™

ICON/UXV User Guide

ICON INTERNATIONAL

764 East Timpanogos Parkway
Orem, Utah 84057
(801) 225-6888

USER GUIDE

ICON/UXV Operating System

© Copyright 1988
Icon International, Inc.
All rights reserved worldwide.

The information contained within this manual is the property of Icon International, Inc. This manual shall not be reproduced in whole nor in part without prior written approval from Icon International, Inc.

Icon International, Inc. reserves the right to make changes, without notice, to the specifications and materials contained herein, and shall not be responsible for any damages (including consequential) caused by reliance on the material as presented, including, but not limited to, typographical, arithmetic, and listing errors.

The UNIX® Software and Text Source for this manual is under license from AT&T.

Order No. 172-036-003 A (Manual Assembly)
Order No. 171-063-003 A (Manual Pages only)

Trademarks

The Icon logo is a registered trademark of Icon International, Inc.
UNIX is a registered trademark of AT&T.
DEC, PDP, UNIBUS, and MASSBUS are trademarks of Digital Equipment Corporation.
HP is a trademark of Hewlett-Packard, Inc.
DIABLO is a trademark of Xerox Corporation.
TEKTRONIX is a registered trademark of Versatec Corporation.
TELETYPE is a trademark of AT&T Teletype Corporation.
3B and DOCUMENTER'S WORKBENCH are trademarks of AT&T Technologies.

Change Record Page

ICON/UXV User Guide

Manual Pages Part No. 171-063-003

Date	Revision	Description	Pages Affected
Mar. 1988	A0	Initial production release	All
Aug. 1988	A1	Add Appendix G "An Introduction to the C Shell" and renumber the Glossary	ix, Appendices contents, Appendix G, Glossary

CONTENTS

PART 2. UNIX SYSTEM TUTORIALS

HOW TO READ THIS GUIDE

The UNIX® system is a family of computer operating systems developed by AT&T Bell Laboratories and licensed by AT&T Technologies, Inc. Because it can run on many sizes and types of computers and because of all it can do, the UNIX system has gained wide popularity since it was introduced in the late 1960s. Now, either by choice or by fate, you are interested in learning something about it.

This guide is written to help you, the user, understand how the ICON/UXV system works and what it can do for you. It introduces you to ICON/UXV, Release 3.2. New versions of the ICON/UXV system, called releases, will be offered as changes are made or as improvements are added.

Who Should Read This Guide

Whether you are a newcomer to the world of computers or an experienced computer user who is unfamiliar with the ICON/UXV system, this guide is for you. Although it contains technical material, it can be understood by either a newcomer or an expert. You will find that learning to use the ICON/UXV system requires some thought and time, but you will be rewarded with power and flexibility unattainable with other operating systems.

This guide assumes that you are one of a number of people using a computer on which the ICON/UXV system is running, and that there is a person responsible for monitoring and controlling the ICON/UXV system you are using. This person is the *system administrator*. If necessary, you also act as the system administrator. In this case, in addition to this guide, you should consult the documents you received when the ICON/UXV system programs were delivered to you. (See *Appendix A* for information on how to order additional copies.)

How This Guide Is Organized

The material in this guide is organized into three major parts: *ICON/UXV System Overview*, *ICON/UXV System Tutorials*, and *Supplementary Information and Reference Tools*. Both the major parts and the chapters in each part are separated by tab dividers.

The following list summarizes the contents of each major part:

- *ICON/UXV System Overview*—This part introduces you to the basic principles of the ICON/UXV operating system. The material in this part is organized into four chapters, each chapter building on information presented in preceding chapters. Therefore, it is recommended that you read chapters 1 through 4 in order. The chapters that make up this part are:

  - *Chapter 1, What is the ICON/UXV System?*—Acquaints you with the ICON/UXV system and explains how it works.

  - *Chapter 2, Basics for ICON/UXV System Users*—Covers topics related to using your terminal, obtaining a system account, and establishing contact with the ICON/UXV system.

- *Chapter 3, Using the File System*--Explains what the file system is, how you can organize information (data, text, and programs) using the file system, and how you can store and retrieve this information using appropriate commands.

- *Chapter 4, ICON/UXV System Capabilities*--Builds on material and terminology presented in the first three chapters. It highlights ICON/UXV system capabilities, such as command execution, text editing, electronic communication, programming, and aids to software development.

- *ICON/UXV System Tutorials*--Each chapter in this part takes a step-by-step approach to teach you about one aspect of the ICON/UXV system. You will gain the greatest benefit from them if you work through the examples and exercises at a terminal connected to the ICON/UXV system you will be using. The tutorials assume that you understand the concepts introduced in chapters 1 through 4. For example, before reading either the *Line Editor Tutorial* or the *Screen Editor Tutorial*, read the explanation of text editors in *Chapter 4*. The chapters that make up this part are:

  - *Chapter 5, Line Editor Tutorial*--Teaches you how to use the **ed** text editor to create and to modify text on a paper printing or a video display terminal.

  - *Chapter 6, Screen Editor Tutorial*--Teaches you how to use the **vi*** text editor to create and to modify text on a video display terminal.

  - *Chapter 7, Shell Tutorial*--Teaches you how to use the shell to automate repetitive jobs. The shell is the part of the ICON/UXV system that interprets the commands you type.

  - *Chapter 8, Communication Tutorial*--Teaches you how to send information to others, whether they are working on your ICON/UXV system or on a different ICON/UXV system.

- *Shell Commands*--Each chapter in this part is intended to provide information on how to use the **shell** provided with ICON/UXV based on UNIX System V Release 2. Knowledge of another programming language is not required when reading this document. Some examples are based on the DOCUMENTER'S WORKBENCH® Software which is available independently for the ICON/UXV system. Make sure that the system has DOCUMENTER'S WORKBENCH Software available before trying any of those examples. The chapters that uake up this part are:

  - *Chapter 9, Using Shell commands*--Builds on *Chapter 7* of this guide or the "hands-on" experience some have acquired. It is intended for those users who have some basic familiarity with **shell** but desire more detailed information.

  - *Chapter10, Shell Programming*--Provides information for programming with **shell**. Those users that intend to do **shell** programming should read Chapter 9 as well as Chapter 10.

- *Chapter 11, Examples of Shell Procedures*--Contains examples of **shell** programs.

It is important to note a few things about **shell**. The **shell** functions as a

- Command language—The **shell** reads command lines entered at a terminal and interprets the lines as requests to execute other programs.
- Programming language—The **shell** is a programming language just like BASIC, COBOL, Fortran, and other languages. The **shell** is a high-level programming language that is easy to learn. The programs written using the **shell** programming language are called **shell** scripts, procedures, or commands. These programs are stored in files and executed just like commands. The **shell** provides variables, conditional constructs, and iterative constructs.
- Working environment—The **shell** also provides an environment that can be tailored to an individual's or group's needs by manipulating environment variables.

All command names in this document are in **bold** font.

Normally when the system is ready for a command from a terminal, a prompt is displayed on the terminal (? by default). With certain commands, the system expects more than one line of terminal input. When this is the case, a secondary prompt is displayed (> by default). To avoid confusion with what the system displays and what the user types, this document does not show prompts displayed by the system unless noted otherwise.

- *Graphics*--This part provides numerical and graphical commands used to construct and edit numerical data plots and hierarchy charts. This part is designed for individuals experienced in using the ICON/UXV system, in a variety of ways, within the office environment (electronic mail, document preparation, data analysis, and so on). These individuals are not expected to know programming languages to use the *ICON/UXV Graphics*, but may write shell procedures for general purposes.

  - *Chapter 12, Overview*--Provides a general description of and an introduction to the ICON/UXV system graphic facility.

  - *Chapter 13, Statistical Network (stat)*--Describes a collection of routines that can be interconnected using the ICON/UXV operating system shell to form numerical processing networks.

  - *Chapter 14, Graphics Editor (ged)*--Describes an interactive editor used to display, edit, and construct drawings on TEKTRONIX† 4010 series display terminals.

  - *Chapter 15, Administrative Information*--Is a reference guide for system administrators. Specific information is contained about directory structure, installation, makefiles, hardware requirements, and miscellaneous facilities of the graphics package.

- *Supplementary Information and Reference Tools*—This part is organized into six appendices, a glossary, and an index. This material contains additional information that you may find useful in learning about the ICON/UXV system. The appendices are:

  - *Appendix A, Selected ICON/UXV System Documentation*—Lists additional ICON/UXV system documentation that enhances or elaborates on the information presented in this guide. This appendix gives document titles, reference numbers, and information on how to obtain the documents.

  - *Appendix B, File System Organization*—Illustrates how information is stored in the ICON/UXV operating system.

  - *Appendix C, Summary of ICON/UXV System Commands*—Describes, in alphabetical order, each ICON/UXV system command discussed in this guide.

  - *Appendix D, Quick Reference to* **ed** *Commands*—Describes the commands used with the line editor (**ed**), first in alphabetical order, and then organized by topic, such as creating text, deleting text, and displaying text.

  - *Appendix E, Quick Reference to* **vi** *Commands*—Describes the commands used with the screen editor (**vi**), first in alphabetical order, and then organized by topics, such as creating text, changing text, and cutting and pasting text.

  - *Appendix F, Summary of Shell Programming Ingredients*—Describes shell command language concepts and shows how to use shell programming language statements.

  Other sections in this part of the guide are:

  - *Glossary*—Defines technical words and terms used in this book.

  - *Index*—Gives an alphabetical listing of topics, together with the page numbers on which they appear in this guide.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *ICON/UXV Administrator Reference Manual*. Each reference of the form **name**(1) and **name**(6) refers to entries in the *ICON/UXV User Reference Manual*. All other references to entries of the form **name**(N), where a letter, refer to entry **name** in section **N** of the *ICON/UXV Programmer Reference Manual*.

The text of this guide was prepared using ICON/UXV system text editors described in this guide, formatted using the ICON/UXV System *DOCUMENTER'S WORKBENCH* **troff**, **tbl**, and **mm** macros, and produced on an IMAGEN, 800/5 laser printer operating under the ICON/UXV system.

* The visual editor is based on software developed by The University of California, Berkeley, California; Computer Service Division, Department of Electrical Engineering and Computer Science, and such software is owned and licensed by the Regents of the University of California.

Chapter 1

WHAT IS THE ICON/UXV SYSTEM?

PAGE

Chapter 1

WHAT IS THE ICON/UXV SYSTEM?

WHAT THE ICON/UXV SYSTEM IS

The ICON/UXV system is a set of programs, called software, that acts as the link between a computer and you, its user. The ICON/UXV system is designed to control the computer on which it is running so the computer can operate efficiently and smoothly and to provide you with an uncomplicated, efficient, and flexible computing environment.

ICON/UXV system software does three things:

- It controls the computer,
- It acts as an interpreter between you and the computer, and
- It provides a package of programs or tools that allows you to do your work.

The ICON/UXV system software that controls the computer is referred to as the operating system. The operating system coordinates all the details of the computer's internals, such as allocating system resources and making the computer available for general purposes. The nucleus of this operating system is called the kernel.

In the ICON/UXV system, the software that acts as a liaison between you and the computer is called the shell. The shell interprets your requests and, if valid, retrieves programs from the computer's memory and executes them.

The ICON/UXV system software that allows you to do your work includes programs and packages of programs called tools for electronic communication, for creating and changing text, and for writing programs and developing software tools.

Put simply, this package of services and utilities called the ICON/UXV system offers:

- A *general purpose* system that makes the resources and capabilities of the computer available to you for performing a wide variety of jobs or applications, not simply one or a few specific tasks.
- A computing environment that allows for an *interactive* method of operation so you can directly communicate with the computer and receive an immediate response to your request or message.
- A technique for sharing what the system has to offer with other users, even though you have the impression that the ICON/UXV system is giving you its undivided attention. This is called *timesharing.* The ICON/UXV system creates this feeling by allowing you and other users—*multiusers*—slots of computing time measured in fractions of seconds.

The rapidity and effectiveness with which the ICON/UXV system switches from working with you to working with other users makes it appear that the system is working with all users simultaneously.

- A system that provides you with the capability of executing more than one program simultaneously, this feature is called *multitasking.*

The ICON/UXV system, like other operating systems, gives the computer on which it runs a certain profile and distinguishing capabilities. But unlike other operating systems, it is largely machine-independent; this means that the ICON/UXV system can run on mainframe computers as well as microcomputers and minicomputers.

From your point of view, regardless of the size or type of computer you are using, your computing environment will be the same. In fact, the integrity of the computing environment offered by the ICON/UXV system remains intact, even with the addition of optional ICON/UXV system software packages that enhance your computing capabilities.

HOW THE ICON/UXV SYSTEM WORKS

After reading the past few pages, you know that the ICON/UXV system offers you a set of software that performs services--some automatically, some you must request. You also know that the system creates a certain environment in which you can use its software. But before you can ask the ICON/UXV system to do something, you need to know what it is capable of doing.

