, 12.7.2).

\ — Unused.

]] — Forward to a section boundary, see [[ for a definition (12.4.2, 12.6.1, 12.6.6, 12.7.2).

^ — Moves to the first non-white position on the current line (12.4.4).

_ — 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 which was marked with this letter with a 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 (12.2.2,12. 5.3).

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 [ESCAPE] (12.3.1, 12.7.2).

b — Backs up to the beginning of a word. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (12.2.4).

c — An operator which changes the following object, replacing it with the following input text up to an [ESCAPE]. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed away is marked with a $. A count causes that many objects to be affected, thus both 3c) and c3) change the following three sentences (12.7.4).

d — An operator which 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 (12.3.3, 12.3.4, 12.4.1, 12.7.4).

e — Advances to the end of the next word, defined as for b and w. A count repeats the effect (12.2.4, 12.3.1).

f — Finds the first instance of the next character following the cursor on the current line. A count repeats the find (12.4.1).

g — Unused.

Arrow keys *h*, *j*, *k*, *l*, and *H*.

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 (CTRL H) has the same effect. A count repeats the effect (12.3.1, 12.7.5).

i — Inserts text before the cursor, otherwise like a (12.7.2).

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 CTRL *J*, LINE FEED, and CTRL *N*.

k — *Up arrow*. Moves the cursor one line up. CTRL *P* is a synonym.

l — *Right arrow*. Moves the cursor one character to the right. SPACE is a synonym.

m — Marks the current position of the cursor in the mark register which is specified by the next character a–z. Return to this position or use with an operator using ' or ' (12.5.3).

n — Repeats the last / or ? scanning commands (12.2.2).

o — Opens new lines below the current line; otherwise like O (12.3.1).

p — Puts text after/below the cursor; otherwise like P (12.6.3).

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 the more usually useful iteration of r (12.3.2).

s — Changes the single character under the cursor to the text which follows up to an ESCAPE; given a count, that many characters from the current line are changed. The last character to be changed is marked with $ as in c (12.3.2).

t — Advances the cursor upto the character before the next character typed. Most useful with operators such as d and c to delete the characters up to a following character. You can use . to delete more if this doesn't delete enough the first time (12.4.1).

u — Undoes the last change made to the current buffer. If repeated, alternates between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers (12.3.5).

v — Unused.

w — Advances to the beginning of the next word, as defined by b (12.2.4).

x — Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (12.6.5).

y — An operator, 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 (12.7.4).

z — Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . the center of the screen, and – at 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. (12.5.4)

{ Retreats to the beginning of 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 [[ above) (12.4.2, 12.6.8, 12.7.6).

| Places the cursor on the character in the column specified by the count (12.7.1, 12.7.2).

} Advances to the beginning of the next paragraph. See { for the definition of paragraph (12.4.2, 12.6.8, 12.7.6).

~ Unused.

CTRL? (CTRL C) Interrupts *vi*, returning it to command accepting state (12.1.5, 12.7.5)

# CHAPTER 13
# EDIT TUTORIAL

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor described here is named *edit*. Creating text using edit is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they were entered by you. Another program, a text formatter, rearranges your text for you into "finished form." This chapter does not discuss the use of a text formatter.

These lessons assume no prior familiarity with with text editing. They consist of a series of text editing sessions that lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, practice the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, you will soon become familiar with using the computer to write and modify text.

## 13.1 SESSION 1

### 13.1.1 Invoking Edit

To begin your editing session, type *edit* followed by a space and then the filename you have selected; for example, "text". Now is a convenient time to choose a name for the file of text you are about to create. When you have completed the command, press the RETURN key and wait for edit's response:

```
% edit text
"text" No such file or directory :
```

If you typed the command correctly, you are now in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. It was

unable to find such a file, since "text" is a new file you are about to create. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that you are in *command mode* and edit expects a command from you.

## 13.1.2 Entering Text

You can now begin entering text into the buffer. This is done by *appending* (or adding) text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, since you are adding text to nothing you are creating text.

Most edit commands have two forms: a word that suggests what the command does, and a shorter abbreviation of that word. Either form can be used. Many beginners find the full command names easier to remember at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append", and it can be abbreviated "a". Type "append" and press the RETURN key.

```
% edit text
:append
```

## 13.1.3 Messages from Edit

If you make a mistake in entering a command and type something that edit does not recognize, edit responds with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you receive this message:

```
:add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that edit is again ready to execute a command.

## 13.1.4 Text Input Mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode*, also known as *append mode*. When you enter text input mode, edit stops sending you a prompt. You do not receive any prompts or error messages while in text input mode. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You can append as much text as you want, and *when you wish to stop entering text lines, type a period as the only character on the line and press the RETURN key*. When you type the period and press RETURN, you signal that you want to stop appending text, and edit responds by allowing you to exit text input mode and reenter command mode. Edit again prompts you for a command by printing ":".

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit believes you want to remain in append mode and does not let you out. As this can be very frustrating, be sure to type *only* the period and the RETURN key.

This is a good place to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you remain in append mode with the last line of text being:

Suppose that the lines of text you enter are (try to type *exactly* what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
```

The last line is the period followed by a RETURN that gets you out of append mode.

## 13.1.5 Making Corrections

You may recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase.

The usual erase character is the backspace (control-H), and you can correct typing errors in the line you are typing by holding down the CTRL key and typing the "H" key. If you type control-H, the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

If you make a bad start in a line and would like to begin again, you can either backspace to the beginning of the line or you can use the at-sign "@" to erase everything on the line:

```
Text edtiing is strange, but@
Text editing is strange, but nice.
```

When you type the at-sign (@), you erase the entire line typed so far and are given a fresh line to type on. You can immediately begin to retype the line. This, unfortunately, does not help after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next session and those that follow.

## 13.1.6 Writing Text to Disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and lasts only until the end of the editing session. Learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:write
```

Edit copies the contents of the buffer to a disk file. If the file does not yet exist, a new file is created automatically and the presence of a "[New file]" is noted. The newly-created file is given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit repeats the filename and gives the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines that were written to disk are still in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit prints

```
No current filename
```

in response to your write command. If this happens, you can specify the filename in a new write command:

```
:write text
```

After the "write" (or "w"), type a space and then the name of the file.

### 13.1.7 Signing Off

You have done enough for this first lesson and are ready to quit the session with edit. To do this, type "quit" (or "q") and press RETURN:

```
:write
"text" [New file] 3 lines, 90 characters
:quit
%
```

The % is from the shell to tell you that your session with edit is over and you can enter further commands.

## 13.2 SESSION 2

You can specify the name of the file you worked on last time. This starts edit working, and it fetches the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it repeats its name and tells you the number of lines and characters it contains. Thus,

```
% edit text
"text" 3 lines, 90 characters
:
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions, and indicates this by its prompt character, the colon (:). In this session, you append more text to your file, print the contents of the buffer, and learn to change the text of a line.

### 13.2.1 Adding More Text to the File

If you want to add more to the end of your text you can do so by using the append command to enter text input mode. When "append" is the first command of your editing session, the lines you enter are placed at the end of the buffer. Use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

You may recall that once you enter append mode using the "a" (or "append") command, you need to type a line containing only a period (.) to exit append mode.

### 13.2.2 Interrupt

If you press the RUB key (sometimes labelled DELETE) while working with edit, it sends this message to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by rub or delete, causing edit to prompt you for a new command. If you are appending text at the time, you exit from append mode and are expected to give another command. The line of text you were typing when the append command was interrupted is not entered into the buffer.

### 13.2.3 Making Corrections

If while typing the line you hit an incorrect key, recall that you can delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer to the last few pages of Session 1 if you need to review the procedures for making a correction. The most important idea to remember is that erasing a character or cancelling a line must be done before you press the RETURN key.

### 13.2.4 Listing What's in the Buffer

Having appended text to what you wrote in Session 1, you might want to see all the lines in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The "1" (not to be confused with the letter "el") stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and "p" (or print) is the command to print from line 1 to the end of the buffer. The command "1,$p" gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently pressing the CTRL key while typing "a". This can happen on many terminals because the CTRL key and the "A" key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

```
it does illustr^Ate the editor.
```

To represent the control-A, edit shows "^A". The sequence "^" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter that appears after the "^". You will soon see the commands that can be used to correct this typing error.

In looking over the text, you can see that "this" is typed as "thiss" in the second line, a deliberate error so that you can learn to make corrections. You will correct the spelling.

## 13.2.5 Finding Things in the Buffer

In order to change something in the buffer, you first need to find it. You can find "thiss" in the text you have entered by looking at a listing of the lines. Physically speaking, you search the lines of text looking for "thiss" and stop searching when you have found it. The way to tell edit to search for something is to type it inside slash marks:

```
:/thiss/
```

By typing */thiss/* and pressing RETURN, you instruct edit to search for "thiss". If you ask edit to look for a pattern of characters that it cannot find in the buffer, it responds "Pattern not found". When edit finds the characters "thiss", it prints the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line it just printed, ready to make a change in the line.

## 13.2.6 The Current Line

Edit keeps track of the line in the buffer where it is located at all times during an editing session. In general, the line that has been most recently printed, entered, or changed is the current location in the buffer. The editor is prepared to make changes at the current location in the buffer, unless you direct it to another location.

In particular, when you bring a file into the buffer, you are located at the last line in the file, where the editor left off copying the lines from the file to the buffer. If your first

editing command is "append", the lines you enter are added to the end of the file, after the current line - the last line in the file.

You can refer to your current location in the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage return you instruct edit to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type .= and press RETURN, and edit responds with the line number:

```
:.=
2
```

If you type the number of any line and press RETURN, edit positions you at that line and print its contents:

```
:2
And thiss is some more text.
```

Experiment with these commands to gain experience in using them to make changes.

## 13.2.7 Numbering Lines

The *number* (*nu*) command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
          2    And thiss is some more text.
```

Note that the shortest abbreviation for the number command is "nu" (and not "n", which is used for a different command). You can specify a range of lines to be listed by the number command in the same way that lines are specified for

print. For example, 1,$nu lists all lines in the buffer with their corresponding line numbers.

## 13.2.8 Substitute Command

Now that you have found the misspelled word, you can change it from "thiss" to "this". As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append*, so *s* stands for *substitute*. Use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command instructs edit to make the change:

```
2s/thiss/this/
```

You first indicate the line to be changed, line 2, and then type an "s" to indicate you want edit to make a substitution. Inside the first set of slashes are the characters that you want to change, followed by the characters to replace them, and then a closing slash mark. To summarize:

```
2s/ what is to be changed / what to change it to /
```

If edit finds an exact match of the characters to be changed, it makes the change *only* in the first occurrence of the characters. If it does not find the characters to be changed, it responds:

```
Substitute pattern match failed
```

indicating that your instructions could not be carried out. When edit does find the characters that you want to change, it makes the substitution and automatically prints the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/
And this is some more text.
```

line 2 (and line 2 only) are searched for the characters "thiss", and when the first exact match is found, "thiss" is changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

```
:s/thiss/this/
```

edit assumes that you mean to change the line where you are currently located ("."). In this case, the command without a line number would have produced the same result because you were already located at the line you wished to change.