Look at *Figure 1-1.* It shows a set of layered circles in graduated sizes. Each circle represents specific ICON/UXV system software, such as:

- Kernel,
- Shell, and
- Programs/tools that run on command.

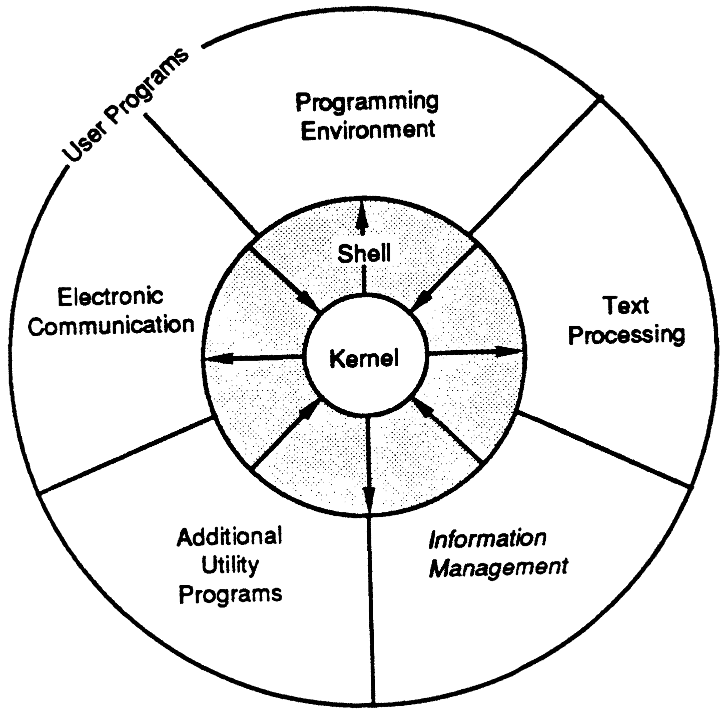


Figure 1-1. ICON/UXV system model

You should know something about the major components of ICON/UXV system software to communicate with the ICON/UXV system. Therefore, the remainder of this chapter introduces you to each component: the kernel, the shell, and user programs or commands.

Kernel

The heart of the ICON/UXV system is called the kernel. *Figure 1-2* gives an overview of the kernel's activities. Essentially, the kernel is software that controls access to the computer, manages the computer's memory, and allocates the computer's resources to one user, then to another. From your point of view, the kernel performs these tasks automatically. The details of how the kernel accomplishes this are hidden from you. This arrangement lets you focus on your work, not on the computer's.

On the other hand, you will become increasingly familiar with another feature of the kernel; this feature is referred to as the file system.

The file system is the cornerstone of the ICON/UXV operating system. It provides you with a logical, straightforward way to organize, retrieve, and manage information electronically. If it were possible to see this file system, it might look like an inverted tree or organization chart made up of various types of files *Figure 1-3*. The file is the basic unit of the ICON/UXV system and it can be any one of three types:

- An *ordinary file* is simply a collection of characters. Ordinary files are used to store information. They may contain text or data for the letters or reports you type, code for the programs you write, or commands to run your programs. In the ICON/UXV system, everything you wish to save must be written into a file.

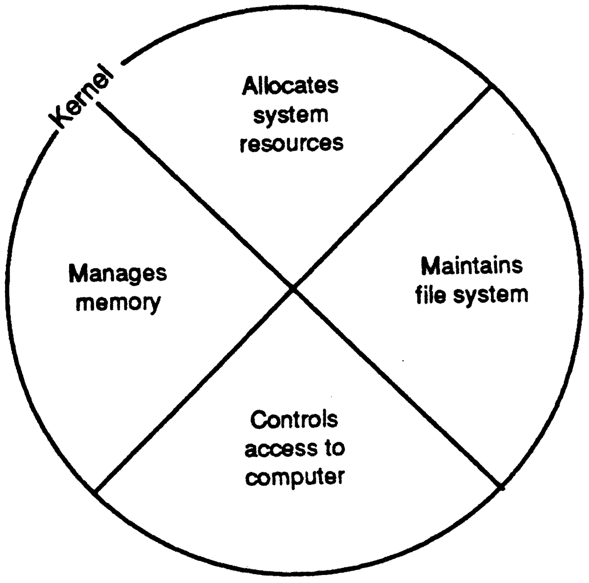


Figure 1-2. Functional view of kernel

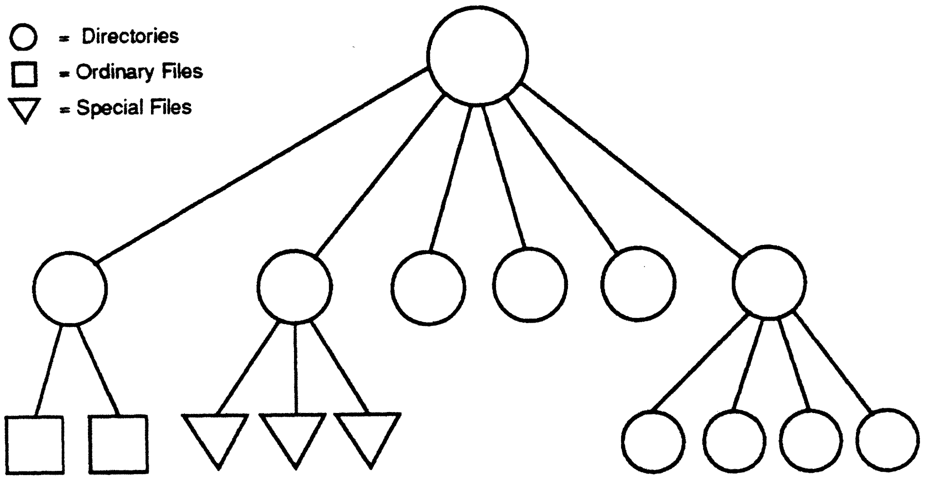


Figure 1-3. Branching directories and files give the ICON/UXV system its treelike structure

In other words, a file is a place for you to put information for safekeeping until you need to recall or use its contents again. You can add material to or delete material from a file once you have created it, or you can remove it entirely when the file is no longer needed.

- A *directory* is a file maintained by the operating system for organizing the treelike structure of the file system. A directory contains files and other directories as designated by you. You can build a directory to hold or organize your files on the basis of some similarity or criterion, such as subject or type.

  For example, a directory might hold files containing memos and reports you write pertaining to a specific project or client. Or a directory might hold files containing research specifications and programming source code for product development. A directory might hold files of executable code allowing you to run your computing jobs. Or a directory might contain files representing any combination of these possibilities.

- A *special file* represents a physical device, such as the terminal on which you do your computing work or a disk on which ordinary files are stored. At least one special file corresponds to each physical device supported by the ICON/UXV system.

In some operating systems, you must define the kind of file you will be working with and then use it in a specified way. You must consider how the files are stored since they can be sequential, random-access, or binary files. To the ICON/UXV system, however, all files are alike. This makes the ICON/UXV system file structure easy to use. For example, you need not specify memory requirements for your files since the system automatically does this for you. Or if you or a program you write needs to access a certain device, such as a printer, you specify the device just as you would another one of your files. In the ICON/UXV system, there is only one interface for all input from you and output to you; this simplifies your interaction with the system.

The source of the ICON/UXV system file structure is a directory known as root, which is designated with a slash (/). All files and directories in the file system are arranged in a hierarchy under root. Root normally contains the kernel as well as links to several important system directories that are shown in *Figure 1-4*:

/bin	Many executable programs and utilities reside in this directory.
/dev	This directory contains special files that represent peripheral devices, such as the console, the line printer, user terminals, and disks.
/etc	Programs and data files for system administration can be found in this directory.
/lib	This directory contains available program and language libraries.
/tmp	This directory is a place where anyone can create temporary files.
/usr	This directory holds other directories, such as **mail** (which further holds files storing electronic mail), **news** (which contains files holding newsworthy items), **rje** (which contains files needed to send data via something called the remote job entry communication link), and **games** (which contains files holding electronic games).

In summary, the directories and files you create comprise the portion of the file system that is structured and, for the most part, controlled by you. Other parts of the file system are provided and maintained by the operating system, such as **bin, dev, etc, lib, tmp** and **usr**, and have much the same structure on all ICON/UXV systems.

Chapter 3 shows how to organize a file system directory structure and how to access and manipulate files. *Chapter 4* gives an overview of ICON/UXV system capabilities. The effective use of these capabilities depends on your familiarity with the file system and your ability to access information stored within it. *Chapter 5* and *Chapter 6* are tutorials designed to teach you how to create and edit files to meet your computing and information

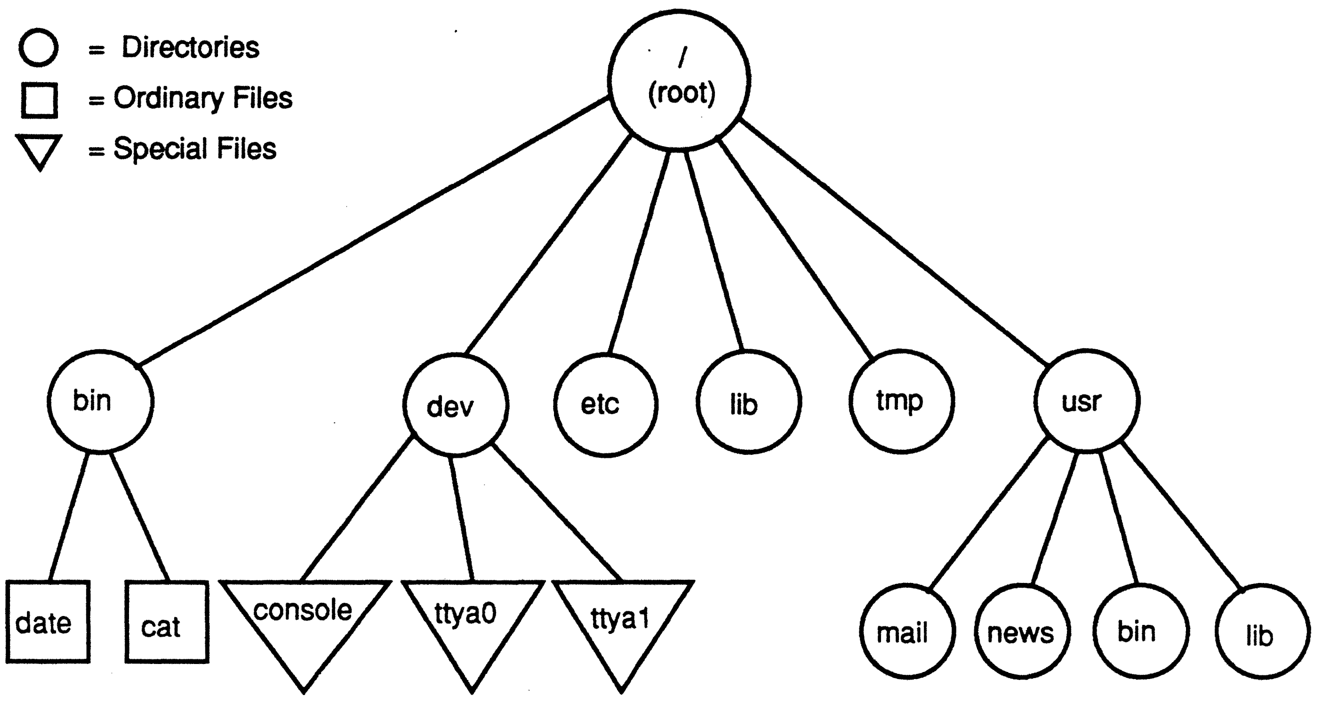


Figure 1-4. Sample of typical file system structure

management needs.

Shell

The shell is a unique ICON/UXV system program or tool that is central to most of your interactions with the ICON/UXV system. *Figure 1-1* illustrates how the shell works. The drawing shows the shell as a circle containing arrows pointing away from the kernel and the file system to the outer circle that contains programs and then back again. The arrows indicate that a two-way flow of communication is possible between you and the computer via the shell.

When you enter a request to the ICON/UXV system by typing on the terminal keyboard, the shell translates your request into language the computer understands. If your request is valid, the computer honors it and carries out an instruction or set of instructions. Because of its job as translator, the shell is called the command language interpreter.

As the command language interpreter, the shell can also help you to manage information. The shell's ability to manage information stems from the design of the ICON/UXV system. Each program in the ICON/UXV system is designed to do one thing well. In a sense, a ICON/UXV system program is a building block or module that you can use in tandem with other programs to create even more powerful tools.

In addition to acting as a command language interpreter, the shell is a programming language complete with variables and control flow capabilities.

A section of *Chapter 4* describes each of the shell's capabilities. *Chapter 7* teaches you how to use these capabilities to write simple shell programs called shell scripts and how to custom-tailor your computing environment.

Commands

A program is a set of instructions that the computer follows to do a specific job. In the ICON/UXV system, programs that can be executed by the computer without need for translation are called executable programs or commands.

As a typical user of the ICON/UXV system, you have many standard programs and tools available to you. If you also use the ICON/UXV system to write programs and to design and develop software, you have system calls, subroutines, and other tools at your disposal. And you have, of course, the programs you write.

This book introduces you to approximately 40 of the most frequently used programs and tools that you will probably use on a regular basis when you interact with the ICON/UXV system. If you need additional information on these or other standard ICON/UXV system programs, check the *ICON/UXV System User Reference Manual.* If you want to use tools and routines that relate to programming and software development, you should consult the *ICON/UXV System Programmer Reference Manual* and the *ICON/UXV System Support Tools Guide. Appendix A* provides you with information on how to obtain copies of these manuals.

The details contained in the two reference manuals may also be available via your terminal in what is called the *on-line* version of the ICON/UXV system reference manuals. This on-line version is made up of formatted text files that look exactly like the printed pages in the manuals. You can summon pages in this electronic manual using the command **man**, which stands for **man**ual page. If the electronic version of the manuals is available on your computer, the **man** command is documented in your copy of the *ICON/UXV System User Reference Manual.*

What Commands Do

The outer circle of *Figure 1-1* organizes ICON/UXV system programs and tools into general categories according to what they do. The programs and tools allow you to:

- *Process text.* This capability includes programs, such as, line and screen editors (which create and change text), a spelling checker (which locates spelling errors), and optional text formatters (which produce high-quality paper copies that are suitable for publication).

- *Manage information.* The ICON/UXV system provides many programs that allow you to create, organize, and remove files and directories.

- *Communicate electronically.* Several programs, such as **mail**, provide you with the capability to transmit information to other users and to other ICON/UXV systems.

- *Use a productive programming and software development environment.* A number of ICON/UXV system programs establish a friendly programming environment by providing ICON/UXV-to-programming-language interfaces and by supplying numerous utility programs.

- *Take advantage of additional system capabilities.* These programs include graphics, a desk calculator package, and computer games.

How Commands Execute

Figure 1-5 gives a general idea of what happens when the ICON/UXV system executes a command.

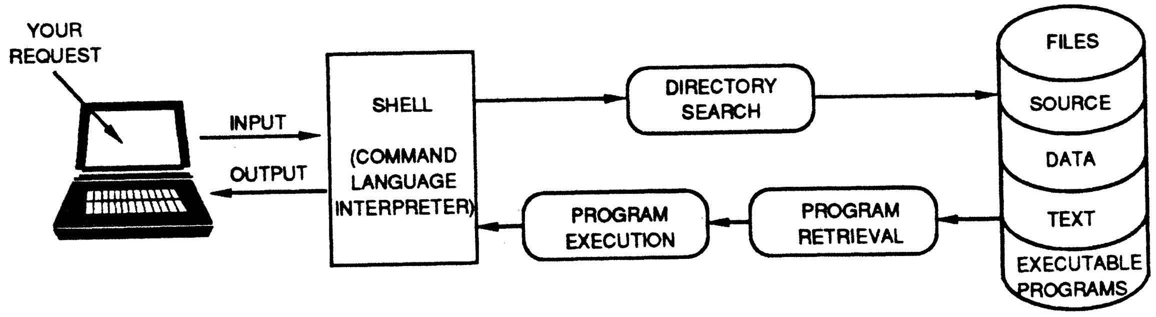


Figure 1-5. Flow of control between you and computer when you request program to run

When the shell signals it is ready to accept your request, you type in the command you wish to execute on the keyboard. The command is considered input, and the shell searches one or more directories to locate the program you specified. When the program is found, the shell brings your request to the attention of the kernel. The kernel then follows the program's instructions and executes your request. After the program runs, the shell asks you for more information or tells you it is ready for your next command.

This is how the ICON/UXV system works when your request is in a format that the shell understands. The structure that the shell understands is called a command line. *Chapter 3* explains what you need to know about the command line so you can request a program to run.

This chapter has outlined some basic principles of the ICON/UXV operating system and explained how they work. The following chapters will help you begin to apply these principles according to your computing needs.

Chapter 2

BASICS FOR ICON/UXV SYSTEM USERS

PAGE

Chapter 2

BASICS FOR ICON/UXV SYSTEM USERS

GETTING STARTED

There are general rules and guidelines with which you should be familiar before you begin to work on the ICON/UXV operating system. For example, you need information about your terminal and how to use its keyboard and about how to begin and end a computing session.

This chapter acquaints you with these rules and guidelines and presents you with information to help to make your first encounter with the ICON/UXV operating system understandable and to lay the groundwork for future computing sessions. Since the best way to learn about the ICON/UXV operating system is to use it, this chapter helps to get you started by providing examples of how to use these rules and guidelines to establish contact with the ICON/UXV operating system and to respond to its requests and prompts.

For your convenience, an outline of a terminal display screen is used to set off examples of interactions between you and the ICON/UXV operating system. These examples apply regardless of the type of terminal you use. Inside the screen, what the ICON/UXV operating system prompts and its responses are printed in *italic*. The commands you type in response to the system prompts and your other input and data are printed in **bold**face type. These include the commands you type that do not appear on the screen (such as, a carriage return), which are enclosed in angle brackets < >. The following screen summarizes these conventions.

italic(ICON/UXV system prompts and responses)

bold(Your commands)

<>(Your commands or parts of commands that do not appear on the screen)

Without further ado, let's begin.

To establish contact with the ICON/UXV operating system, you need:

- A terminal,
- An identification name, called a login name, by which the ICON/UXV operating system recognizes you as one of its authorized users,
- A password with which the ICON/UXV operating system double-checks and verifies your identity after you log in and before it allows you to use its resources, and
- The telephone number to the ICON/UXV operating system to which your login name is assigned if your terminal is not directly connected or wired to the computer.

ABOUT THE TERMINAL

A terminal is an input/output device: through it you input a request to the ICON/UXV operating system and the system, in turn, outputs a response to you. The terminal is equipped with a keyboard, a monitor or display unit (much like the screen on a television set), a control unit, and a link that allows it to communicate with the computer (*Figure 2-1*).

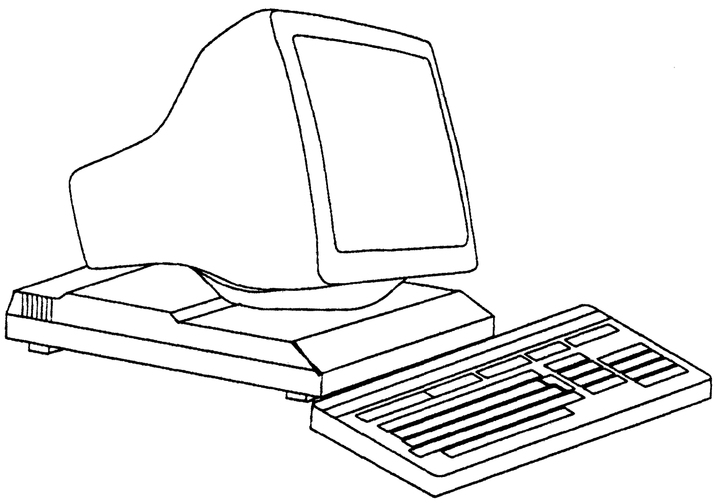


Figure 2-1. Video display terminal (DT1200™);

These terminals differ in how they monitor or display input/output. The video display terminal uses a display screen, whereas the printing terminal uses continuously fed paper.

™ Trademark of ICON International

Required Terminal Settings

Regardless of the type of terminal you use, you must set it up or configure it in a certain way to insure proper communication with the ICON/UXV operating system.

If you have not set terminal options before, you might feel more comfortable seeking help from someone who has. Or you can, of course, be adventurous.

How you configure a terminal depends on the type of terminal that you are using. Some terminals are configured with switches, whereas other terminals are configured directly from the keyboard using a set of function keys. To determine how to configure your terminal, consult the owner's manual provided by the manufacturer.

Following is a list of configuration checks to be performed on any terminal before attempting to establish contact with the ICON/UXV operating system.

- Turn on the power.
- Set the terminal to ON-LINE or REMOTE operation. This setting insures that the terminal is under direct control of the computer.
- Set the terminal to FULL DUPLEX mode. The full duplex mode insures two-way communication or input/output between you and the ICON/UXV operating system.
- If your terminal is not directly connected or hard wired to the computer, make sure the acoustic coupler or data phone set you are using is set to the FULL DUPLEX mode.
- Set character generation to LOWERCASE. If the terminal, however, generates only uppercase letters, the ICON/UXV operating system will accommodate it by printing everything that transpires during the computing session in uppercase letters.
- Set the terminal to NO PARITY.
- Set the speed or rate at which the computer communicates with the terminal. This rate of communication is called the baud rate. Typical terminal speeds are 30 and 120 characters per second or 300 and 1,200 baud, respectively. Occasionally, speeds such as 240, 480, and 960 characters per second or 2,400, 4,800, and 9,600 baud, respectively, are available.

Keyboard Characteristics

If you have seen or had some experience with a typewriter, the keyboard shown in *Figure 2-2* should look somewhat familiar.

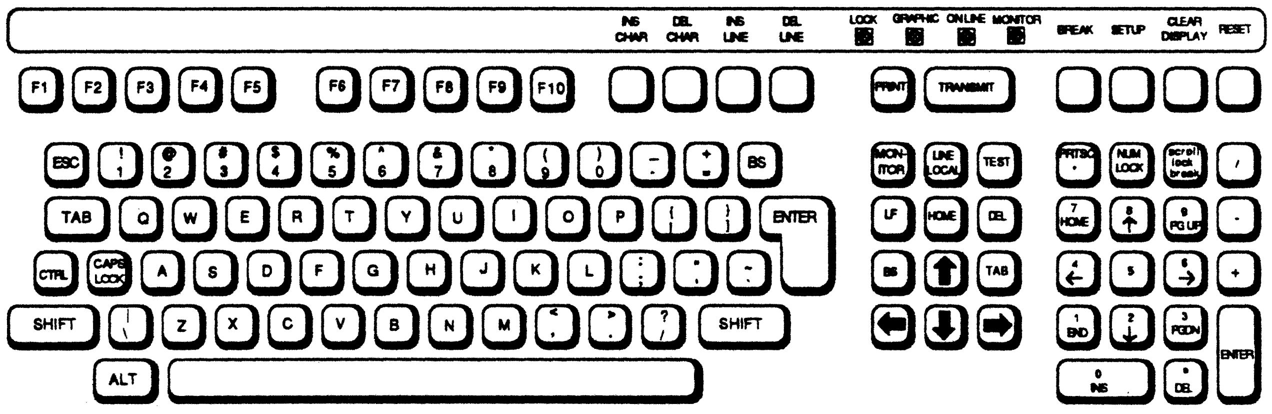


Figure 2-2. Example of keyboard layout (DT1200)

Its keys correspond to:

- Letters of the English alphabet **a** through **z** and **A** through **Z** when you are holding down a shift key,
- Numeric characters 0 through 9,
- A variety of symbols, such as ! @ # $ % ^ & ( ) _ − + = ~ ` { } [ ] \ : ; " ' < > , ? /
- Words, such as RETURN and BREAK, and abbreviations, such as DEL (delete), CTRL (control), and ESC (escape).

Many of the keys corresponding to symbols, words, and abbreviations have been added to the keyboard layout and the placement of these characters or symbols on a keyboard may vary from terminal to terminal.

Consequently, there is not a truly standard layout for terminal keyboard characters. There is, however, a standard set of characters that keyboards have, consisting of 128 characters, called the ASCII character set. ASCII is pronounced " *as kee*" and is the abbreviation for American Standard Code for Information Interchange. When you depress a key or combination of keys, the appropriate ASCII code is sent to the computer for translation from the alphabetic and numeric characters that we understand to electronic signals that the computer can decode.

Typing Conventions

To interact effectively with the ICON/UXV system, you should be familiar with certain typing conventions. An example of a ICON/UXV typing convention is using lowercase letters when you issue commands. Other typing conventions require that you use specific characters to erase letters and delete lines, or combinations of characters to stop the ICON/UXV from printing output on your terminal monitor temporarily.

The next few pages introduce you to these conventions. *Table 2-1* lists these special characters, keystrokes, and their meanings for your quick reference.

Responding to the Command Prompt

The standard ICON/UXV operating system command prompt is the dollar sign, $. When the $ appears on your terminal monitor, it means that ICON/UXV is waiting for you to tell it to do something. Your response to the $ prompt is to issue commands followed by depressing the carriage return key, designated as <CR> throughout this guide.

The $ is the default value for the command prompt. *Chapter 7* explains how to change the default value to another prompt.

Correcting Typing Errors

You can correct typing errors in two ways providing you have not pressed <CR>. The # symbol allows you to erase previously typed characters on a line, and the @ sign allows you to delete the line on which you are working. The # and the @ characters are default values for character and line deletion, respectively.

Pressing the # key erases the character previously typed, whereas repetitive use of the # sign erases any number of characters back to the beginning of the line, but not beyond that. For example, typing

```
helo#lo
```

on your terminal keyboard is interpreted by the ICON/UXV operating system as "hello" correctly typed.