For another illustration of the substitute command, choose the line:

```
Text editing is strange, but nice.
```

You can make this line a bit more positive by taking out the characters "strange, but " so the line reads:

```
Text editing is nice.
```

A command that first positions edit at the desired line and then makes the substitution is:

```
:/strange/s/strange, but //
```

What you have done here is combine your search with your substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you can identify the line you want to change by asking edit to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

| | |
|---|---|
| `/strange/` | tells edit to find the characters "strange" in the text |
| `s` | tells edit to make a substitution |
| `/strange, but //` | substitutes nothing at all for the characters "strange, but " |

Note the space after "but" in "/strange, but /". If you do not indicate that the space is to be taken out, your line will read:

```
Text editing is  nice.
```

which looks a little funny because of the extra space between "is" and "nice". Again, a blank space is a real character to a computer, and in editing text you need to be aware of spaces within a line just as you would be aware of an "a" or a "4".

### 13.2.9 Another Way to List What's in the Buffer

Although the print command is useful for looking at specific lines in the buffer, other commands may be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command *z*. If you type

```
:1z
```

edit starts with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command

```
:z
```

If no starting line number is given for the z command, printing starts at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as *paging*. Paging can also be used to print a section of text on a hard-copy terminal.

### 13.2.10 Saving the Modified Text

This seems to be a good place to end the second session. If you (in haste) type "q" to quit the session your dialogue with edit is:

```
:q
No write since last change (:quit! overrides)
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you did during the editing session since you typed the latest write command. Because in this lesson you have not written to disk at all, everything you have done would have been lost if edit had obeyed the q command. If you did not want to save the work done during this editing session, you would have to type "q!" or ("quit!") to confirm that you indeed wanted to end the session immediately, leaving the file as it was after the most recent "write" command. However, since you want to save what you have edited, you need to type:

```
:w
"text" 6 lines, 171 characters
```

and then follow with the commands to quit and logout:

## 13.3 SESSION 3

### 13.3.1 Bringing Text into the Buffer

Try to make contact with edit without looking at the notes. Did you remember to give the name of the file you wanted to edit? That is, did you type

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way brings a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by typing:

```
:e text
"text" 6 lines, 171 characters
```

The command *edit*, which can be abbreviated e, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You can also use the edit (e) command to change files in the middle of an editing session, or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

## 13.3.2 Moving Text in the Buffer

Edit allows you to move lines of text from one location in the buffer to another by means of the *move* (m) command. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

```
:2,4m$
```

directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. So,

```
:1,3m6
```

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be "4m5".

Move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then use the print (p), z, or number (nu) command to view more text. The buffer now contains:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

You can restore the original order by typing:

```
:4,$m1
```

or, combining context searching and the move command:

```
:/And this is some/,/This is text/m/This is some sample/
```

(Do not type both examples here!) The problem with combining context searching with the move command is that your chance of making a typing error in such a long command is greater than if you type line numbers.

## 13.3.3 Copying Lines

The *copy* command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:2,5copy $
```

makes a copy of lines 2 through 5, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is co (and not the letter "c", which has another meaning).

## 13.3.4 Deleting Lines

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by delete or d. This example deletes line 4, which is "This is text added in Session 2." if you typed the commands suggested so far.

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted, and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line that has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the line containing the indicated text, starting its search at the current line and moving line by line until it finds the text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line ("."), that is, the line found by your search. After the deletion, your buffer contains:

```
This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
It doesn't mean much here, but
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
2 lines deleted
```

which specifies the range of lines from 2 to 3, and the operation on those lines: "d" for delete. If you delete more than one line, you receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command:

```
:/And this is some/,/Text editing is nice./d
```

## 13.3.5 A Word or Two of Caution

In using the search function to locate lines to be deleted, be *absolutely sure* the characters you give as the basis for the search take edit to the line you want deleted. Edit searches for the first occurrence of the characters starting from where you last edited – that is, from the line you see printed if you type dot (.).

A search based on too few characters can result in the wrong lines being deleted, which edit does as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing RETURN to send the command on its way.

## 13.3.6 Undo to the Rescue

The *undo* (*u*) command has the ability to reverse the effects of the last command that changed the buffer. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give.

It is possible to undo only commands that have the power to change the buffer – for example, delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e), which interact with disk files, cannot be undone, nor can com-

mands that do not change the buffer, such as print. Most importantly, the *only* command that can be reversed by undo is the last "undo-able"

command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

To illustrate, issue an undo command. Recall that the last buffer-changing command you gave deleted the lines formerly numbered 2 and 3. Typing undo at this moment reverses the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line that is now "dot" (the current line).

## 13.3.7 More About the Dot and Buffer End

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode; type dot (and only a dot) on a line and press RETURN;
2. to refer to the line you are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

If you type ".=" you are asking for the number of the line, and if you type "." you are asking for the text of the line.

In this editing session and the last, you used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=) edit prints the line number corresponding to the last line in the buffer.

"." and "$", then, represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,$d
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

## 13.3.8 Moving Around in the Buffer

When you are editing you often want to go back and reread a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few, say 3, lines ago, you can type

```
-3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line that is 2 ahead of your current position.

You can use "+" and "-" in any command where edit accepts line numbers. Line numbers specified with "+" or "-" can be combined to print a range of lines. The command

```
:-1,+2copy$
```

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines are placed after the last line in the buffer ($), and the original lines referred to by "-1" and "+2" remain where they are.

Try typing only "-"; you move back one line just as if you had typed "-1p". Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see edit's response. Typing RETURN alone on a line is the equivalent of typing "+1p"; it moves you one line ahead in the buffer and prints that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, edit reminds you that you are at the end of the buffer:

```
Atend-of-file
```

or

```
Not that many lines in buffer
```

Similarly, if you try to move to a position before the first line, edit prints one of these messages:

```
Nonzero address required on this command
```

or

```
Negative address - first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

## 13.3.9 Changing Lines

You can also delete certain lines and insert new text in their place. This can be accomplished easily with the *change* (*c*) command. The change command instructs edit to delete specified lines and then switches to text input mode to accept the text that replaces them. Suppose that you want to change the first two lines in the buffer:

```
This is some sample text.
And this is some more text.
```

to read

```
This text was created with the text editor.
```

To do so, you type:

```
:1,2c
2 lines changed
This text was created with the text editor.
.
:
```

In the command 1,2c you specify that you want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines are deleted. After you type RETURN to end the change command, edit notifies you if more than one line is changed and places you in text input mode. Any text typed on the following lines is inserted into the position where lines were deleted by the change command. *You remain in text input mode until you exit in the usual way, by typing a period alone on a line*. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

## 13.4 SESSION 4

This lesson covers several topics, starting with commands that apply throughout the buffer, characters with special meanings, and how to issue commands to the shell while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

### 13.4.1 Making Commands Global

One disadvantage to the commands you have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer – the *global* (*g*) command.

To print all lines containing a certain sequence of characters (say, "text") the command is:

```
:g/text/p
```

The "g" instructs edit to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed for the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the "g" at the end of the global command, which instructs edit to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" *in each line* is changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. You can give a command such as:

```
:5s/text/material/g
```

to change every instance of "text" in line 5 alone. Further, neither command changes "text" to "material" if "Text" begins with a capital rather than a lower-case *t*.

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

Be careful about using the global command in combination with any other – in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

deletes every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit prints a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. Experiment with the global command on a small file of text to see what it can do for you.

## 13.4.2 More About Searching and Substituting

In using slashes to identify a character string that you want to search for or change, you have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "text" to "texts" you can type either

```
:/text/s/text/texts/
```

as you have done in the past, or a somewhat abbreviated command:

```
:/text/s//texts/
```

In this example, the characters to be changed are not specified – there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters you last searched for as the characters to be changed."

Similarly, the last context search can be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

(Note that the search command found the characters "does" in the word "doesn't" in the first search request.) Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute com-

mand, using the same search and replacement patterns. After altering the current line by typing

```
:s/text/texts/
```

you type

```
:/text/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters "text".

### 13.4.3 Special Characters

Two characters have special meanings when used in specifying searches: "$" and "^". "$" is taken by the editor to mean "end of the line" and is used to identify strings that occur at the end of a line.

```
:g/text.$/s//material./p
```

tells the editor to search for all lines ending in "text." (and nothing else, not even a blank space), to change each final "text." to "material.", and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to lose temporarily its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search rather than the character "$". The backslash retains its special significance unless it is preceded by another backslash.

### 13.4.4 Issuing Commands to the Shell

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes commands). You do not need to quit the editor to execute a shell command as long as you indicate that it is to be sent to the shell for execution. For example, to use the shell command *rm* to remove the file named "junk" type:

```
:!rm junk
!
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a shell command. If the buffer contents have not been written since the last change, a warning is printed before the command is executed:

```
[No write since last change]
```

The editor prints a "!" when the command is completed.

## 13.4.5 Filenames and File Manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename*. Edit remembers as the current filename the name given when you entered the editor. The current filename changes when ever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as you have seen, supplies the current filename. If you are editing a file named "draft3" having 283 lines in it, you can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" [new file] 283 lines, 8698 characters
```

The current filename remembered by the editor *is not changed as a result of the write command*. Thus, if the next write command does not specify a name, edit writes onto the current file ("draft3") and not onto the file "chapter3".

## 13.4.6 The File Command

To ask for the current filename, type *file* (or *f*). In response, the editor provides current information about the buffer, including the filename, your current position, the number of lines in the buffer, and the percent of the distance through the file your current location is.

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor tells you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer is longer considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

## 13.4.7 Reading Additional Files

The read (r) command allows you to add the contents of a file to the buffer at a specified location, essentially copying new lines between two existing lines. To use it,

specify the line after which the new text is to be placed, the read (r) command, and then the name of the file. If you have a file named "example", the command

```
:$r example
"example" 18 lines, 473 characters
```

reads the file "example" and adds it to the buffer after the last line. The current filename is not changed by the read command.

## 13.4.8 Writing Parts of the Buffer

The *write* (*w*) command can write all or part of the buffer to a file you specify. You are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending*. The lines remain in the buffer as part of the document you are editing, and you can continue to edit the entire buffer. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

## 13.4.9 Recovering Files

Although it does not happen very often, there are times when the system stops working because of some malfunction. This situation is known as a *crash*. Under most circumstances, edit's crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, edit sends you mail that gives the name of the recovered file. To recover the file, enter the editor and type the command *recover* (rec), followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file "chap6", the command is:

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command rm.

## 13.4.10 Other Recovery Techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command *preserve* (pre), which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case, use the shell escape from the editor (!) to remove some files

you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

and wait for the reply,

```
File preserved.
```

If you do not receive this reply, seek help immediately. Do not simply leave the editor. If you do, the buffer is lost, and you may not be able to save your file. If the reply is "File preserved." you can leave the editor (or logout) to remedy the situation. After a preserve, you can use the recover command once the problem has been corrected, or the –r option of the edit command if you leave the editor and want to return.

If you make an undesirable change to the buffer and type a write command before discovering your mistake, the modified version replaces any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

## 13.4.11 Further Reading and Other Information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its more commonly-used commands. You have not covered all of the editor's commands, but a selection of commands that should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in Chapter 14, which is applicable to both *ex* and *edit*. One way to become familiar with it is to begin by reading the description of commands that you already know.

## 13.4.12 Using Ex

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name *ex* in your command instead of *edit*.

Edit commands work the same way in ex, but the editing environment is somewhat different. A few differences exist between the two versions of the editor. In edit, only the characters "^", "$", and "\" have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in Chapter 14.

Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual*, in which the editor behaves quite differently from normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing "o". Type the ESC key and then a "Q" to get out of open or visual mode and back into the regular editor command mode. Chapter 12 provides a full discussion of visual mode.

# CHAPTER 14

# EX REFERENCE

**Version 3.5**

*Ex* a line oriented text editor that supports both command and display oriented editing. This chapter describes the command oriented part of *ex*; the display editing features of *ex* are described in Chapter 12, *Display Editing with Vi*. An introduction to *ex* is provided in Chapter 13, *Edit Tutorial*.

## 14.1 INVOKING EX

Each instance of the editor has a set of options that can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description that follows we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) the editor seeks 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 executes 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* are executed before each editor session.

A command to invoke *ex* has the form:

```
ex [option...] name...
```

The following options can be used:

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

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

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

-r 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.

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

-w*n* sets the default window size to *n*, and is useful on dialups to start in small windows.

-x causes *ex* to prompt for a *key* that is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt*(1).

-R sets the *readonly* option at the start.

+*command*
indicates that the editor should begin by executing the specified command. If *command* is omitted, 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.

*name* indicates a file to be edited.

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

```
ex name
```

## 14.2 FILE MANIPULATION

### 14.2.1 Current File

*Ex* is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually 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 unless and 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. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited*. This means that the contents of the buffer are logically connected with the current file name, 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, ex* does not normally write on it if it already exists (the *file* command says "[Not edited]" if the current file is not considered edited).

### 14.2.2 Alternate File

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

### 14.2.3 Filename Expansion

Filenames within the editor can be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name. (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 is received.)

### 14.2.4 Multiple Files and Named Buffers

If more than one file is given on the command line, 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 can be displayed with the *args* command. The next file in the argument list can be edited with the *next* command. The argument list can also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes 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 upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.)

### 14.2.5 Read Only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidently 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, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. It is possible to write, even while in read only mode, by indicating that you really know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

## 14.3 EXCEPTIONAL CONDITIONS

### 14.3.1 Errors and Interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing terminates. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, *ex* exits when this occurs.

### 14.3.2 Recovering from Hangups and Crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) attempts 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 hangup or editor crash. To recover a file you can use the *-r* option. If you were editing the file *resume*, 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 indeed ok, you can *write* it over the previous contents of that file.

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

```
ex -r
```

prints a list of the files that have been saved for you. (In the case of a hangup, the file does not appear in the list, although it can be recovered.)

## 14.4 EDITING MODES

*Ex* has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. 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 when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *Display Editing with Vi*.

## 14.5 COMMAND STRUCTURE

Most command names are English words, and 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 command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

### 14.5.1 Command Parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses are discussed below. A number of commands also can take a trailing *count* specifying the number of lines to be involved in the command. Thus the command "10p" prints the tenth line in the buffer while "delete 5" deletes five lines from the buffer, starting with the current line. Counts are rounded down if necessary.

Some commands take other information or parameters, this information always being given after the command name. Examples would be option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

### 14.5.2 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 can be controlled by options; in this case, the '!' serves to toggle the default.

### 14.5.3 Flags After Commands

The characters '#', 'p' and 'l' can 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 can 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.

### 14.5.4 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 " can also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

### 14.5.5 Multiple Commands Per Line

More than one command can 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 '|'.

### 14.5.6 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 can be quickly and easily reversed with an *undo*. After commands with more global effect such as *global* or *visual*, you are informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

## 14.6 COMMAND ADDRESSING

### 14.6.1 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's buffer, lines being numbered sequentially from 1. |
| $ | The last line in the buffer. |
| % | An abbreviation for "1,$", the entire |
| +*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*, the trailing / or ? can be omitted. If *pat* is omitted or explicitly empty, 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 can also be established by the *mark* command, using single lower case letters *x* and the marked lines referred to as ''*x*'. |

### 14.6.2 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, 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.)

## 14.7 COMMAND DESCRIPTIONS

The following form is a prototype for all *ex* command descriptions:

*address* command ! *parameters count flags*

All parts are optional; the degenerate case is the empty command, which prints the next line in the file. For use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions:

- Default addresses shown in parentheses are *not* part of the command.
- Abbreviated forms are shown in bold. For example, the abbreviated form of "append" is "**a**".

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

( . ) **a**ppend
*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 ']'.

( . , . ) **c**hange *count*
*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*.

( . , . ) **co**py *addr flags*
A *copy* of the specified lines is placed after *addr*, which can be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy*.

( . , . ) **d**elete *buffer count flags*
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 upper case letter is used.

**e**dit *file*
ex *file*
Used to begin an editing session on a new file. The editor first checks to see if the 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 cur-

rent file and prints the new filename. After insuring that this file is sensible (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) 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 last line of the input file is missing the trailing newline character, it is supplied and a complaint is issued. This command leaves the current line '.' at the last line read. (If executed from within *open* or *visual*, 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 that 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* can also be an editor command containing no spaces, e.g.: "+/pat".

file Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and 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 will not destroy a file unrelated to the current contents of the buffer.

file *file*
The current file name is changed to *file* which is considered '[Not edited]'.

( 1 , $ ) **g**lobal */pat/ cmds*
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 can 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. *Append*, *insert*, and *change* commands and associated input are permitted; the '.' terminating input can be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

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

g! */pat/ cmds* (abbreviation: **v**)
The variant form of *global* runs *cmds* at each line not matching *pat*.

( . )insert
*text*
.
Places the given text before the specified line. The current line is left at the last line input; if there were none 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*
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 was 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 is discarded.

j!
The variant causes a simpler *join* with no white space processing; the 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. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it is as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See *Display Editing with Vi* for more details.

( . ) mark *x*
Gives the specified line mark *x*, a single lower case 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*
The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next
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), it is executed after editing the first such file.

( . , . ) number *count flags* (alternate abbreviation: #)
Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

( . ) open *flags*
( . ) open */pat/ flags*
Enters intraline editing *open* mode at each addressed line. If *pat* is given, the cursor is placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *Display Editing with Vi* for more details.

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

( . , . ) print *count* (alternate abbreviation: **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*
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, the last *deleted* or *yanked* text is restored. By using a named buffer, text can be restored that was saved there at any previous time.

> *Note: No modifying commands can intervene between the delete or yank and the put, nor can lines be moved between files without using a named buffer.*

quit Causes *ex* 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*. (*Ex* also issues a diagnostic if there are more files in the argument list.) Normally, you wish to save your changes, so 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*
Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name, *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 *command* into the buffer after the specified line. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename*; a blank or tab before the ! is mandatory.

recover *file*
Recovers *file* from the system save area. Used after a accidental hangup of the phone or a system crash (the system saves a copy of the file you were editing only if you have made changes to the file) or *preserve* command. Except when you use *preserve* you are notified by mail when a file is saved.

rewind 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 of the 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 no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

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

shell A new shell is created. When it terminates, editing

source *file*
Reads and executes commands from the specified file. *Source* commands can be nested.

stop Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form *stop*! is used. This commands is only available where supported by the teletype driver and operating system.

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

Lines can be split by substituting new–line 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 flag*
If *pat* and *repl* are omitted, 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* reuses the previous tag.)

The tags 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.

The tag names in the tags file must be sorted alphabetically.

**un**abbreviate *word*
Delete *word* from the list of abbreviations.

undo Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of

*undo* (as are *open* and *visual*.) Also, the commands *write* and *edit*, which interact with the file system, cannot be undone. *Undo* is its own inverse.

*Undo* always marks the previous value of the current line '.' as '"'. After an *undo* 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 it's 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 Prints the current version number of the editor as well as the date the editor was last changed.

( . ) visual *type count flags*
Enters visual mode at the specified line. *Type* is optional and can 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 the document *Display Editing with Vi* for more details. To exit this mode, type Q.

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

( 1 , $ ) write *file*
Writes 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, text is written to that file. If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never changed.

The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty*, */dev/null*. 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* (abbreviation: w>>)
Writes the buffer contents at the end of an existing file.

Overrides the checking of the normal *write* command, and writes to any file that the system permits.

( 1 , $ ) w !*command*
Writes the specified lines into *command*. Note the difference between w!, which overrides checks, and w ! which writes to a command.

wq *name*
Like a *write* and then 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, in any case, quits.

( . , . ) yank *buffer count*
Places the specified lines in the named *buffer*, for later retrieval via *put*. If no buffer name is specified, the lines go to a more volatile place; 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. 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.

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 '–' can be repeated for cumulative effect.

! *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 is 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, a diagnostic is printed before the command is executed as a warning. 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*
Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non–white characters are discarded in a left–shift. The current line becomes the last line that changed due to the shifting.

^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.

## 14.8 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. *Ex* 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, e.g. '//' or '??'.*

### 14.8.1 Magic and Nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* 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*, the default for *edit*, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic*. To discern what is true with *nomagic* it suffices to remember that the only special characters in this case are '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

The following basic constructs are used to construct *magic* mode regular expressions.

*char* An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.

↑ 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.

\< Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character that is neither a letter, nor a digit, nor the underline character.

[*string*] Matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by '–' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a–z]' as a regular expression matches any (single) lower–case letter. If the first character of *string* is an '↑' the construct matches those characters that it otherwise would not; thus '[↑a–z]' matches anything but a

To place any of the characters '↑', '[', or '–' in *string* you must escape them with a preceding '\'.

### 14.8.2 Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string that can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above can be followed by the character '*' to form a regular expression that matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' can be used in a regular expression, and matches the text that defined the replacement part of the last *substitute* command. A regular expression can be enclosed between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

### 14.8.3 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 escaping character '\'. 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 sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper– or lower–case 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.

## 14.9 OPTION DESCRIPTIONS

autoindent, ai (default: noai)

Can be used to ease the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line starts aligned with the first non–white character of the previous line. To back the cursor up to the preceding tab stop one can press ^D. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end–of–file with a ^D.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a ^D. This causes the input to be repositioned

at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a ^D repositions at the beginning but without retaining the previous indent.

*Autoindent* doesn't happen 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. *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*, *stop*, *tag*, or ! command, or a ^↑ (switch files) or ^] (tag goto) command in *visual*. Note, that

case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind*! for .I rewind , *stop*! for *stop*, *tag*! for *tag*, *shell* for !, and :e # and a :ta! command from within *visual*).

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. *Beautify* does not apply to command input.

directory, dir (defaultdir=/tmp)
Specifies the directory in which *ex* places its buffer file. If this directory in not writable, the editor exits abruptly when it fails to be able to create its buffer there.

edcompatible (default: noedcompatible)
Causes the presence of 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 be as in the ~ command, instead of like &.

errorbells, eb (default: noeb)
Error messages are preceded by a bell. (Bell ringing in *open* and *visual* 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).

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

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

list (default: nolist)
All printed lines are displayed (more) unambiguously, showing tabs and end–of–lines as in the *list* command.

magic (default: magic for *ex* and *vi*, nomagic for *edit*)
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 can be made *magic* when *nomagic* is set by preceding them with a '\'.

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

number, nu (default: nonumber)
Causes all output lines to be printed with their line numbers. In addition each input line is prompted for by supplying its line number.

open (default: open)
If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt (default: optimize)
Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para (default: para=IPLPPPQPP LIbp)
Specifies the paragraphs for the { and } operations in *open* and *visual*. The pairs of characters 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 ':'.

redraw (default: noredraw)
The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

remap (default: remap)
If on, macros are repeatedly tried until they are unchanged. For example, if *o* is mapped to *O*, and *O* is mapped to *I*, then if *remap* is set, *o* maps to *I*, but if *noremap* is set, it maps to *O*.

report (default: report=5, 2 for *edit*)
Specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines, provide feedback as to 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=1/2 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*. The pairs of characters in the options's value are the names of the macros that start paragraphs.

shell, sh (default: sh=/bin/sh)
Gives the path name 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 a software tab stop, used in reverse tabbing with ^D when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm
In *open* and *visual* mode, when a ) or } is typed, move 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. See *Display Editing with Vi* for more details.

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. 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 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, 29600
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 using the regular expressions in addressing 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. See *An Introduction to Text Editing with Vi* for details.

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

## 14.10 LIMITATIONS

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

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

# CHAPTER 15
# INTRODUCTION TO ED

*Ed* is a text editor, that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, a program, or perhaps data for a program.

This introduction is meant to simplify learning *ed*. Read this document, simultaneously using *ed* to do the exercises, then read the description of *ed* in the *Commands and Applications Manual*, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.) Do the exercises! They cover material not completely discussed in the actual text.

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try Chapter 16.

It is assumed that you know how to log in, and that you have some understanding of what a file is. You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, this character, whatever it is, is referred to as RETURN.

## 15.1 GETTING STARTED

Once you have logged in to your system and it has printed the prompt character, usually a $ or a %, type