TABLE 2-1

ICON/UXV Typing Conventions

Key(s)	Meaning
$	System's command prompt (your cue to respond)
#	Erase a character
@	Erase or kill an entire line
BREAK*	Stop execution of a program or command
DEL*	Delete or kill the current command line
ESC*	Use with another character to perform specific function (called escape sequence) **OR** Use to indicate end of create mode when using screen editor (**vi**)
RETURN*	End a line of typing; designated as **<CR>**
Control d*	Stop input to system or log off; designated as **<^d>**
Control h*	Backspace for terminals without a backspace key; designated as **<^h>**
Control i*	Horizontal tab for terminals without a tab key; designated as **<^i>**
Control s*	Temporarily stops output from printing on screen; designated as **<^s>**
Control q*	Resumes printing after typing **<^s>**; designated as **<^q>**

NOTE: All control characters are sent by holding down the control key and pressing the appropriate letter.

* Nonprinting characters.

To delete the entire line on which you are working, press the @ key. When you do, the ICON/UXV system moves you to the beginning of the next line.

If you want to use the # or the @ characters literally, that is, you would like a file to contain the line

Only one # appears on this sheet of music.

or

I purchased three books @ $15.75 per book.

you would have to press the backslash (\) key before pressing the # key. Otherwise, the # would erase the space after the word "one" and the line would print as

Only one appears on this sheet of music.

If you press the @ key without first pressing the \ key while typing the second example, the @ would erase the entire line. On the other hand, the leading \ removes the special meaning attached to characters like # and @ so that they can be understood literally by the computer.

Typing Speed

After the $ appears on your terminal monitor you can type as fast as you want, even during periods when the ICON/UXV system is responding to or executing a command. The printout on your terminal monitor will appear garbled because your input is intermixed with the system's output. The ICON/UXV system, however, has what is referred to as read-ahead capability, which allows it to separate input from output and to respond to your command properly.

With read-ahead capability, the ICON/UXV system stores your next request while the system is outputting information on your terminal monitor in response to a previous request.

Stopping a Command

If you wish to stop the execution of a command, simply depress the BREAK or DEL key. In turn, you will receive the $ prompt indicating that the ICON/UXV system terminated the running of the program and is ready to accept your next command.

Using Control Characters

Locate the control key on your terminal keyboard. The key may be labeled CTRL or CONTROL and is probably to the left of the A key or below the Z key. The control character is used in combination with other keyboard characters to initiate a physical controlling action across a line of typing, such as backspacing or tabbing. In addition, some control characters define ICON/UXV-system-specific commands, such as temporarily halting output from printing on a terminal monitor.

Type a control character by holding down the CTRL key and depressing an appropriate alphabetic key. Control characters do not print on the terminal when typed. In this book, control characters are designated with a preceding carat (^), such as <^s> for control s, to help identify them.

Let's take a look at the capabilities of the control character combinations you will be using regularly when working with the ICON/UXV system.

Temporarily Stopping Output. At times, you may wish to stop the ICON/UXV system temporarily from printing output on your terminal monitor. This could surely be the case when you wish to keep information from rolling off the screen monitor on a video display terminal. If you type <^s>, printing of output ceases; typing <^q> causes the printing to resume.

Terminating a Computing Session. When you have completed a session with the ICON/UXV operating system, you should type <^d>. This is the recommended way to log off the system and is described in detail later in this chapter.

Additional Control Character Capabilities. The ICON/UXV system furnishes other control character capabilities. For instance, if your terminal keyboard does not have a backspace key, typing <^h> gives you a backspace. Typing <^i> gives you a tab key if your terminal is set properly. (Refer to the section entitled *Possible Problems When Logging In* for information on how to set the tab key.)

After you configure the terminal and survey its keyboard, you are ready to establish communication with the ICON/UXV system if you have a login name.

OBTAINING A LOGIN NAME

Generally speaking, a log contains a record of information or data that notes a series of events or measures progress or performance.

The ICON/UXV system procedure for logging in is based on this idea. When you attempt to establish contact with the system, the ICON/UXV system verifies that you are an authorized user. If you pass the system's security checks, the ICON/UXV system allows you to log in. After you are logged in, the system maintains a record of the resources you use, the way in which you use them, and for how long. This log helps the people who manage and maintain the system by giving them complete user and resource allocation information.

To receive a login name, set up a ICON/UXV system account through your local system administrator or the person in charge of your ICON/UXV system installation. When the account is approved you should receive notification of your login name and the telephone number of the system to which your login is assigned.

Your login name is determined by local practices. Possible examples are your last name, your nickname, or a ICON/UXV system account number. Typically, a login name is three to eight characters in length. It can contain any combination of alphanumeric characters, as long as it starts with a letter. It cannot, however, contain any symbols. According to these rules, the following examples are legal ICON/UXV system login names: *starship*, *mary2*, and *jmrs*.

ESTABLISHING CONTACT WITH THE ICON/UXV SYSTEM

When you attempt to contact the ICON/UXV system, you will typically be using a terminal that is directly wired to a computer or a terminal that communicates with the system via a telephone connection.

If your terminal has a direct-wired connection, turn on the power and the message *login* should appear on the upper left side of the screen monitor or paper display.

Login Procedure

When the connection is made and the ICON/UXV system prompts for your login name, type in your login name and depress <CR>. In the following examples, *starship* is the login name.

```
login: starship<CR>
```

Remember to type in lowercase letters. If you use uppercase letters, the ICON/UXV system will also use uppercase letters until you log out and log in again.

Password

After typing in your login name, the ICON/UXV system prompts you for your password. In a typical session, you would simply type in your password followed by <CR>. For security reasons, the ICON/UXV system will not print (echo) your password on the terminal monitor.

If both your login name and password are acceptable to the ICON/UXV system, the system prints newsworthy messages for users. These items might include details about a new system tool or furnish a schedule for system maintenance. The news items are followed by the ICON/UXV system command prompt, which is the $ symbol.

Your terminal monitor should look something like the one that follows when you complete the login sequence successfully:

```
login: starship<CR>
password:
ICON/UXV system news
$
```

If you made a typing mistake that you did not correct before depressing <CR>, the ICON/UXV system displays the message *login incorrect* on your terminal monitor and asks you to try again by printing the login prompt. It is also possible that your communication link with the ICON/UXV system might be dropped in which case you would have to try to log in again.

```
login: ttarship<CR>
password:
login incorrect
login:
```

If you have never logged into the ICON/UXV system, your login procedure will differ somewhat from the typical one just described. This is because as a first-time user you were probably assigned a temporary password when your system account was set up and the system will not allow you to access its resources until you choose a new one.

This extra step maintains a security requirement, which is that you choose a password for your exclusive use. Protection of system resources and your personal files depends on you keeping the password you select private.

The actual procedure you will follow is determined according to administration procedures at your computer installation site. A typical example of what you might be expected to do if you have a new ICON/UXV system account and you are logging in for the first time follows.

1. The ICON/UXV system displays the login prompt when you establish contact with it. You should type in your login name followed by **<CR>**.

2. When the ICON/UXV system prints the password prompt, you should type in your temporary password and depress **<CR>**.

3. At this point, the system tells you the temporary password has expired and that it is time to select a new one.

4. The ICON/UXV system asks you to input the old password again. Type in your temporary password.

5. The system prompts you to input your new password. Type in the password you choose.

   The password you select is usually six to eight characters in length and contains at least one numeric character. In addition, you can also use special characters. Examples of valid passwords are: *mar84ch*, *JonathOn*, and *BRAV3S*.

   The ICON/UXV system you are using may have different requirements to consider when choosing a password. Ask another system user or contact the system administrator if you are not sure of the specifics.

6. For verification, the system requests that you re-enter your new password. Type in the new password once again.

   This is a valuable check for you and the ICON/UXV system since a password is not printed on the terminal monitor.

7. If you do not re-enter the new password exactly as you typed it the first time, the system tells you that the passwords do not match and asks you to try the procedure again. On some systems, however, the communication link may be dropped if you do not re-enter the password exactly as you typed it the first time. If this is the case, you must begin the login procedure again.

   When the passwords match, the system displays the **$** command prompt.

The following screen summarizes this procedure for first-time ICON/UXV system users.

```
login: starship            <CR>
password:                  <CR>
Your password has expired.
Choose a new one.
Old password:              <CR>
New password:              <CR>
Re-enter new password:     <CR>
ICON/UXV system news
$
```

External Security Code

If you are able to access the ICON/UXV system from outside your computer installation site, you may need additional information to establish contact with the ICON/UXV system, such as a special telephone number or another security code. To determine if this feature is available to you, contact your system administrator.

Possible Problems When Logging In

A terminal usually behaves predictably providing you have configured it properly. Sometimes, however, it may act peculiarly. For example, each character you type may appear twice on the terminal monitor or the carriage return may not work properly.

Some problems can be corrected by simply logging off the system and logging on again. If logging on a second time does not remedy the problem, you should first check the following and try logging in once again:

- *Keyboard*—Keys that are marked CAPS, LOCAL, BLOCK, and so on should not be enabled, that is, in the locked position. You can usually disable these keys simply by depressing them.

- *Data phone set or modem*—If your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly specified.

- *Switches*—Some terminals have several switches that must be set to be compatible with the ICON/UXV system. If this is the case with the terminal you are using, make sure they are set properly.

Refer to the section *Required Terminal Settings* in this chapter if you need information to verify the terminal configuration. If you need additional information about the keyboard, terminal, and data phone or modem, check the owner's manuals for the equipment.

Table 2-2 presents a list of procedures you can follow to detect, diagnose, and correct some problems you may experience when trying to establish contact with the ICON/UXV system. If none of the possibilities covered in the table helps you, contact the system administrator or the person in charge of the ICON/UXV installation at your location.

TABLE 2-2
Troubleshooting Problems When Logging in*

Problem†	Possible Cause	Action/Remedy
Stream of meaningless characters when logging in	UNIX system attempting to communicate at wrong speed	Depress RETURN or BREAK key
Input and output is printed in uppercase letters	Terminal configuration includes UPPERCASE setting	Log off, set character generation to LOWERCASE, and log in again
Input is printed in UPPERCASE letters, output in LOWERCASE	Key marked CAPS or CAPS LOCK is locked or enabled	Depress the CAPS or CAPS LOCK key to disable setting
Input is printed (echoed) twice	Terminal is set to HALF DUPLEX mode	Change setting to FULL DUPLEX mode
Tab key does not work properly	Tabs are not set to advance to next	Type **stty -tabs**‡
Communication link cannot be established in spite of receiving high pitched tone when dialing in	Terminal is set to LOCAL or OFF-LINE mode	Set terminal to ON-LINE operation and try logging in again
Communication link between terminal and UNIX system is repeatedly dropped on logging in	Terminal is set to LOCAL or OFF-LINE mode	Call system administrator

* Numerous problems can occur if your terminal is not configured properly. To eliminate these possibilities before attempting to log in, perform the configuration checks listed on page 2-4.

† Some problems may be specific to your terminal, data set, or modem, check the owner's manual for this equipment if suggested actions do not remedy the problem.

‡ Typing **stty -tabs** corrects tab setting only for your current computing session. To insure correct tab setting for all sessions, add the line **stty -tabs** to your profile (see *Chapter 7*).

Simple Commands

When the $ command prompt is displayed on your monitor, you know that the ICON/UXV system recognizes you as an authorized user. Your response to the $ command prompt is to request ICON/UXV system programs to run.

Type in the command **date** and press **<CR>** after the command prompt. When you do this, the ICON/UXV system retrieves the **date** program and executes it. As a result, your terminal monitor should look something like the following.

```
$ date<CR>
Wed Oct 12 09:49:44 CDT 1983
$
```

As you can see, the ICON/UXV system prints the date and the time. In this example, the CDT stands for Central Daylight Time. Your terminal monitor will display the appropriate time for your geographical location.

Now type the command **who** and depress **<CR>**. Your screen will look something like this.

```
$ who<CR>
starship            tty00Oct 12  8:53
mary2 tty02         Oct 12  8:56
acct123             tty05Oct 12  8:54
jmrs    tty06       Oct 12  8:56
$
```

The **who** command lists the login names of everyone currently working on your system. The tty designations refer to the names of the special files that correspond to the terminals on which you and other users are currently working. The login date and time for each are also given.

Logging Off

When you have completed a session with the ICON/UXV system, you should type **<^d>** after the $ command prompt. (Remember that control characters such as the **<^d>** are typed by holding down the control key and depressing the appropriate alphabetic key.) Since they are nonprinting characters, they do not appear on the terminal monitor. In a few seconds, the ICON/UXV system should display the login message again. This indicates you have logged off successfully and someone else can log in at this time. Your terminal monitor should look like the one that follows.

```
$ <^d>
login:
```

It is strongly recommended that you log off the system using <^**d**> before turning off the terminal or hanging up the phone. It is the only way to assure you have been logged off the ICON/UXV system.

Chapter 3

USING THE FILE SYSTEM

PAGE

Chapter 3

USING THE FILE SYSTEM

INTRODUCTION

To use the file system effectively you must be familiar with its structure, know something about your relationship to this structure, and understand how the relationship changes as you move around within it. Reading this chapter serves as preparation to use this file system.

The first ten or so pages should help to give you a working perspective of the file system. These pages contain information on the makeup of the file system and on how you fit into its organization. The remainder of the chapter introduces you to a number of ICON/UXV commands. Some you can use to build your own directory structure, whereas others allow you to access and manipulate the subdirectories and files you organize within it. And others still allow you to examine the contents of other directories in the system that you have permission to look at or to use.

Each command is discussed in a separate subsection in a way that will allow you to use it effectively. Many of the commands presented in this section have additional, sophisticated uses; these, however, are left for more experienced users and are described in other ICON/UXV documentation. You can choose to read these sections in the order in which they are presented in the text or you can opt to read about the commands and their capabilities in the order that best suits your interests and purpose. Nevertheless, all the commands presented are basic to using the file system efficiently and easily. It is recommended that you read through them thoroughly and then try them out. Before viewing how the file system is structured, however, let's take a look at the structure of a command.

For the ICON/UXV operating system to understand your intentions when using commands, you must take care to see that you input commands using the correct format, called the command line syntax. The command line syntax provides a procedure for ordering elements in a command line. It serves the same purpose as putting words in a certain sequence or order so that you can meaningfully express your ideas and thoughts to others. Without sentence structure, people would have difficulty interpreting what you mean. Similarly, without command line syntax, the ICON/UXV shell cannot interpret your request.

Command line syntax consists of one or more of the following elements separated by a blank or blanks and followed by pressing the carriage return **<CR>** key:

command option(s) argument(s)

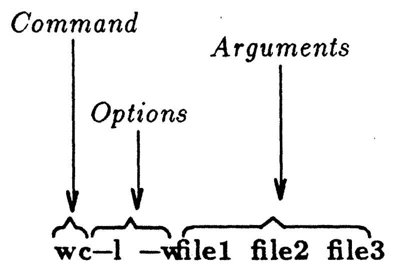
where

command is the name of the program you wish to run, *option* modifies how the command runs, and

argument specifies data on which the command is to focus or operate (usually a directory or file name).

A command line can simply contain a command name followed by **<CR>**, or it can list options and/or arguments in addition to the command. If you specify options and arguments on the command line, you must separate them with at least one blank. Blanks can be typed by pressing the space bar or the tab key. If a blank is part of the argument name, enclose the argument in double quotation marks, for example, "sample 1".

Some commands allow you to specify multiple options and/or arguments on a command line. Consider the following command line:



In this example, **wc** is the name of the command and two options **–l** and **–w** have been specified. (The ICON/UXV system usually allows you to group options such as these to read **–lw** if you prefer.) In addition, three files--*file1, file2,* and *file3*--are specified as arguments. Although most options can be grouped together, arguments cannot.

The following examples show the proper sequence and spacing in command line syntax:

Incorrect	**Correct**
wcfile	wc file
wc–lfile	wc –l file
wc –l w file	wc –lw file *or* wc –l –w file
wc file1file2	wc file1 file2

You can refer back to the ground rules on command line syntax as you read and work through the chapter.

HOW THE FILE SYSTEM IS STRUCTURED

The file system is comprised of a set of directories, ordinary files, and special files. These components provide you with a way to organize, retrieve, and manage information electronically. *Chapter 1* introduced you to directories and files, but let's review what they are before learning how to use them to tap the resources of the file system.

In general, a directory is a collection of files and other directories. Specifically, it contains the names of these files and directories. You can build a directory to organize the files you create on the basis of some similarity. An ordinary file is a collection of characters that is stored on a disk. Such a file may contain text for a status report you type or code for a program you write. Any information you wish to save must be written into a file. And a special file represents a physical device, such as your terminal.

The set of all the directories and files is organized into a treelike structure. *Figure 3-1* helps you to visualize this. It shows a single directory called *root* as the source of a sample file structure. By descending the branches that extend from root, several other major system directories can be reached. By branching down from these, you can, in turn, reach all the directories and files in the file system. In this hierarchy, files and directories that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many generations of files and directories, giving you the capability to organize your files in a variety of ways.

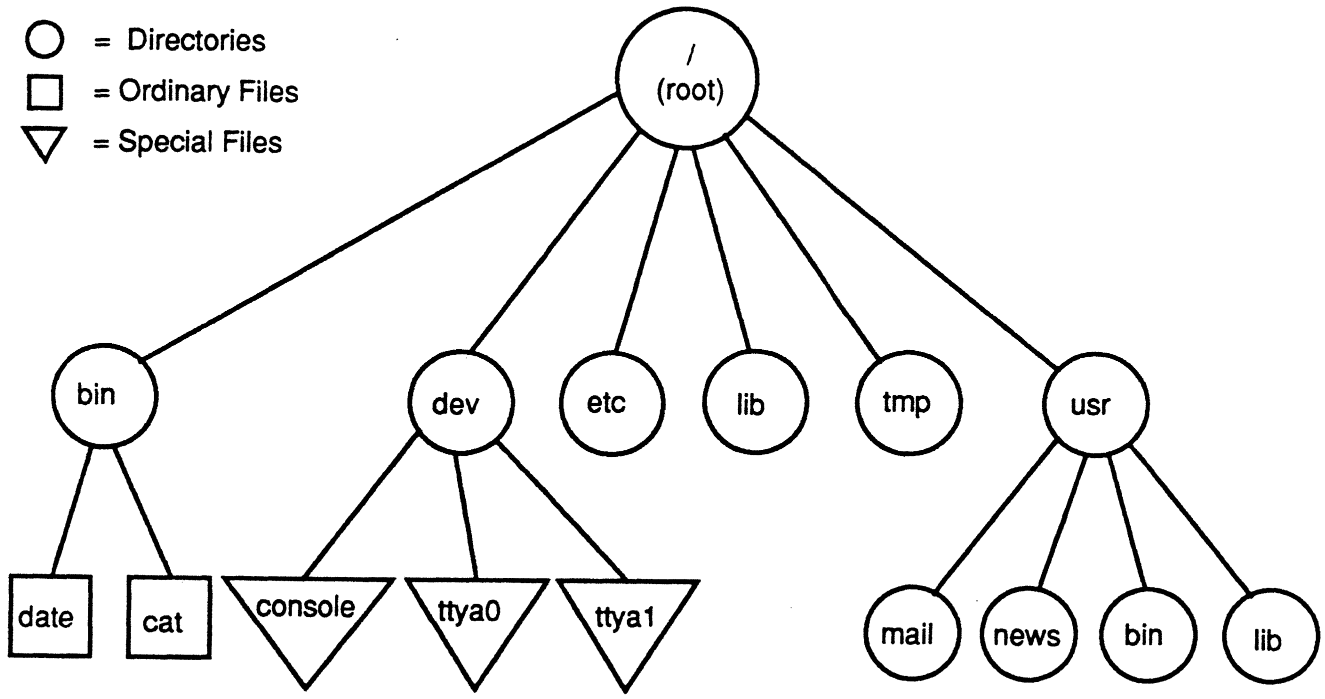


Figure 3-1. Sample file system

YOUR PLACE IN THE FILE SYSTEM STRUCTURE

When you are interacting with the ICON/UXV system, you will be doing so from a location in its file system structure. The ICON/UXV system automatically places you at a specific point in its file system every time you log in. From that point, you can move through the hierarchy to work in any of the directories and files you own and to access those belonging to others that you have permission to use.

The following sections describe your place in relation to the file system structure and how this relationship changes as you move through the file system.

Your Home Directory

When you successfully complete the login procedure, the ICON/UXV system positions you at a specific point in its file system structure called your login or home directory. The login name that was assigned to you when your ICON/UXV account was set up is usually the name of this home directory. In fact, every user with an authorized login name has a unique home directory in the file system.

The ICON/UXV system is able to keep track of all these home directories by maintaining one or more system directories that organize them. For example, let's say that the name of one of these system directories is *user1*, and that it contains the home directories of the login names *starship*, *mary2*, and *jmrs*. *Figure 3-2* shows you how a system directory like *user1* ranks in relation to the other important ICON/UXV directories you read about in *Chapter 1*.

Within your home directory, you can create files and additional directories (sometimes called subdirectories) to organize them, you can move and delete these files and directories, and you can control who can access your files and directories. You have full responsibility for everything you create in your home directory because you own it. Your home directory is a vantage point from which to view all the files and directories it holds. It is also a point from which to view the file system all the way up to root.

Your Working Directory

As long as you continue to work in your home directory, it is considered your current or working directory. If you move to another directory, that directory becomes your new working directory.

There is a ICON/UXV command called **pwd**, which stands for **p**rint **w**orking **d**irectory, that you can use to verify the name of the directory in which you are currently working. For example, if your login name is *starship* and you issue the **pwd** command in response to the first $ prompt after logging in, the ICON/UXV system should respond as follows:

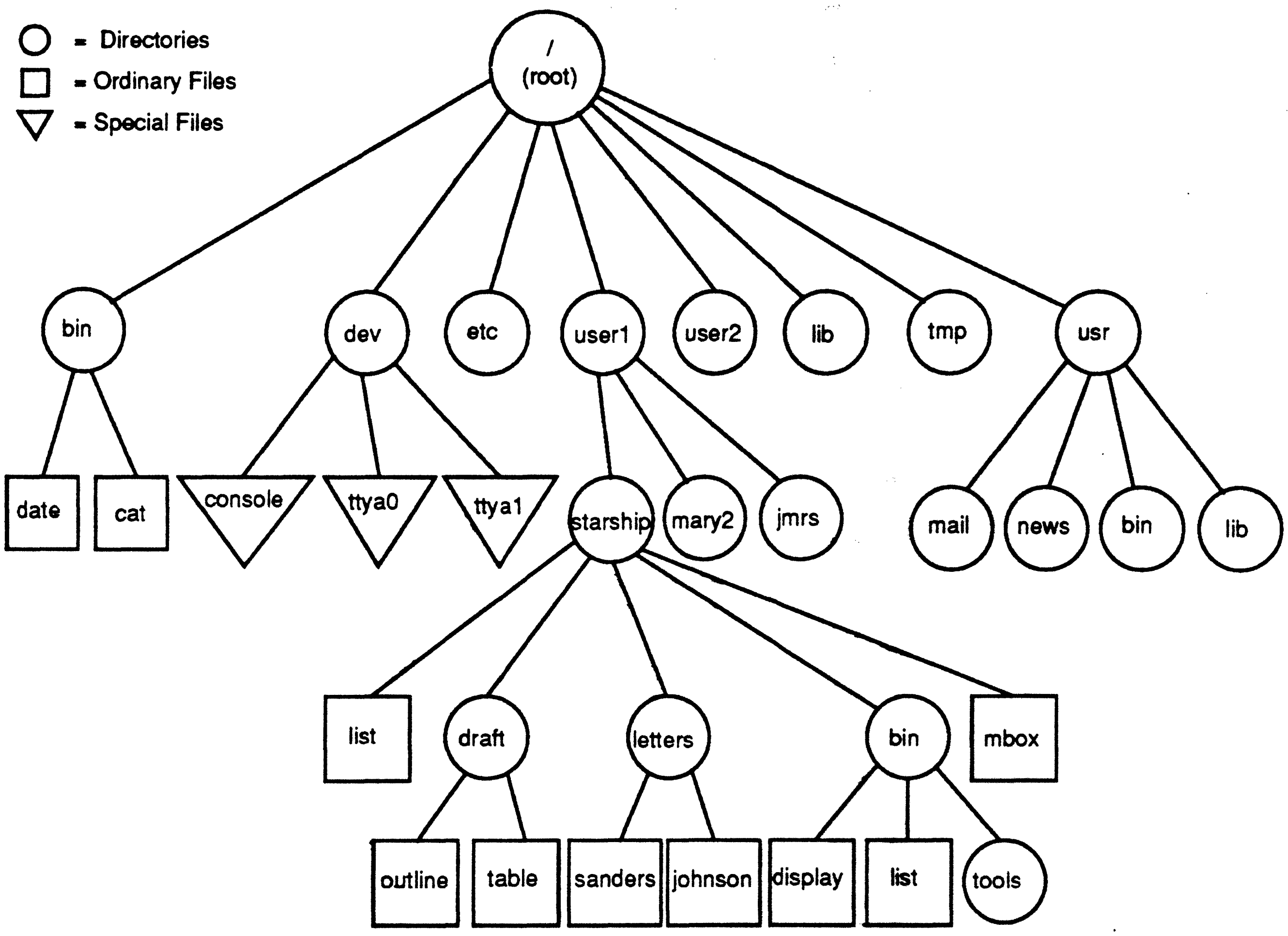


Figure 3-2. A directory that organizes home directories is equivalent to directories like bin and tmp in the file system

```
$ pwd<CR>
/user1/starship
$
```

The system reply indicates that your working directory is */user1/starship*. Technically, */user1/starship* is the full or complete name of the working directory. The name of a directory like */user1/starship* or a file is also referred to as a path name.