```
ed
```

You are now ready to go – *ed* is waiting for you to tell it what to do.

## 15.2 CREATING TEXT

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it has to start somewhere, and undergo modifications later. This section shows how to get some text in, just to get started. Later, it talks about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed*; it is usually done by typing in the text, or by reading it into *ed* from a file.

First 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 you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which you write things, then change some of them, and finally file the whole thing away for another day.

You tell *ed* what to do to your text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – these are discussed shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

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

```
a
```

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an *a* followed by a RETURN, followed by the lines of text you want, like this:

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

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer contains the three lines

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

The "*a*" and "." aren't there, because they are not text.

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

## 15.3 ERROR MESSAGES

If at any time you make an error in the commands you type to *ed*, it tells you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

## 15.4 WRITING TEXT OUT AS A FILE

It's likely that you'll 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 you want to write on. This copies the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named *junk*, for example, type

```
w junk
```

Leave a space between *w* and the file name. *Ed* responds 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 you can go on adding lines to it. This is an important point. *Ed* at all times 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, since if the system crashes or if you make some horrible mistake, you lose all the text in the buffer, but any text that was written onto a file is relatively safe.)

## 15.5 LEAVING ED

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the *w* command, and then type the command

```
q
```

which stands for *quit*. The system responds with the prompt character ($ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. (Actually, *ed* prints ? if you try to quit without writing. At that point, write if you want; if not, another *q* gets you out regardless.)

## 15.6 EXERCISE 1

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using *w*. Then leave *ed* with the *q* command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.)

## 15.7 READING TEXT FROM A FILE – THE EDIT COMMAND

A common way to get text into the buffer is to read it from a 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* fetches the entire contents of a file into the buffer. So if you had saved the three lines "Now is the time", etc., with a *w* command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file *junk* into the buffer, and respond

```
68
```

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

If you use the *e* command to read a file into the buffer, then you need not use a file name after a subsequent *w* command; *ed* remembers the last file name used in an *e* command, and *w* writes on this file. Thus a good way to operate is

```
ed
e file
[editing session]
w
q
```

This way, you can simply say *w* from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

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

```
f
```

*ed* would reply

```
junk
```

## 15.8 READING TEXT FROM A FILE - THE READ COMMAND

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command *r*. The command

```
r junk
```

reads the file *junk* into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
r junk
```

the buffer contains *two* copies of the text (six lines).

```
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*.

## 15.9 EXERCISE 2

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 doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. 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?

## 15.10 PRINTING THE CONTENTS OF THE BUFFER

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter *p*. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

```
1,2p        (starting line=1, ending line=2 p)
```

*Ed* responds with

```
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use *1,3p* as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" – the dollar sign $. Use it this way:

```
1,$p
```

This prints *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* types

```
?
```

and wait for the next command.

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

```
$,$p
```

but *ed* lets you abbreviate this to

```
$p
```

You can print any single line 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*. So if you say

```
$
```

*ed* prints the last line of the buffer.

You can also use $ in combinations like

```
$-1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

## 15.11 EXERCISE 3

As before, create some text using the *a* command and experiment with the *p* command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

```
3,1p
```

don't work.

## 15.12 THE CURRENT LINE

Suppose your buffer still contains the six lines as above, that you have just typed

```
1,3p
```

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

```
p
```  (no line numbers)

This prints

```
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 you have done anything with. (You just printed it!) You can repeat this *p* command without line numbers, and it continues to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

(pronounced "dot").

Dot is a line number in the same way that $ is; it means exactly "the current line", or loosely, "the line you most recently did something to." You can use it in several ways – one possibility is to say

```
.,$p
```

This prints all the lines from (including) the current line to the end of the buffer. In the example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The *p* command sets dot to the number of the last line printed; the last command sets both . and $ to 6.

Dot is most useful when used in combinations like this one:

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

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

```
.-1
```  (or .-1p )

which means "print the line *before* the current line." This enables you to go backwards if you wish. Another useful one is something like

```
.-3,.-1p
```

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

```
.=
```

*Ed* responds by printing the value of dot.

Let's summarize some things about the *p* command and dot. Essentially *p* can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter *p*), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return causes printing of the next line – it's equivalent to *.+1p*. Try it. Try typing a –; you will find that it's equivalent to *.–1p*.

## 15.13 DELETING LINES

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

```
d
```

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

*starting line, ending line* d

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left, as you can check 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 $.

## 15.14 EXERCISE 4

Experiment with *a*, *e*, *r*, *w*, *p* and *d* until you are sure that you know what they do, and until you understand how dot, $, and line numbers are used.

If you are adventurous, try using line numbers with *a*, *r* and *w* as well. You will find that *a* appends lines *after* the line number that you specify (rather than after dot); that *r* reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that *w* writes out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

```
0r filename
```

and you can enter lines at the beginning of the buffer by saying

```
0a
. . . text . . .
.
```

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

```
. w
```

## 15.15 MODIFYING TEXT

You are now ready to try one of the most important of all commands – the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

– the *e* has been left off *the*. You can use *s* to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* does not print the result automatically) say

```
p
```

and get

```
Now is the time
```

which is what you wanted. 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 you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for *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 ? as a warning.)

Thus you can say

```
1,$s/speling/spelling/
```

and correct the first spelling mistake 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 that you 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, to make sure it worked out right. If it didn't, 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 command; no other multi-command lines are legal.)

It's 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 you had

```
Nowxx is the time
```

you can say

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of //.)

## 15.16 EXERCISE 5

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, do this:

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

You get

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a *g* (for "global") 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 works except blanks or tabs.

(If you get funny results using any of the characters

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

read the section on "Special Characters".)

## 15.17 CONTEXT SEARCHING

With the substitute command mastered, you can move on to another highly important idea of *ed* – context searching.

Suppose you have the original three line text in the buffer:

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

Suppose you want to find the line that contains *their* so you can change it to *the*. Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

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

```
/string of characters you want to find/
```

For example, the *ed* command

```
/their/
```

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

```
to come to the aid of their party.
```

"Next occurrence" means that *ed* starts looking for the string at line *.+1*, 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't be found in any line, *ed* types the error message

```
?
```

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this:

```
/their/s/their/the/p
```

which yields

```
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, print the line.

The expression */their/* is a context search expression. In their 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, you could say

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. You could print all three lines by, for instance

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

```
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.

## 15.18 EXERCISE 6

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. This is sometimes useful if you go too far while looking for some string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

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

read the section on "Special Characters".)

*Ed* provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

```
/string/
```

finds 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 shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which finds 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."

## 15.19 CHANGE AND INSERT

This section discusses the *change* command

```
c
```

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

```
i
```

which is used for inserting a group of one or more lines.

"Change",writtenas

```
c
```

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

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

The lines you type between the *c* command and the . takes the place of the original lines between start line and end line. This is most useful in replacing a line or several lines that have errors in them.

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 . to end the input – this works just like the . in the append command and must appear by itself on 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 instance

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

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

## 15.20 EXERCISE 7

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

```
start, end d
i
. . . text . . .
.
```

is almost the same as

```
start, end c
. . . text . . .
.
```

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

Experiment with *a* and *i*, to see that they are similar, but not the same.

```
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, while *a* appends after line dot.

## 15.21 MOVING TEXT AROUND

The move command *m* is used for cutting and pasting – it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

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

(Do you see why?) but you can do it a lot easier with the *m* command:

```
1,3m$
```

The general case is

*start line, end line* `m` *after this line*

Notice that there is a third line to be specified – the place where the moved stuff gets put. 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.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the *-1*: the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

## 15.22 THE GLOBAL COMMANDS

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 where the s command will.*

There can be several commands (including *a*, *c*, *i*, *r*, *w*, but not *g*); in that case, every line except the last must end with a backslash \:

```
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*:

```
v/ /d
```

deletes every line that does not contain a blank.