Printing the complete or full path name of your working directory in response to a **pwd** command is a courtesy that the ICON/UXV system extends to you. The full path name indicates your exact position in terms of the file system structure.

We will analyze and trace this path name in the next few pages so you can start to move around in the file system. For now, it is sufficient to say that what */user1/starship* tells you is that the root directory / (indicated by the leading slash in the line) contains the directory *user1*, which in turn contains the current working directory, which is *starship*. All other slashes in the path name are simply used to separate names of directories and files.

Remember, you are never more than issuing a **pwd** command away from determining where you are in the file system. Issuing the **pwd** command will be especially helpful if you try to read or copy a file and the ICON/UXV system tells you that the file you are trying to access does not exist. You may be surprised to find that you are in a different directory than you thought.

To provide you with a quick summary of what you can expect the **pwd** command to do, a recap of how to use it follows.

Command Recap

pwd - print full name of working directory

command	*options*	*arguments*
pwd	none	none

Description:	**pwd** prints the full path name of the directory in which you are currently working.
Remarks:	If the system responds with messages, such as, *cannot open directory* or *read error in directory*, there may be problems with the file system. Inform the system administrator.

Path Names

Every file and directory in the ICON/UXV system is identified by a unique path name. The path name tracks or indicates the location of the file or directory relative to the structure of the system. In addition to identifying the location of a file or directory in the file system structure, a path name provides directions to that file or directory. Knowing how to follow the directions the path name gives is your key to moving around the directory structure successfully.

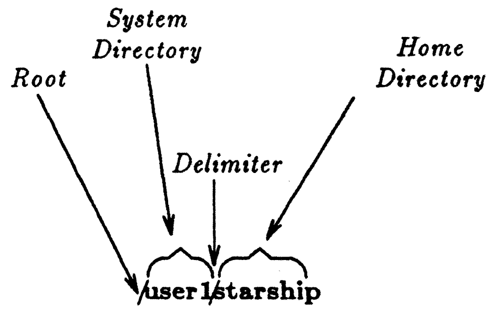
In the file system, there are two types of path names--full and relative. Let's take a closer look at both types.

Full Path Names

A full path name (sometimes called an absolute path name) gives you directions that take you from the root directory down through a unique sequence of directories that leads to a particular directory or file. You can use a full path name to reach any file or directory in the ICON/UXV system in which you are working. A full path name always starts at the root of the file system and its leading character is a / (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and where it can lead you, let's use the sample file system (*Figure 3-2*) and say that you are in the directory *starship*. If you issue the **pwd** command, the system responds by printing the full path name of your working directory, which is */user1/starship*.

We can analyze the elements of this path name using the following diagram.



where:

/ *(leading)*	= Root of the file system when it is the first character in the path name,
user1	= System directory one level below root in the hierarchy to which root points or branches,
/ *(subsequent)*	= Slash that separates or delimits the directory names, ***user1*** and ***starship***, and
starship	= Current working directory, which is also the home directory.

Now look at *Figure 3-3*, it traces the full path to */user1/starship* through the sample file system we are using.

Relative Path Names

A relative path name is the name of a file or directory that varies with relation to the directory in which you are currently working. From your working directory, you can move "down" in the file system structure to access files and directories you own or you can move "up" in the hierarchy through generations of parent directories to the grandparent of all system directories, the root. A relative path name begins with a directory or file name, with a **.** (dot), which is a shorthand notation for the directory in which you are currently located, or a **..** (dot dot), which is a shorthand notation for the directory immediately above your current working directory in the file system hierarchy. The **..** (dot dot) is called the parent directory of the one in which you are currently located, which is the current directory or **.** (dot).

For example, if you are in the home directory ***starship*** in the sample system and ***starship*** contains directories named ***draft***, ***letters***, and ***bin*** and a file named ***mbox***, the relative path name to any of these is simply its name, be it ***draft***, ***letters***, ***bin***, or ***mbox***. ***Figure 3-4*** traces the relative path name from ***starship*** to ***draft***.

Now, let's say the ***draft*** directory belonging to ***starship*** contains the files ***outline*** and ***table***. Then, the relative path name from ***starship*** to the file ***outline*** is written as ***draft/outline***.

Figure 3-5 traces this relative path. Notice that the slash in this path name separates the directory named ***draft*** from the file named ***outline***. Here, the slash is a delimiter that indicates that ***outline*** is subordinate to ***draft***; that is, ***outline*** is a child of its parent, ***draft***.

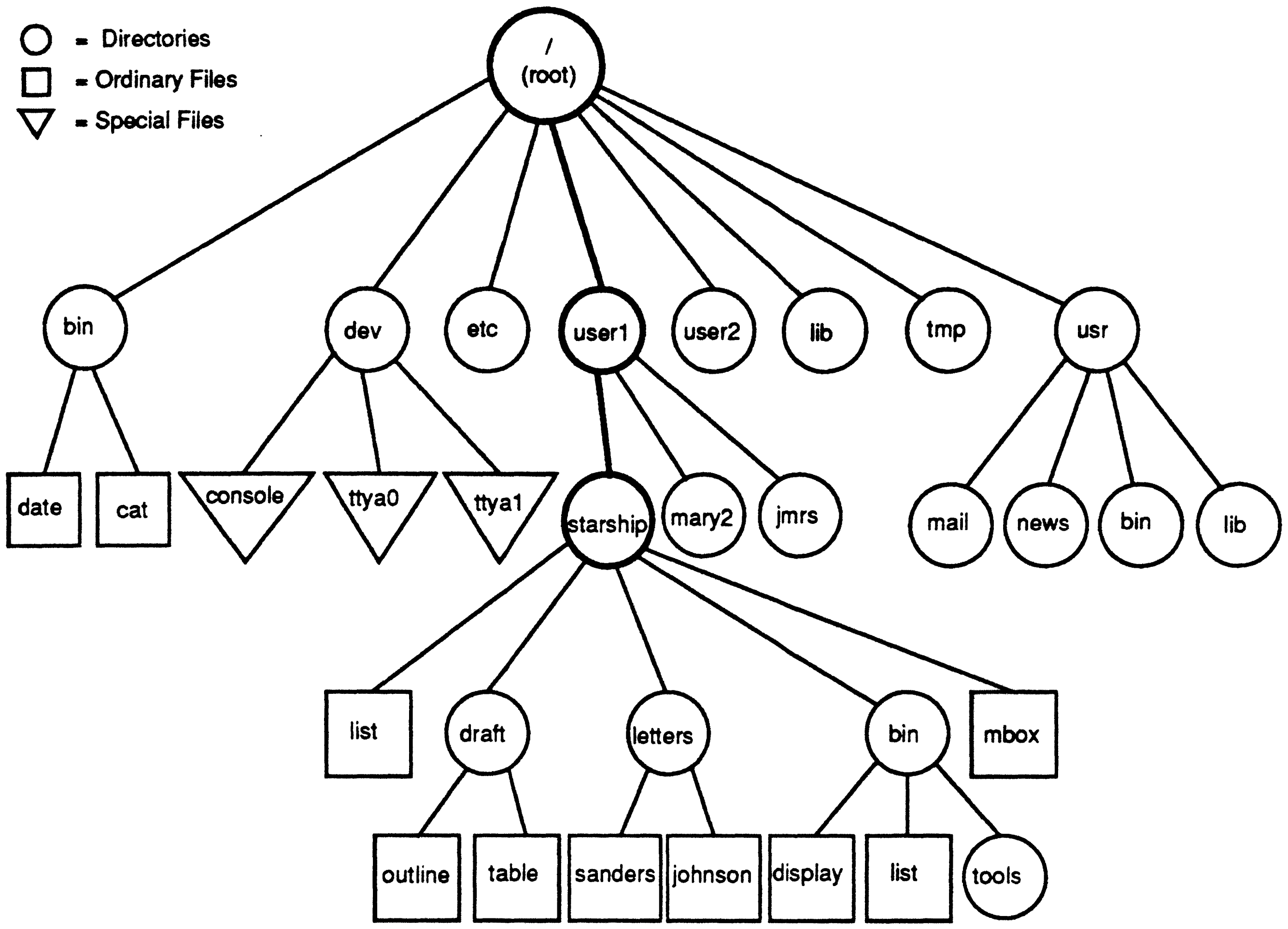


Figure 3-3. Heavy bold lines trace the full path name of the directory /user1/starship

Thus far, the discussion of relative path names covered how to specify names and directories of files that belong to, or are children of, your current directory--in other words, to descend the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of your working directory, you can use the **..** notation. This means that if you are in the directory named *draft* in the sample file system, **..** is the path name to *starship*, and **../..** is the path name to *starship*'s parent directory *user1*. From *draft*, you could also trace a path to the file *sanders* in the sample system by using the path name *../letters/sanders* (**..** brings you up to *starship*, then down to *letters*, and finally *sanders*).

Keep in mind that you can always use a full path name in place of a relative one.

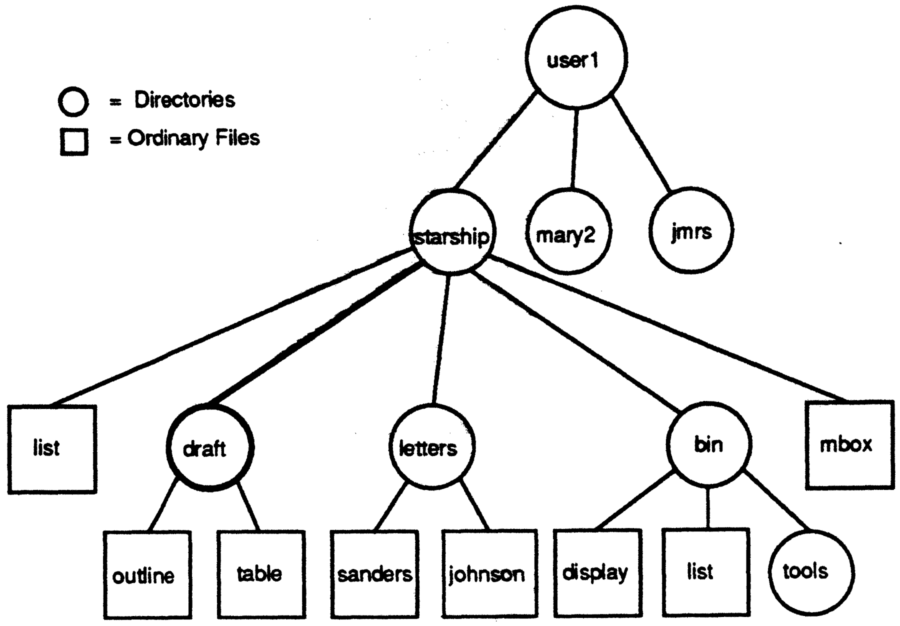


Figure 3-4. Relative path name for the draft directory is traced with heavy bold lines

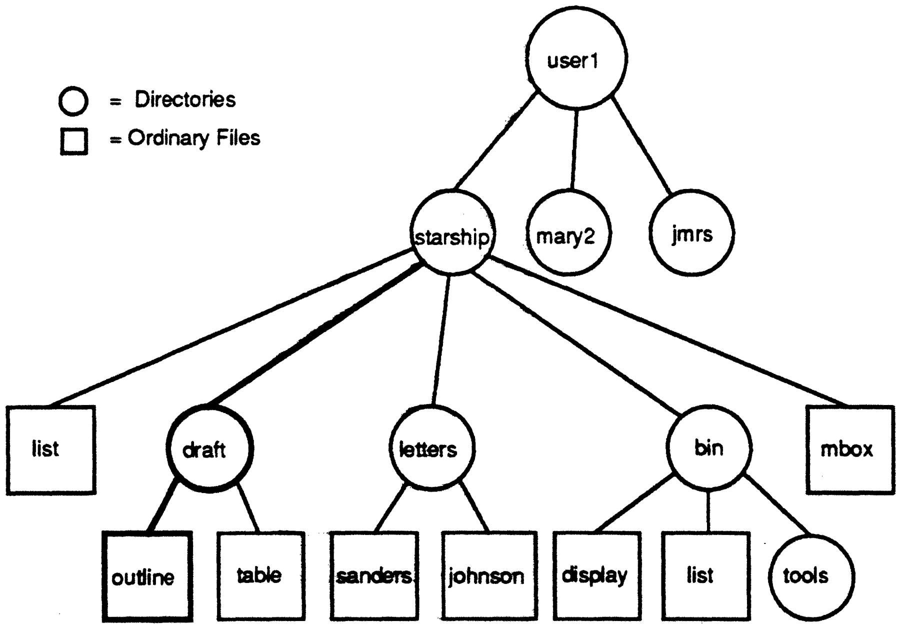


Figure 3-5. The relative path draft/outline is traced in bold lines

In summary, some examples of full and relative path names would be:

Path Name	Meaning
/	Full path name of the root directory for the file system.
/bin	Full path name of the *bin* directory that contains most executable programs and utilities.
/user1/starship/bin/tools	Full path name of the directory called *tools* belonging to the directory *bin* that belongs to the directory *starship* belonging to *user1* that belongs to root.
bin/tools	Relative path name to the file or directory *tools* in the directory *bin*. If the current directory is /, then the ICON/UXV system searches for */bin/tools*. But, if the current directory is *starship*, then the system searches the full path */user1/starship/bin/tools*.
tools	Relative path name of a file or directory *tools* in the working directory.

Knowing how to follow path names, such as in these examples, and move about in the file system is a skill tantamount to being able to read and follow a map when you are traveling in a new or unfamiliar place.

It might take some practice to move around in the file system with confidence. But this is to be expected when learning a new concept and the techniques to use it.

To give you a chance to try your hand at moving about in the system's structure, the remainder of the chapter introduces you to the ICON/UXV commands that make it possible for you to peruse the file system. If you lose track of where you are in the system's structure, use the **pwd** command to identify your location.

ORGANIZING A DIRECTORY STRUCTURE

This section introduces you to four ICON/UXV commands that make it possible for you to organize and use a directory structure. These commands and what you can expect them to do are as follows:

mkdir -- Allows you to create or make new directories and subdirectories within your current directory,

ls	--	Allows you to list the names of all the subdirectories and files in a directory,
cd	--	Provides you with the ability to change your location from one directory to another in the file system, and
rmdir	--	Lets you remove a directory when you no longer have a need for it.

All of the commands can be used with path names--full or relative--when organizing a directory structure and when moving to the directories and subdirectories you organize, as well as when navigating to directories in the file system that belong to others that you have permission to access. Two of the commands--**ls** and **cd**--can also be used without a path name.

Each of the commands is described more fully in the four sections that follow. In addition, a summary called a command recap is given for each command. The command recaps allow you to review quickly the command line syntax and the capabilities of each command.

Creating Directories *(mkdir)*

It is recommended that you create subdirectories in your home directory according to some logical and meaningful scheme to help you retrieve information you will keep in files. A convenient way to organize your files is to put all files pertaining to one subject together in a directory.

To create a directory, the ICON/UXV system provides you with the **mkdir** command, which stands for **make dir**ectory. In the sample file system, the *draft* subdirectory in the home directory *starship*, for example, may have been built by inputting the following while located in *starship*:

```
$ mkdir draft<CR>
$
```

The $ response to the **mkdir** command indicates that a directory named *draft* was successfully created.

Similarly, the other subdirectories named *letters* and *bin* were created with the same command, as indicated in the following screen:

```
$ mkdir letters<CR>
$ mkdir bin<CR>
$
```

All the subdirectories (*draft, letters, bin*) could have been created in one command with the same results, as the following screen shows:

```
$ mkdir draft letters bin<CR>
$
```

You can also move to a subdirectory you created and build additional directories if necessary and reasonable. When you build directories, or create files for that matter, you can name them anything you wish as long as you keep in mind the guidelines presented in the following list.

- The name of a directory (or file) can be from one to fourteen characters in length.
- All characters other than / are legal.
- Some characters are best avoided, such as a blank or space, a tab, or a backspace, and the following:

  @ # $ ^ & * ( ) ` [ ] \ | ; ' " < >

  If you use a blank or tab in a directory or file name, you must enclose the name in quotation marks on the command line.
- Avoid using the +, − or . as the first character in names.
- Uppercase and lowercase characters are distinct to the ICON/UXV system. For example, the directory or file named *draft* would not be the same as the directory or file named *DRAFT*.

Examples of legal directory or file names would be:

memo	MEMO	section2	ref:list
file.c	chap3+4	item1-10	outline

See the command recap that follows for a quick reference to **mkdir**'s capabilities.

Command Recap

mkdir - make a new directory

command	*options*	*arguments*
mkdir	none	**directoryname(s)**

Description: **mkdir** creates a new directory (subdirectory).

Remarks: The system returns the $ prompt if the directory is successfully created.

Listing the Contents of a Directory *(ls)*

All directories in the file system have information about the files and directories they contain, such as name, size, and the date last modified. You can obtain this information about what your working directory and other system directories contain by using the **ls** command.

The **ls** command, which stands for **list**, lists the names of the files and subdirectories of the directory you specify by path name. If you do not specify a path name, **ls** lists the names of files and directories in your working directory. To demonstrate how the **ls** command works, let's use the sample file system (*Figure 3-2*) once again.

You are logged into the ICON/UXV system and the shell responds to your **pwd** command with the line */user1/starship*. To display the names of files and directories in the working directory, you would type **ls<CR>**. After this sequence, your terminal should read:

```
$ pwd<CR>
$ /user1/starship
$ ls<CR>
bin
draft
letters
list
mbox
$
```

As you can see, the system responds by listing the names of files and directories in the working directory *starship* in alphabetical order. If the first character of any of the file or directory names was a number, or a capital letter, it would have been printed first.

Now, if you want to print the names of files and subdirectories in a directory other than your working directory without moving from your working directory, you should use the command format:

ls directoryname<CR>

where the directory name is the full or relative path name of the desired directory. This means that you can print the contents of *draft* while you are working in *starship* by inputting **ls draft<CR>**.

```
$ ls draft<CR>
outline
table
$
```

In the example, *draft* is a relative path name from *starship* to *draft*. By the same token, you could print the contents of the *user1* directory, which is the parent of the *starship* by typing:

```
$ ls ..<CR>
jmrs
mary2
starship
$
```

where .. is the relative path name from *starship* to *user1*. You could also list the contents of *user1* by typing **ls /user1<CR>** (since */user1* is the full path name from root to *user1*) and get the identical listing.

Similarly, you can list the contents of any system directory that you have permission to access using the **ls** command and a full or relative path name.

The **ls** command is particularly useful if you have a long list of files and you are trying to determine whether one of them exists in your working directory. For example, if you are in the directory *draft* and you wish to determine if the files named *outline* and *notes* are there, you can use the **ls** command as follows:

```
$ ls outline notes<CR>
outline
notes not found
$
```

The output on the terminal monitor shows that the system acknowledges the existence of *outline* by printing its name, but says that the file *notes* is not found.

By the way, the **ls** command will not print the contents of a file. If you wish to see what a file contains, you can use the **cat**, **pg**, or **pr** command, which are described in the section of this chapter entitled *Accessing and Manipulating Files.*

Frequently Used *ls* Options

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. There are more than a dozen available options for the **ls** commands. Of these, the **−a** and **−l** will probably be most valuable in your basic use of the ICON/UXV system. Refer to the *ICON/UXV User Reference Manual* for information and details on the other options.

Listing All Names in a File. Some important file names in your home directory begin with a **.** (dot), such as .profile, **.** (the current directory), and **..** (the parent directory). The **ls** command will not print these names unless you use the **−a** option in the command line. Thus, to list all files in your working directory *starship*, including those that start with a **.** (dot), type **ls −a<CR>**. The terminal should look something like this:

```
$ ls -a<CR>
.
..
.profile
bin
draft
letters
list
mbox
$
```

Listing Contents in Long Format. Probably the most informative **ls** option is **−l**. If you type **ls −l<CR>** while in the *starship* directory, you would get the following:

```
$ ls -l<CR>
total 30
drwxr-xr-x        3      starship project     96Oct 2708:16bin
drwxr-xr-x        2      starship project     64Nov  114:19draft
drwxr-xr-x        2      starship project     80Nov  808:41letters
-rwx------        2      starship project 12301Nov  210:15list
-rw-------        1      starship project     40Oct 2710:00mbox
$
```

After the command line, the first line of output, *total 30*, shows the amount of memory used, which is measured in chunks called blocks. Next is one line for each directory and file. The first character in each of these lines tells you what kind of file is listed, where:

d = Directory,

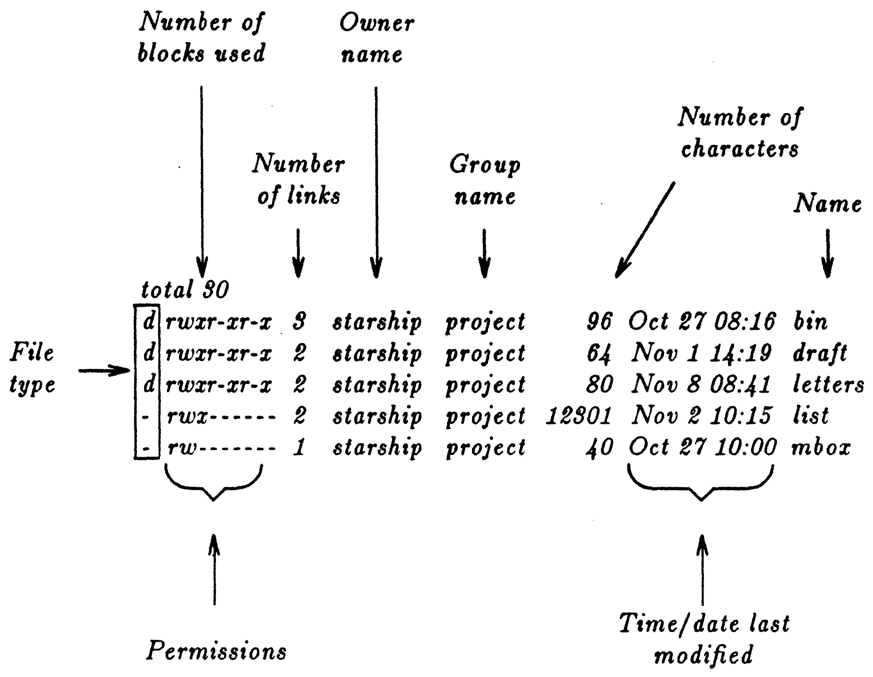
– = Ordinary disk file,

b = Block special file, and

c = Character special file.

The next several characters, which are either letters or hyphens, describe who has permission to read and use the file or directory. (Permissions are discussed with the **chmod** command in the section entitled *Accessing and Manipulating Files* in this chapter.) The following number is the link count, which in the case of a file, equals the number of directories it is in, or in the case of a directory, also includes the number of directories immediately under it in the file system structure. Next is the login name of the owner of the file, which is *starship*, and then the group name of the file or directory, which is *project*. The following number indicates the length of the file or directory entry measured in units of information (or memory) called bytes. Then there is the month, day, and time that the file was last modified. Finally, the file or directory name is given.

Figure 3-6 sums up what you get when you list the contents of a directory in long format.



```
total 30
drwxr-xr-x  3  starship  project     96  Oct 27 08:16  bin
drwxr-xr-x  2  starship  project     64  Nov 1 14:19   draft
drwxr-xr-x  2  starship  project     80  Nov 8 08:41   letters
-rwx------  2  starship  project  12301  Nov 2 10:15   list
-rw-------  1  starship  project     40  Oct 27 10:00  mbox
```

Figure 3-6. Description of output produced by the ls –l command

Command Summary. Following is a recap of capabilities provided by the **ls** command and two available options. See the *ICON/UXV User Reference Manual* for information on other available options.

Command Recap

ls - list contents of a directory

command	*options*	*arguments*
ls	**−a**, **−l**, and others*	**directoryname(s)**

Description: **ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, the contents of your working directory are listed.

Options:

−a Lists all entries, including those beginning with . (dot).

−l Lists contents of a directory in long format furnishing mode, permissions, size in bytes, and time of last modification.

Remarks: If you want to read the contents of a file, use the **cat** command.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Changing Your Working Directory *(cd)*

When you first log into the ICON/UXV system, you are placed in your home directory, which becomes your current or working directory. You may, however, wish to work in a different directory for any number of reasons. For example, you might want to create a file in a specific directory, you may need to make corrections to a file in another directory, or you may wish to obtain information by reading a file in a different directory.

Whatever the reason, the ICON/UXV system provides you with the **cd** command that allows you to move around in its directory structure. When you use the **cd** command to move to a new directory, that directory becomes your working directory.

To use the **cd** command, enter the command:

```
cd newdirectory-pathname<CR>
```

where the path name, whether full or relative, to the new directory is optional. Any valid path name of a directory can be used as an argument to the **cd** command. If you use the **cd** command without specifying a path name, it will move you to your login directory regardless of where you are in the file system.

When you specify a valid directory path name on the command line, the ICON/UXV system moves you to that directory. For example, to move from the *starship* directory to the child directory *draft* in the sample file system, type **cd draft<CR>**. In this example, *draft* is the relative path name to the desired directory. When you get the $ prompt, verify your new location by typing **pwd<CR>**. Your terminal monitor should look something like the following after going through this sequence:

```
$ cd draft<CR>
$ pwd<CR>
/user1/starship/draft
$
```

Now that you are in the *draft* directory you can access the files and directories in it, in this case, the files *outline* and *table*. You can also create subdirectories in *draft* with **mkdir** and additional files with the **ed** and **vi** commands. (See *Chapter 4* for general information on the **ed** and **vi** commands and *Chapter 5* and *Chapter 6* for tutorials on using the **ed** and **vi** commands, respectively.)