## 15.23 SPECIAL CHARACTERS

You may have noticed that things just don't work right when you used some characters like ., *, $, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only*, . means "any character," not a period, so

```
/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*." A complete list of the special characters that can cause trouble is the following:

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

*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. Thus

```
s/\\\.\*/backslash dot star/
```

changes \.* into "backslash dot star".

Here is a hurried 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 finds

```
string
```

but not

```
the string...
```

The dollar-sign $ is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

finds only an occurrence of *string* that is at the end of some line. This implies, of course, that

```
/^string$/
```

finds only a line that contains just *string*, and

```
/^.$/
```

finds a line containing exactly one character.

The character ., as mentioned above, matches anything;

```
/x.y/
```

matches any of

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

This is useful in conjunction with *, which is a repetition character; *a** is a shorthand for "any number of *a*'s," so .* matches any number of anythings. This is used like this:

```
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.)

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

```
/[0123456789]/
```

matches any single digit – any one of the characters inside the braces causes 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. You could just retype the line, but this is tedious. Or you could say

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

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

```
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 don't 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 & saves 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 \:

```
s/ampersand/\&/
```

converts the word "ampersand" into the literal symbol & in the current line.

## 15.24 SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of *e*, *r*, and *w*, followed by a file name. Only one command is allowed per line, but a *p* command can follow any other command (except for *e*, *r*, *w* and *q*).

**a** Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until . is typed on a new line. Dot is set to the last line appended.

**c** Change the specified lines to the new text that follows. The new lines are terminated by a ., as with *a*. If no lines are specified, replace line dot. Dot is set to 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 deleted, in which case dot is set to $.

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

**f** Print remembered filename. If a name follows *f* the remembered name is set to it.

**g** The command

```
g/---/commands
```

executes the commands on those lines that contain ---, which can be any context search expression.

**i** Insert lines before specified line (or dot) until a . is typed on a new line. Dot is set to last line inserted.

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

**p** Print specified lines. If none specified, print line dot. A single line number is equivalent to *line–number p*. A single return prints *.+1*, the next line.

**q** Quit *ed*. Wipes out all text in buffer if you give it twice in a row without first giving a *w* command.

**r** Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

**s** The command

```
s/string1/string2/
```

substitutes the characters *string1* into *string2* in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. *s* changes only the first occurrence of *string1* on a line; to change all of them, type a *g* after the final slash.

**v** The command

```
v/---/commands
```

executes *commands* on those lines that *do not* contain ---.

**w** Write out buffer onto a file. Dot is not changed.

**.=** Print value of dot. (= by itself prints the value of $.)

**!** The line

```
!command-line
```

causes *command–line* to be executed as a shell command.

**/-----/**

Context search. Search for next line that contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at *.+1*, wraps around from $ to 1, and continues to dot, if necessary.

**?-----?**

Context search in reverse direction. Start search at *.–1*, scan to 1, wrap around to $.

# CHAPTER 16
# ADVANCED GUIDE TO ED

This chapter is meant to help you make effective use of the editor *ed*. It provides explanations and examples of:

- special characters, line addressing and global commands.
- commands for "cut and paste" operations on files and parts of files, and the *r*, *w*, *m* and *t* commands.
- editing scripts.

Although aimed at non-programmers, this chapter should help new users with any background get their jobs done more easily.

## 16.1 INTRODUCTION

This document is intended as a sequel to Chapter 15, providing explanations and examples of how to edit with less effort. Further information on all commands discussed here can be found in Chapter 17.

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying.

A word of advice. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A chapter like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

## 16.2 SPECIAL CHARACTERS

It is worthwhile to know how to get the most out of *ed* for the least effort. The next few sections discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will, and the others should give you ideas to store away for future use. And as always, until you try these things, they remain theoretical knowledge, not something you have confidence in.

### 16.2.1 The List Command

*Ed* provides two commands for printing the contents of the lines you're editing. Most people are familiar with *p*, in combinations like

```
1,$p
```

to print all the lines you're editing, or

```
s/abc/def/p
```

to change 'abc' to 'def' on the current line. Less familiar is the *list* command *l* (the letter '*l*'), which gives slightly more information than *p*. In particular, *l* makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, *l* prints each tab as – and each backspace as –. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The *l* command also 'folds' long lines for printing – any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the *l* command prints in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary – they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

### 16.2.2 The Substitute Command

Most of the next few sections discuss the substitute command *s*. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any *ed* command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing *g* after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing *g* changes *all* of them.

Either form of the *s* command can be followed by *p* or *l* to 'print' or 'list' (as described in the previous section) the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all valid, and mean slightly different things. Make sure you know what the differences are.

Of course, any *s* command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

If you add a *p* or *l* to the end of any of these substitute commands, only the last line that got changed is printed, not all the lines. How to print all the lines that were modified is discussed later.

## 16.2.3 The Undo Command

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command *u* lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

## 16.2.4 Metacharacters

As you have undoubtedly noticed when you use *ed*, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. The next several sections talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
x-y
x□y
x.y
```

and so on. (The symbol "□" stand for a space in order to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by *l*. Suppose you have a line that, when printed with the *l* command, appears as

```
.... th\07is ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this fails. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if you say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is*.

Bear in mind that since '.' matches any single character, the command

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

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in *ed*, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/./.
```

The first '.' is a line number, the number of the line you are editing, which is called 'line dot'. (Line dot is discussed more in Section 16.3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result is

```
.ow is the time.
```

which is probably not what you intended.

### 16.2.5 The Backslash

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters '\.' is considered by *ed* to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/.PP/
```

isn't adequate, for it finds a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
/\.PP/
```

you find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

```
/\/
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
/\\/
```

does work. Similarly, you can search for a forward slash '/' with

```
/\//
```

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which converts the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\./
s/x../x/
s/..y/y/
```

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an *s* command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter – to delete all the slashes, type

```
s:/::g
```

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to *ed* or any other program.

When you are adding text with *a* or *i* or *c*, backslash is not special. Only put in one backslash for each one you really want.

## 16.2.6 The Dollar Sign

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

```
Now is the
```

and you wish to add the word 'time' to the end. Use the $ like this:

```
s/$/□time/
```

to get

```
Now is the time
```

Notice that a space is needed before 'time' in the substitute command, or you get

```
Now is thetime
```

As another example, replace the second comma in the following line with a period without altering the first:

```
Now is the time, for all good men,
```

The command needed is

```
s/,$/./
```

The $ sign here provides context to make specific which comma you mean. Without it, of course, the *s* command would operate on the first comma to produce

```
Now is the time. for all good men,
```

As another example, to convert

```
Now is the time.
```

into

```
Now is the time?
```

as you did earlier, you can use

```
s/.$/?/
```

Like '.', the '$' has multiple meanings depending on context. In the line

```
$s/$/$/
```

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

### 16.2.7 The Circumflex

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

```
/the/
```

you in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

```
s/^/□/
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.PP
```

you can use the command

```
/^\.PP$/
```

### 16.2.8 The Asterisk

Suppose you have a line that looks like this:

*text x  y text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the *x* and the *y*. Suppose the job is to replace all the spaces between *x* and *y* by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by an asterisk stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

```
s/x□*y/x□y/
```

The construction '□*' means 'as many spaces as possible'. Thus 'x□*y' means 'an x, as many spaces as possible, then a y'.

The asterisk can be used with any character, not just space. If the original example was instead

*text x--------y text*

then all '–' signs can be replaced by a single space with the command

```
s/x-*y/x□y/
```

Finally, suppose that the line was

*text x.....y text*

Can you see what trap lies in wait for the unwary? If you blindly type

```
s/x.*y/x□y/
```

what happens? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

*text x text x.....y text y text*

then saying

```
s/x.*y/x□y/
```

takes everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

```
s/x\.*y/x□y/
```

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

```
Now is the time for all good men ....
```

into

```
Now is the time.
```

use '.*' to eat up everything after the 'for':

```
s/□for.*/./
```

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if your line contained

*text xy text x y text*

and you said

```
s/x□*y/x□y/
```

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

```
/x□□*y/
```

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by an asterisk. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a valid number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

```
s/xx*/y/g
```

'xx*' is one or more x's.

## 16.2.9 The Brackets

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*// 1,$s/^2*// 1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [ and ].

The construction

```
[0123456789]
```

matches any single digit – the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

```
/[.\$^[]/
```

Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lower case letters, and [A–Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

```
[^0-9]
```

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

## 16.2.10 The Ampersand

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

```
s/the/& best/
```

and the '&' stands for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

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

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

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

converts the original line into

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

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

## 16.2.11 Substituting Newlines

*ed* provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

```
text xy text
```

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the *roff* or *nroff* formatting command '.ul'.

```
text a very big text
```

The command

```
s/□very□/\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

## 16.2.12 Joining Lines

Lines can also be joined together, but this is done with the *j* command instead of *s*. Given the lines

```
Now is
□the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, which is why a blank is shown at the beginning of the second line.

All by itself, a *j* command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section x.3.)

## 16.2.13 Rearranging a Line

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an *s* command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \( and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \(...\) pair, '\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\),□*\(.*\)/\2□\1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands *g* and *v* discussed in Section 16.4 provide a way for you to

print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

## 16.3 LINE ADDRESSING IN THE EDITOR

The next general area to be discussed is that of line addressing in *ed*, that is, how you specify what lines are to be affected by editing commands. You have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

### 16.3.1 Address Arithmetic

The next step is to combine the line numbers like '.', '$', '/.../' and '?...?' with '+' and '–'. Thus

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

```
$-5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

```
.-3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

```
.-3,.3p
```

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '–' and '+' as line numbers by themselves.

```
-
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does '-3'. Thus

```
-3,+3p
```

is also identical to the examples above.

```
Since `-' is shorter than `.-1', constructions like

-,.s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '-' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search

```
/thing/--
```

finds the line containing 'thing', and positions you two lines before it.

## 16.3.2 Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
.... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//&□&/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

### 16.3.3 Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines are affected by a command if you don't specify the lines it is to act on, and on what line you are positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like *s* to make a substitution on that line, or *p* to print it, or *l* to list it, or *d* to delete it, or *a* to append text after it, or *c* to change it, or *i* to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were – dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command *d* leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands *a*, *c* and *i* by default all affect the current line – if you give no line number with them, *a* appends text after the current line, *c* changes the current line, and *i* inserts text before the current line.

*a*, *c*, and *i* behave identically in one respect – when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
... text ...
... botch ...          (minor error)
.
s/botch/correct/       (fix botched line)
a
... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
... text ...
... horrible botch ...          (major error)
.
c                               (replace entire line)
... fixed up line ...
```

Experiment to determine what happens if you add *no* lines with *a*, *c* or *i*.

The *r* command reads a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line

read in. Remember that you can even say *0r* to read a file in at the beginning of the text. (You can also say *0a* or *1i* to start adding text at the beginning.)

The *w* command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The *w* command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the *w* command is so easy to use, save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the *s* command. The rule is simple – you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

## 16.3.4 Semicolon

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is invalid to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In *ed*, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in the example above, the command

```
/a/;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;//
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a valid line number), for this starts the search at line 1.

## 16.3.5 Interrupting the Editor

As a final note on what dot gets set to, be aware that if you press the interrupt or delete or rubout or break key while *ed* is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable – if you are reading or writing a file or making substitutions or deleting lines, these are stopped in some clean but unpredictable state in

the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then press delete, you are *not* sitting on that line or even near it. Dot is left where it was when the *p* command was started.

## 16.4 GLOBAL COMMANDS

The global commands *g* and *v* are used to perform one or more editing commands on all lines that either contain (*g*) or don't contain (*v*) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '.').

The *v* command is identical to *g*, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '.' – the actual text lines.

The command that follows *g* or *v* can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, you could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice '//' in the substitute command means 'the previous pattern', in this case, 'Unix'. The *p* command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is

possible for the command that follows a *g* or *v* to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The *g* and *v* commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

## 16.4.1 Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The '\' signals the *g* command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a *g* command.)

Watch out for this problem: the command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it is 'x' (as expected), and sometimes it is 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute *a*, *c* and *i* commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a '.' to terminate the *i* command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

## 16.5 CUT AND PASTE

This section describes how to manipulate pieces of files – individual lines or groups of lines. This is an area where new users seem unsure of themselves.

### 16.5.1 Filenames

The first step is to ensure that you know the *ed* commands for reading and writing files. Of course you can't go very far without knowing *r* and *w*. Equally useful, but less well known, is the 'edit' command *e*. Within *ed*, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The *e* command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the *q* command, then re-entered *ed* with a new file name, except that if you have a pattern remembered, then a command like // still works.

If you enter *ed* with the command

```
ed file
```

*ed* remembers the name of the file, and any subsequent *e*, *r* or *w* commands that don't contain a filename refer to this remembered file. Thus

```
ed file1
... (editing) ...
w          (writes back in file1)
e file2    (edit new file, without leaving editor)
... (editing on file2) ...
w          (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving *ed* and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many systems use *e* as a synonym for *ed*.)

You can find out the remembered file name at any time with the *f* command; just type *f* without a file name. You can also change the name of the remembered file name with *f*; a useful sequence is

```
ed precious
f junk
... (editing) ...
```

which gets a copy of a precious file, then uses *f* to guarantee that a careless *w* command won't clobber the original.

### 16.5.2 Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it is formatted properly by *nroff* or *troff*. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r  table
```

The critical line is the last one. As said earlier, the *r* command reads a file; here you asked for it to be read in right after line dot. An *r* command without any address adds lines at the end, so it is the same as $*r*.

## 16.5.3 Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited there is

```
.TS
...[lots of stuff]
.TE
```

which is the way a table is set up for the *tbl* program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the *w* command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it – don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

## 16.5.4 Moving Lines Around

Suppose you want to move a paragraph from its present position in a file to the end. How would you do it? As a concrete example, suppose each paragraph in the file begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/-w temp
.,//-d
$r temp
```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/-') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

That's the brute force way. The easier way (often) is to use the *move* command *m* that *ed* provides – it lets you do the whole set of operations at one crack, without any temporary file.

The *m* command is like many other *ed* commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
.,/^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m--
```

does the interchange.

As you can see, the *m* command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste – do what you have most confidence in. The main difficulty with the *m* command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may not move the lines you thought you did. The result of a botched *m* command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a *w* command before doing anything complicated; then if you goof, it's easy to back up to where you were.

## 16.5.5 Marks

*Ed* provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is *k*; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the *k*, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *'a*. Then find the last line and mark it with *'b*. Now position yourself at the place where the stuff is to go and say

```
'a,'bm.
```

Bear in mind that only one line can have a particular mark name associated with it at any given time.

## 16.5.6 Copying Lines

As mentioned earlier, you can save a line that is hard to type or used often, in order to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

*ed* provides another command, called *t* (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The *t* command is identical to the *m* command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for *t* is for creating a series of lines that differ only slightly. For example, you can say

```
a
..... x .....          (long line)
.
t.                     (make a copy)
s/x/y/                 (change it a bit)
t.                     (make third copy)
s/y/z/                 (change it a bit)
```

and so on.

## 16.5.7 The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other shell command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command ! provides a way to do this.

If you say

```
!any shell command
```

your current editing state is suspended, and the shell command you asked for is executed. When the command finishes, *ed* signals you by printing another !; at that point you can resume editing.

You can really do *any* shell command, including another *ed*. (This is quite common, in fact.) In this case, you can even do another !.

# CHAPTER 17
# ED REFERENCE

*Ed* operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*.

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command*, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command can appear on a line. Certain commands allow the addition of text to the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period '.' alone at the beginning of a line.

*Ed* supports a limited form of *regular expression* notation. 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. In the following specification for regular expressions the word 'character' means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \[. and sometimes ^*$.
2. A . matches any character.
3. A \ followed by any character except a digit or () matches that character.
4. A nonempty string *s* bracketed [*s*] (or [^*s*]) matches any character in (or not in) *s*. In *s*, \ has no special meaning, and ] can only appear as the first letter. A substring *a*-*b*, with *a* and *b* in ascending ASCII order, stands for the inclusive range of ASCII characters.
5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.
6. A regular expression, *x*, of form 1-8, bracketed \(*x*\) matches what *x* matches.
7. A \ followed by a digit *n* matches a copy of the string that the bracketed regular expression beginning with the *n*th \( matched.

8. A regular expression of form 1–8, *x*, followed by a regular expression of form 1–7, *y* matches a match for *x* followed by a match for *y*, with the *x* match being as long as possible while still permitting a *y* match.
9. A regular expression of form 1–8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.
10. A regular expression of form 1–9 picks out the longest among the leftmost matches in a line.
11. An empty regular expression stands for a copy of the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line that is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character can be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character '.' addresses the current line.
2. The character '$' addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*–th line of the buffer.
4. ''*x*' addresses the line marked with the name *x*, which must be a lower–case letter. Lines are marked with the *k* command described below.
5. A regular expression enclosed in slashes '/' addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.
6. A regular expression enclosed in queries '?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.
7. An address followed by a plus sign '+' or a minus sign '–' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign can be omitted.
8. If an address begins with '+' or '–' the addition or subtraction is taken with respect to the current line; e.g. '–5' is understood to mean '.–5'.
9. If an address ends with '+' or '–', 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address '–' refers to the line before the current line. Moreover, trailing '+' and '–' characters have cumulative effect, so '––' refers to the current line less 2.
10. To maintain compatibility with earlier versions of the editor, the character '^' in addresses is equivalent to '–'.

Commands can require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma ','. They can also be separated by a semicolon ';'. In this case the current line '.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address. The special form '%' is an abbreviation for the address pair '1,$'.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally invalid for more than one command to appear on a line. However, most commands can be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below. Commands can also be suffixed by 'n', meaning the output of the command is to be line numbered. These suffixes can be combined in any order.

(.)a <text> .

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is valid for this command; text is placed at the beginning of the buffer.

(., .)c <text> .

The change command deletes the addressed lines, then accepts input text that replaces these lines. '.' is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(., .)d The delete command deletes the addressed lines from the buffer. The line originally 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.

e filename The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '.' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent *r* or *w* command. If 'filename' is missing, the remembered name is used.

E filename This command is the same as *e*, except that no diagnostic results when no *w* has been given since the last buffer alteration.

f filename The filename command prints the currently remembered file name. If 'filename' is given, the currently remembered file name is changed to 'filename'.

(1,$)g/regular expression/command list

In the global command, the first step is to mark every line that matches the given regular expression. Then for every such line, the given command list is executed with '.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multiline list except the last line must be ended with '\'. *A*, *i*, and *c* commands and associated input are permitted; the '.' terminating input mode can be omitted if it would be on the last line of the command list. The commands *g* and *v* are not permitted in the command list.

(.)i

<text> . This command inserts the given text before the addressed line. '.' is left at the last line input, or, if there were none, at the line before the addressed

line. This command differs from the *a* command only in the placement of the text.

(., .+1)j This command joins the addressed lines into a single line; intermediate newlines simply disappear. '.' is left at the resulting line.

( . )k*x* The mark command marks the addressed line with name *x*, which must be a lower-case letter. The address form ''*x*' then addresses this line.

(., .)l The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in two-digit octal, and long lines are folded. The *l* command can be placed on the same line after any non-i/o command. Note: the *l* command mishandles DEL.

(., .)m*a* The move command repositions the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

(., .)n The number command prints the addressed lines with line numbers and a tab at the left.

(., .)p The print command prints the addressed lines. '.' is left at the last line printed. The *p* command can be placed on the same line after any non-i/o command.

(., .)P This command is a synonym for *p*.

q The quit command causes *ed* to exit. No automatic write of a file is done.

Q This command is the same as *q*, except that no diagnostic results when no *w* has been given since the last buffer alteration.

($)r filename

The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The file name is remembered if there was no remembered file name already. Address '0' is valid for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '.' is left at the last line read in from the file.

( ., .)s/regular expression/replacement/

( ., .)s/regular expression/replacement/g

The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any punctuation character can be used instead of '/' to delimit the regular expression and the replacement. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context can be suppressed by preceding it by '\'. The characters '*n*' where *n* is a digit, are 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.

Lines can be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by '\'.

One or two trailing delimiters can be omitted, implying the 'p' suffix. The special form 's' followed by *no* delimiters repeats the most recent substitute command on the addressed lines. The 's' can be followed by the letters r

(use the most recent regular expression for the left hand side, instead of the most recent left hand side of a substitute command), p (complement the setting of the *p* suffix from the previous substitution), or g (complement the setting of the *g* suffix). These letters can be combined in any order.

(., .)t*a* This command acts just like the *m* command, except that a copy of the addressed lines is placed after address *a* (which can be 0). '.' is left on the last line of the copy.

(., .)u The undo command restores the buffer to it's state before the most recent buffer modifying command. The current line is also restored. Buffer modifying commands are *a*, *c*, *d*, *g*, *i*, *k*, and *v*. For purposes of undo, *g* and *v* are considered to be a single buffer modifying command. Undo is its own inverse.

When *ed* runs out of memory (at about 8000 lines on any 16 bit mini-computer such as the PDP-11) This full undo is not possible, and *u* can only undo the effect of the most recent substitute on the current line. This restricted undo also applies to editor scripts when *ed* is invoked with the - option. Note: the *undo* command causes marks to be lost on affected lines.

(1, $)v/regular expression/command list

This command is the same as the global command *g* except that the command list is executed *g* with '.' initially set to every line *except* those matching the regular expression.

(1, $)w filename

The write command writes the addressed lines onto the given file. If the file does not exist, it is created. The file name is remembered if there was no remembered file name already. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). '.' is unchanged. If the command is successful, the number of characters written is printed.

(1, $)W filename

This command is the same as *w*, except that the addressed lines are appended to the file.

(1, $)wq filename

This command is the same as *w* except that afterwards a *q* command is done, exiting the editor after the file is written.

x A key string is demanded from the standard input. Later *r*, *e* and *w* commands encrypt and decrypt the text with this key by the algorithm of *crypt*. An explicitly empty key turns off encryption. (.+1)z or,

(.+1)z*n* This command scrolls through the buffer starting at the addressed line. 22 (or *n*, if given) lines are printed. The last line printed becomes the current line. The value *n* is sticky, in that it becomes the default for future *z* commands.

($)= The line number of the addressed line is typed. '.' is unchanged by this command.

!<shell command>

The remainder of the line after the '!' is sent to *sh* to be interpreted as a command. '.' is unchanged.

(.+1,.+1)<newline>

An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text. If two

addresses are present with no intervening semicolon, *ed* prints the range of lines. If they are separated by a semicolon, the second line is printed.

If an interrupt signal (ASCII DEL) is sent, *ed* prints '?interrupted' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and, on mini computers, 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 2 words.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non–ASCII characters.

# APPENDIX A
# GLOSSARY

**a.out**

Compilers that create executable images create them, by default, in the file *a.out*. for historical reasons.

**absolute pathname**

A *pathname* that begins with a slash (/) is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system: the *root* directory. *Pathnames* that are not *absolute* are called *relative* (see *relative pathname*).

**alias**

An *alias* specifies a shorter or different name for a command, or a transformation on a command to be performed in the C shell. The C shell has a command *alias* that establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases*.

**argument**

Commands receive a list of *argument* words. Thus the following command consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'.

```
echo a b c
```

The set of *arguments* after the *command name* is said to be the *argument list* of the command.

**argv**

The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language.

**background**

Commands started without waiting for them to complete are called *background* commands.

**base**

A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* – the part after the '.'. See *filename* and *extension*.

**bg**

The *bg* command causes a *suspended* job to continue execution in the *background*.

**bin**

A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are '/bin' containing the most heavily used commands and '/usr/bin' that contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

**break**

*Break* is a builtin command used to exit from loops within the control structure of the C shell.

**breaksw**

The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops.

**builtin**

A command executed directly by the C shell is called a *builtin* command. Most commands in are not built into the C shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

**case**

A *case* command is used as a label in a *switch* statement in the C shell's control structure, similar to that of the language C. Details are given in the C shell documentation.

**cat**

The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file'.

**cd**

The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory.

**chdir**

The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

**chsh**

The *chsh* command is used to change the shell that you use. For example:

```
chsh jones /bin/csh
```

It is only necessary to do this once. The next time jones logs in, he will be using *csh* rather than the Bourne shell.

**cmp**

*Cmp* is a program that compares files. It is usually used on binary files, or to see if two files are identical. For comparing text files the program *diff*, described in 'diff' is used.

**command**

A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory, is called a *command*.

**command name**

When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention is that the first word of a command names the function to be performed.

**command substitution**

The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution*.

**component**

A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable that has multiple strings as value is said to have several *components*; each string is a *component* of the variable.

**continue**

A builtin command that causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C.

**core dump**

When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb' or 'sdb' in order to determine what went wrong with the program. If the shell produces a message of the form

```
Illegal instruction (core dumped)
```

(where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

**cp**

The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used commands.

**csh**

The name of the C shell program.

**.cshrc**

The file *.cshrc* in your *home* directory is read by each C shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters that are to take effect globally.

**CTRL**

Certain special characters, called *control* characters, are produced by holding down the CTRL key on your terminal and simultaneously pressing another character. For example, CTRL C is produced by holding down the CTRL key while pressing the C key.

Usually the system prints an up-arrow (^) followed by the corresponding letter when you type a *control* character (e.g. ^C for CTRL C).

**cwd**

The *cwd* variable in the C shell holds the *absolute pathname* of the current *working directory*. It is changed by the C shell whenever your current *working directory* changes and should not be changed otherwise.

**date**

The *date* command prints the current date and time.

**debugging**

*Debugging* is the process of correcting mistakes in programs and shell scripts. The C shell has several options and variables that can be used to aid in *debugging*.

**default:**

The label *default*: is used within C shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on.

DELETE

The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be CTRL C.

**detached**

A command that continues running in the *background after you logout is said to be detached.*

**diagnostic**

An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output,* since that is often directed away from the terminal. Error messsages are instead written to the *diagnostic output* that can be directed away from the terminal, but usually is not. Thus *diagnostics* usually appear on the terminal.

**directory**

A structure that contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command changes you to another *directory*, and makes the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory.

**directory stack**

The C shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left.

**dirs**

The *dirs* command prints the C shell's *directory stack*.

**. (dot)**

Your current directory has the name '.' as well as the name printed by the command *pwd*; see also *dirs*. The current directory '.' is usually the first *component* of the search path contained in the variable *path*, thus commands that are in '.' are found first The character '.' is also used in separating *components* of filenames. The character '.' at the

beginning of a *component* of a *pathname* is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs.

**.. (dot-dot)**

Each directory has a file '..' in it that is a reference to its parent directory.

**du**

The *du* command prints the number of disk blocks is all directories below and including your current *working directory*.

**echo**

The *echo* command prints its arguments.

**else**

The *else* command is part of the 'if–then-else–endif' control command construct .

**endif**

If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true.

**EOF**

An *end–of–file* is generated by the terminal by a CTRL D, and whenever a command reads to the end of a file that it has been given as input. Commands receiving input from a *pipe* receive an *end–of–file* when the command sending them input completes. Most commands terminate when they receive an *end–of–file*. The C shell has an option to ignore *end–of–file* from a terminal input that may help you keep from logging out accidentally by typing too many CTRL D's.

**escape**

A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

```
echo \*
```

echoes the character '*' while just

```
echo *
```

echoes the names of the file in the current directory. In this example, \ *escapes* '*'.

ESCAPE

A nonprinting character, usually labelled ESC or ALTMODE on terminal keyboards, that is often used as a prefix character in text editors.

**/etc/passwd**

This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters. You can look at this file by typing:

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See 'finger', 'passwd', and 'grep' for more details.

### exit

The *exit* command is used to force termination of a C shell script, and is built into the C shell.

### exit status

A command that discovers a problem can reflect this back to the command (such as a C shell) that invoked (executed) it. It does this by returning a non–zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a C shell command script to give a non–zero *exit status*.

### expansion

The replacement of strings in the C shell input that contain metacharacters by other strings is referred to as the process of *expansion*. Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions*.

### expressions

*Expressions* are used in the C shell to control the conditional structures used in the writing of C shell scripts and in calculating values for these scripts. The operators available in C shell *expressions* are those of the language C.

### extension

Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '–me' nroff macro package might be stored in 'paper.me' while a formatted version might be kept in 'paper.out' and a list of spelling errors in 'paper.errs'.

### fg

The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground*.

### filename

Each file has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension*.

### filename expansion

*Filename expansion* uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files that have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily.

### flag

Many commands accept arguments that are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '–'. Thus the *ls* (list files) command has an option '–s' to list the sizes of files. This is specified

```
ls -s
```

**foreach**

The *foreach* command is used in C shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain C shell variable ranges through a specified list.

**foreground**

When commands are executing in the normal way such that the C shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground*. This is as opposed to *background*. *Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard.

**goto**

The C shell has a command *goto* used in C shell scripts to transfer control to a given label.

**grep**

The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

prints each line in the file */etc/passwd* which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed' and 'ex'. *Grep* stands for 'globally find *regular expression* and print'.

**head**

The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text that you are wondering about it is sometimes useful to run *head* with these files as arguments. This usually shows enough of what is in these files to let you decide which you are interested in.

*Head* is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used.

**history**

The *history* mechanism of the C shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The C shell has a *history list* where these commands are kept, and a *history* variable that controls how large this list is.

**home directory**

Each user has a *home directory* that is given in your entry in the password file, */etc/passwd*. This is the directory that you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~'.

**if**

A conditional command within the C shell, the *if* command is used in C shell command scripts to make decisions about what course of action to take next.

**ignoreeof**

Normally, your C shell exits, printing 'logout' if you type a CTRL D at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you

wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many CTRL D characters, logging yourself off.

### input

Many commands take information from the terminal or from files that they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands also read from a file specified as argument. Commands placed in *pipelines* read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts.

### interrupt

An *interrupt* is a signal to a program that is generated by pressing the DELETE key (although users can and often do change the interrupt character, usually to CTRL C). It causes most programs to stop execution. Certain programs, such as the C shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the C shell is executing another command and waiting for it to finish, the C shell does not listen to *interrupts*. The C shell often wakes up when you press *interrupt* because many commands die when they receive an *interrupt*.

### job

One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended*.

### job control

The builtin functions that control the execution of jobs are called *job control* commands. These are *bg*, *fg*, *stop*, *kill*.

### job number

When each job is started it is assigned a small number called a *job number* that is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job.

### jobs

The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended*.

### kill

A command that sends a signal to a job causing it to terminate.

### .login

The file *.login* in your *home* directory is read by the C shell each time you log in and the commands there are executed. There are a number of commands that are usefully placed here, especially *set* commands to the C shell itself.

### login shell

The shell that is started on your terminal when you login is called your *login shell*. It is different from other shells that you may run (e.g. on shell scripts) in that it reads the

*.login* file before reading commands from the terminal and it reads the *.logout* file after you logout.

**logout**

The *logout* command causes a login C shell to exit. Normally, a login C shell exits when you press CTRL D generating an *end-of-file*, but if you have set *ignoreeof* in you *.login* file then this does not work and you must use *logout* to log out.

**.logout**

When you log out, the C shell executes commands from the file *.logout* in your *home* directory after it prints 'logout'.

**lpr**

The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline*.

**ls**

The *ls* (list files) command is one of the most commonly used commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories.

**mail**

The *mail* program is used to send and receive messages from other system users.

**make**

The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts.

**makefile**

The file containing commands for *make* is called *makefile*.

**metacharacter**

Many characters that are neither letters nor digits have special meaning to the shell. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a *metacharacter* is the character '>' that is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words.

**mkdir**

The *mkdir* command is used to create a new directory.

**modifier**

Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear.

**more**

The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by

line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file.

**noclobber**

The C shell has a variable *noclobber* that can be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the C shell.

**noglob**

The C shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']'.

**notify**

The *notify* command tells the C shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the C shell to always report the termination of *background* jobs exactly when they occur.

**onintr**

The *onintr* command is built into the C shell and is used to control the action of a C shell command script when an *interrupt* signal is received.

**output**

Many commands result in some lines of text that are called their *output*. This *output* is usually placed on what is known as the *standard output* which is normally connected to the user's terminal. The C shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file. Using the *pipe* mechanism and the metacharacter '|' it is also possible for the *standard output* of one command to become the *standard input* of another command. Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer. Similarly the *write* command places its output on another user's terminal rather than its *standard output*. Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the *standard output* has been sent to a file or another command, but it is possible to direct error diagnostics along with *standard output* using a special metanotation.

**pushd**

The *pushd* command, that means 'push directory', changes the C shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name.

**path**

The C shell has a variable *path* that gives the names of the directories in which it searches for the commands that it is given. It always checks first to see if the command it is given is built into the C shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the C shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

```
path (. /usr/ucb /bin /usr/bin)
```

the C shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command. If the command cannot be found, the C shell prints an error diagnostic. Scripts of C shell commands are

executed using another C shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

```
chmod 755 script
```

as executed to turn this execute permission on. If you add new commands to a directory in the *path*, you should issue the command *rehash*.

**pathname**

A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* that begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* can name a directory, but usually names a file.

**pipeline**

A group of commands that are connected together, the *standard output* of each connected to the *standard input* of the next, is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|'.

**popd**

The *popd* command changes the C shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so.

**port**

The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial–up access, and some of which are permanently wired directly to specific terminals.

**pr**

The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified.

**printenv**

The *printenv* command is used to print the current setting of variables in the environment.

**process**

An instance of a running program is called a *process*. The system assigns each *process* a unique number when it is started – called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.

**program**

Usually synonymous with *command*; a binary file or shell command script that performs a useful function is often called a *program*.

**prompt**

Many programs print a *prompt* on the terminal when they expect input. Thus the editor 'ex' prints a ':' when it expects input. The C shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal. The C shell has a

variable *prompt* that can be set to a different value to change the C shell's main *prompt*. This is mostly used when debugging the C shell.

**ps**

The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked. Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

**pwd**

The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.

**quit**

The *quit* signal, generated by a CTRL\, is used to terminate programs that are behaving unreasonably. It normally produces a core image file.

**quotation**

The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\', is referred to as *quotation*.

**redirection**

The routing of input or output from or to a file is known as *redirection* of input or output.

**rehash**

The *rehash* command tells the C shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories.

**relative pathname**

A *pathname* that does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the working directory, and subsequent *components* between '/' characters refer to directories below the wor*king directory*. *Pathnames* that are not relative are called *absolute pathnames*.

**repeat**

The *repeat* command iterates another command a specified number of times.

**root**

The directory that is at the top of the entire directory structure is called the *root* directory since it is the 'root' of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is '/'. *Pathnames* starting with '/' are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation.

**scratch file**

Files whose names begin with a '#' are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non–use, or more frequently if disk space becomes tight.

**script**

Sequences of shell commands placed in a file are called shell command *scripts*. It is often possible to perform simple tasks using these *scripts* without writing a program in a language such as C, by using the shell to selectively run other programs.

**set**

The builtin *set* command is used to assign new values to C shell variables and to show the values of the current variables. Many C shell variables have special meaning to the C shell itself. Thus by using the *set* command the behavior of the C shell can be affected.

**setenv**

Variables in the environment 'environ' can be changed by using the *setenv* builtin command. The *printenv* command can be used to print the value of the variables in the environment.

**shell**

A *shell* is a command language interpreter. It is possible to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned.

**shell script**

See *script*.

**signal**

A *signal* is a short message that is sent to a running program that causes something to happen to that process. *Signals* are sent either by typing special *control* characters on the keyboard or by using the *kill* or *stop* commands.

**sort**

The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags*.

**source**

The *source* command causes the C shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them.

**special character**

See *metacharacters*.

**standard**

Refers often to the *standard input* and *standard output* of commands. See *input* and *output*.

**status**

A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands can return non–zero *status* to indicate that some abnormal event has occurred. The C shell variable *status* is set to the *status* returned by the last command. It is most useful in C shell commmand scripts.

**stop**

The *stop* command causes a *background* job to become *suspended*.

**string**

A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters.

**stty**

The *stty* program changes certain parameters inside the kernel that determine how your terminal is handled. See 'stty' for a complete description.

**substitution**

The C shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history s*ubstitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. *Substitutions* are also referred to as *expansions*.

**suspended**

A job becomes *suspended* after a STOP signal is sent to it, either by typing a CTRL Z at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command.

**switch**

The *switch* command of the C shell allows the C shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C.

**termination**

When a command that is being executed finishes, it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end–of–file* from their standard input. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal. The *kill* program terminates specified jobs.

**then**

The *then* command is part of the C shell's 'if-then–else–endif' control construct used in command scripts.

**time**

The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command.

**tset**

The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file.

**tty**

The word *tty* is a historical abbreviation for 'teletype' which is frequently used to indicate the *port* to which a given terminal is connected. The *tty* command prints the name of the *tty* or *port* to which your terminal is presently connected.

**unalias**

The *unalias* command removes aliases.

**unset**

The *unset* command removes the definitions of C shell variables.

**variable expansion**

See *variables* and *expansion*.

**variables**

*Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the C shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing C shell scripts.

**verbose**

The *verbose* C shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging C shell scripts. The *verbose* variable is set by the C shell's *–v* command line option.

**wc**

The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments.

**while**

The *while* builtin control construct is used in C shell command scripts.

**word**

A sequence of characters that forms an argument to a command is called a *word*. Many characters that are neither letters, digits, '–', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters can be made into a *word* by surrounding it with "' characters except for the characters "' and '!' which require special treatment. This process of placing special characters in *words* without their special meaning is called *quoting*.

**working directory**

At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.

**write**

The *write* command is used to communicate with other users who are logged in.

# APPENDIX B

# THE ASCII CHARACTER SET

The file /usr/pub/ascii contains the following map of the ASCII character set, and can be printed as needed.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | nul | 001 | soh | 002 | stx | 003 | etx | 004 | eot | 005 | enq | 006 | ack | 007 | bel |
| 010 | bs | 011 | ht | 012 | nl | 013 | vt | 014 | np | 015 | cr | 016 | so | 017 | si |
| 020 | dle | 021 | dc1 | 022 | dc2 | 023 | dc3 | 024 | dc4 | 025 | nak | 026 | syn | 027 | etb |
| 030 | can | 031 | em | 032 | sub | 033 | esc | 034 | fs | 035 | gs | 036 | rs | 037 | us |
| 040 | sp | 041 | ! | 042 | " | 043 | # | 044 | $ | 045 | % | 046 | & | 047 | ´ |
| 050 | ( | 051 | ) | 052 | * | 053 | + | 054 | , | 055 | - | 056 | . | 057 | / |
| 060 | 0 | 061 | 1 | 062 | 2 | 063 | 3 | 064 | 4 | 065 | 5 | 066 | 6 | 067 | 7 |
| 070 | 8 | 071 | 9 | 072 | : | 073 | ; | 074 | < | 075 | = | 076 | > | 077 | ? |
| 100 | @ | 101 | A | 102 | B | 103 | C | 104 | D | 105 | E | 106 | F | 107 | G |
| 110 | H | 111 | I | 112 | J | 113 | K | 114 | L | 115 | M | 116 | N | 117 | O |
| 120 | P | 121 | Q | 122 | R | 123 | S | 124 | T | 125 | U | 126 | V | 127 | W |
| 130 | X | 131 | Y | 132 | Z | 133 | [ | 134 | \ | 135 | ] | 136 | ^ | 137 | _ |
| 140 | ` | 141 | a | 142 | b | 143 | c | 144 | d | 145 | e | 146 | f | 147 | g |
| 150 | h | 151 | i | 152 | j | 153 | k | 154 | l | 155 | m | 156 | n | 157 | o |
| 160 | p | 161 | q | 162 | r | 163 | s | 164 | t | 165 | u | 166 | v | 167 | w |
| 170 | x | 171 | y | 172 | z | 173 | { | 174 | \| | 175 | } | 176 | ~ | 177 | del |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | nul | 01 | soh | 02 | stx | 03 | etx | 04 | eot | 05 | enq | 06 | ack | 07 | bel |
| 08 | bs | 09 | ht | 0a | nl | 0b | vt | 0c | np | 0d | cr | 0e | so | 0f | si |
| 10 | dle | 11 | dc1 | 12 | dc2 | 13 | dc3 | 14 | dc4 | 15 | nak | 16 | syn | 17 | etb |
| 18 | can | 19 | em | 1a | sub | 1b | esc | 1c | fs | 1d | gs | 1e | rs | 1f | us |
| 20 | sp | 21 | ! | 22 | " | 23 | # | 24 | $ | 25 | % | 26 | & | 27 | ´ |
| 28 | ( | 29 | ) | 2a | * | 2b | + | 2c | , | 2d | - | 2e | . | 2f | / |
| 30 | 0 | 31 | 1 | 32 | 2 | 33 | 3 | 34 | 4 | 35 | 5 | 36 | 6 | 37 | 7 |
| 38 | 8 | 39 | 9 | 3a | : | 3b | ; | 3c | < | 3d | = | 3e | > | 3f | ? |
| 40 | @ | 41 | A | 42 | B | 43 | C | 44 | D | 45 | E | 46 | F | 47 | G |
| 48 | H | 49 | I | 4a | J | 4b | K | 4c | L | 4d | M | 4e | N | 4f | O |
| 50 | P | 51 | Q | 52 | R | 53 | S | 54 | T | 55 | U | 56 | V | 57 | W |
| 58 | X | 59 | Y | 5a | Z | 5b | [ | 5c | \ | 5d | ] | 5e | ^ | 5f | _ |
| 60 | ` | 61 | a | 62 | b | 63 | c | 64 | d | 65 | e | 66 | f | 67 | g |
| 68 | h | 69 | i | 6a | j | 6b | k | 6c | l | 6d | m | 6e | n | 6f | o |
| 70 | p | 71 | q | 72 | r | 73 | s | 74 | t | 75 | u | 76 | v | 77 | w |
| 78 | x | 79 | y | 7a | z | 7b | { | 7c | \| | 7d | } | 7e | ~ | 7f | del |

# INDEX

ALLIANT COMPUTER B.V.
Weverstede 1
3431 JS NIEUWEGEIN
Tel. 03402 - 46624
Fax: 03402 - 31767