You may also use full path names with the **cd** command. For example, to move to the *letters* directory from the *draft* directory, you could use the command

cd /user1/starship/letters<CR>

where */user1/starship/letters* is the full path name to *letters*.

Or, since *letters* and *draft* are both children of *starship*, you could use the **cd** command with the relative path name *../letters*. The **..** notation moves you to the directory *starship*, and the remainder of the path name moves you to *letters*.

If you wish to return to your home directory after perusing the file system, simply type **cd<CR>**. The **cd** command with no arguments returns you to your login directory.

Command Recap

cd - change your working directory

command	*options*	*arguments*
cd	none	**directoryname**

Description:	**cd** changes your position in the file system from the current directory to the directory specified. If no directory name is given as an argument, the **cd** command places you in your home directory.
Remarks:	When the shell places you in the directory specified, the $ prompt is returned to you. You will also receive a $ prompt when you issue the **cd** command with no argument. To access a directory that is not in your working directory, you must substitute the full or relative path name in place of a simple directory name.

Removing Directories *(rmdir)*

If you decide you no longer need a directory, you can remove it with the **rmdir** command. The **rmdir** command, which stands for **rem**ove a **dir**ectory, removes a directory if that directory does not contain subdirectories and files, or, in other words, if the directory is empty.

If the directory you are attempting to remove is not empty, **rmdir** will not remove it unless you remove the contents of the directory first. In addition, you are not allowed to remove directories belonging to other system users unless you have permission to do so.

The standard format for the **rmdir** command is:

```
rmdir directoryname(s)<CR>
```

where one or more directory names can be specified.

If you were to attempt to remove the directory *bin* in the sample file system, the ICON/UXV system would respond in the following manner:

```
$ rmdir bin<CR>
rmdir: bin not empty
$
```

To remove the directory *bin* with the **rmdir** command, you would first have to remove the files *display* and *list* and the subdirectory *tools*. If you wish to remove files, see the section entitled *Accessing and Manipulating Files* in this chapter. To remove any subdirectories like *tools*, use the **rmdir** command. The system will return the $ prompt in response to the **rmdir** command when the directory specified in the command line is empty.

The command recap that follows summarizes how **rmdir** works.

Command Recap

rmdir - remove a directory

command	*options*	*arguments*
rmdir	none	**directoryname(s)**

Description:	**rmdir** removes named directories if they do not contain files and/or subdirectories.
Remarks:	If the directory is empty, the system returns the $ prompt when the directory is removed. If the directory contains files or subdirectories, the message, *rmdir: directory name not empty* is returned to you.

ACCESSING AND MANIPULATING FILES

This section introduces you to several ICON/UXV commands that access and manipulate files in the file system structure. Information in this section is organized into two parts--basic and advanced. The part devoted to basic commands is fundamental to your using the file system; the advanced commands offer you more sophisticated information processing techniques when working with files. You may skip reading the advanced section if you do not need to use the commands it covers.

Basic Commands

This section discusses ICON/UXV commands that are important to your being able to access and use the files in your directory structure. Specifically, these commands and their capabilities are:

cat	-- Outputs the contents of a file you name,
pg	-- Prints on a video display terminal the contents of a file you name in chunks or pages,
pr	-- Prints on your terminal a partially formatted version of the file you name,
lp	-- Allows you to request a paper copy of a file from a device called the line printer,
cp	-- Makes a duplicate copy of an existing file,
mv	-- Moves and renames a file,
rm	-- Permanently removes a file when you no longer need it,
wc	-- Counts the lines, words, and characters in a file, and
chmod	--Changes permission modes for a file (and a directory).

Each command is covered in one of following sections. A command recap follows the discussion of each command allowing you to review quickly the command line syntax and command capabilities.

Displaying a File's Contents *(cat, pg, pr)*

The ICON/UXV system provides three commands that allow you to display and print the contents of a file or files--**cat, pg,** and **pr**. The **cat** command, which stands for con**cat**enate, outputs the contents of files you specify by name on the command line, and displays the result on your terminal unless you tell **cat** to direct the output to another file or a new command. The **pg** command is particularly useful when you wish to read the contents of a lengthy file or a number of files because the command displays the text of a file in chunks or pages, a screenful at a time at your direction on a video display terminal. The **pr** command partially formats and outputs the files you specify on your terminal unless you direct the output to a paper printing device (see the **lp** command in this chapter).

The following three sections describe how to use these commands.

Concatenate and Print Contents of a File *(cat).* The **cat** command displays the contents of a file or files. For example, if you are located in directory *letters* in the sample file system and you wish to display the contents of the file *johnson*, you would type **cat johnson<CR>** and the following output would appear on the terminal.

```
$ cat johnson<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
$
```

As you can see, the contents of the file are displayed after the command line and are followed by the **$** prompt.

To display the contents of two (or more) files, like *johnson* and *sanders*, simply type **$ cat johnson sanders<CR>** and the **cat** command reads *johnson* and *sanders* and displays their contents in that order on your terminal.

```
$ cat johnson sanders<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
This file contains a letter
to Mrs. Sanders inviting her to
speak at our departmental
meeting.
$
```

To direct the output of the **cat** command to another file or to a new command, see the section in *Chapter 7* that discusses redirecting input and output.

The command recap that follows summarizes what you can expect the **cat** command to do.

Command Recap

cat - concatenate and print a file's contents

command	*options*	*arguments*
cat	available*	**filename(s)**

Description:	**cat** reads the name of each file given on the command line and displays the contents of the files.
Remarks:	If the file(s) exist, the contents are displayed on the terminal monitor; if not, the message *cat: cannot open filename* is returned to you. If you wish to display the contents of a directory, use the **ls** command.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Paging Through the Contents of a File (pg). The **pg** command, short for **page**, allows you to examine the contents of a file or files screenful by screenful on a video display terminal. The **pg** command displays the text of a file in chunks or pages followed by a colon (:). After displaying the colon, the system pauses and waits for your instructions to proceed. For example, your instructions can request **pg** to continue displaying the file's contents a page at a time or you can ask **pg** to search through the file(s) to locate a specific character pattern. *Table 3-1* summarizes some of the instructions you can give **pg** after the colon is displayed.

TABLE 3-1

Summary of Selected Commands for pg*

Command†	Meaning
h	Help; display list of available **pg** commands
q *or* Q	Quit **pg** perusal mode
<CR>	Display next page of text
l	Display next line of text
d or ^d	Display additional half page of text
. or ^l	Redisplay current page of text
f	Skip next page of text, and display following one
n	Begin displaying next file you specified on command line
p	Display previous file specified on command line
$	Display last page of text in file currently displayed
/pattern/	Search forward in file for specified character pattern
^pattern^	Search backward in file for specified character pattern

* See the *ICON/UXV User Reference Manual* for a detailed explanation of all available **pg** commands.

† Most commands can be typed with a number preceding them: +1 (display next page), −1 (display previous page), or 1 (display first page of text).

The **pg** command is especially useful when you wish to peruse a long file or a series of files because the system pauses after displaying each page allowing you as much time as you need to examine it. The size of the page displayed depends on the terminal you are using. For example, on a video display terminal with a window capable of showing 24 lines, 23 lines of text and a line containing the colon will be displayed as a page. However, if the file is less than 23 lines long, the page size will be the number of lines in the file plus the line containing the colon.

To peruse the contents of a file with **pg**, use the following command line format:

pg filename(s)<CR>

For example, to display the contents of the file **outline** in the sample file system, type **pg outline<CR>** and the first page of the file will appear on the screen. Since the file has more lines in it than can be displayed in one page, the colon indicates there is more to be looked at when you are ready. Pressing the **<CR>** key will print the next page of the file.

The following screen summarizes what has been done thus far.

```
$ pg outline<CR>
After you analyze the subject for your
report, you must consider organizing and
arranging the material you wish to use in
writing it.
        .
        .
        .
An outline is an effective method of
organizing the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
:<CR>
```

After pressing the <CR> key, the **pg** program will resume outputting the file's contents on the screen as follows:

```
that contains the names of the
ingredients and the order in which
to use them.
        .
        .
        .
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

In addition to the remainder of the file's contents, a line with the output *(EOF):* is displayed. The EOF designates that you have reached the end of the file and the colon is your cue for the next instruction.

When you have completed examining the file, you can type **q** or **Q** followed by pressing the <CR> key and the $ prompt will appear on your screen. Or you can choose to use one of the other available commands given in *Table 3-1* depending on your needs.

In addition, there are a number of options that can be specified on the **pg** command line. Refer to the *ICON/UXV User Reference Manual* if you are interested in learning more about them.

The following command recap summarizes the highlights of **pg**'s capabilities.

Command Recap

pg - display a file's contents in chunks or pages

command	*options*	*arguments*
pg	available*	**filename(s)**

Description: **pg** reads the name of each file given on the command line and displays the contents of the file(s) in chunks or pages, screenful by screenful.

Remarks: After displaying a screenful of text, the **pg** command awaits your instruction to continue to display text, to search for a pattern of characters, or to exit the **pg** perusal mode. In addition, a number of options are available for you to use with **pg** on the command line. For example, you can start to display the contents of file at a specific line or at a line containing a certain sequence or pattern or you can opt to go back and review text that has already been displayed.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Print Partially Formatted Contents of a File (pr). The **pr** command is typically used to prepare files for printing. You can expect the **pr** command to title, paginate, supply headings, and print a file according to varying page lengths and widths on your terminal monitor unless you specify that it prints on another output device, such as a line printer (read the discussion on the **lp** command in this section), or you direct the printing to a different file (see the section on redirecting input and output in *Chapter 7).*

If you choose not to specify any of the available options, the **pr** command produces output that is in a single column with 66 lines per page and is preceded by a short heading. The heading consists of five lines--two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. And the formatted file is followed by five blank lines. (Complete sets of text formatting tools, called **nroff** and **troff**, are available on ICON/UXV systems equipped with the appropriate application software. Check with your system administrator to see if this software is available to you.)

Typically, the **pr** command is used in tandem with the **lp** command to provide a paper copy of text as it was entered into a file. (See the section discussing the **lp** command for details.) However, you can also use the **pr** command to format partially and print the contents of a file on your terminal.

For example, to review the contents of the file *johnson* in the sample file system, type in the command **pr johnson<CR>**. The following screen summarizes this activity.

```
$ pr johnson<CR>

Nov 29 09:19 1988  johnson Page1

This file contains a letter
to Mr. Johnson on the topic of
office automation.
.
.
.
$
```

Note that the ellipses after the last line in the file stand for the remaining 58 lines (all blanks in this case) that **pr** formatted into the output. If you are working on a video display terminal, which typically allows you to view about 24 lines at a time, the entire 66 lines of the formatted file will print continuously and rapidly to the end of file. This means that the first 41 lines will "roll" off the top of your screen making it impossible for you to read them unless you have the ability to "roll" or "page" back a screen or two. If you are looking at a particularly long file, this feature might not solve the problem.

In this case, you should use the control-s **<^s>** combination to stop printing on your terminal temporarily and control-q **<^q>** to resume the printing.

The command recap that follows summarizes what you can expect the **pr** command to do.

Command Recap

pr - print partially formatted contents of a file

command	*options*	*arguments*
pr	available*	**filename(s)**

Description: **pr** produces a partially formatted copy of a file(s) on your terminal monitor unless otherwise specified. The program prints the text of the file(s) on 66-line pages and places five blank lines at the bottom of each page and a five-line heading at the top of each page. The heading consists of two blank lines; a line containing the date, time, file name and page number; and two additional blank lines.

Remarks: If the specified file(s) exists, the contents are partially formatted and displayed on the screen; if not, the message *pr: can't open filename* is returned to you.

The **pr** command is most commonly used with the **lp** command when a paper copy of a file is needed. However, when using the **pr** command to review a file on a video display terminal, use <**^s**> and <**^q**> to temporarily stop and start printing the file.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Requesting a Paper Copy of a File *(lp)*

At some point in time, you may want a paper copy of a file. Some terminals have built-in printers that allow you to get paper copies of files. In this case you simply need to turn the printer on and then use **cat** or **pr** to print the file. If, however, you wish to obtain a higher quality paper copy, you should consider using the **lp** command. The **lp** command, which stands for line printer, allows you to request a line-printing device to furnish you with a paper copy of a file or files.

The line printer or types of line printers that you have access to depends on what your UNIX system facility has to offer. You should ask your system administrator for the names of the line printers available to you. Or you can type **lpstat -v<CR>** to obtain a complete listing of every accessible line-printing device.

The basic format for the command is:

lp file<CR>

For example, to print the file *letters* on a line printer, you would type **lp letters<CR>** on the command line. In turn, the system would provide you with the name of the device or type of device on which the file will be printed and an identification (id) number indicating your request. The following screen summarizes this activity.

```
$ lp letters<CR>
Request id is laser-6885  1 file
$
```

The system response indicates that your job is to be printed on a laser line-printing device (the system default), has a request id number of 6885, and is to include the printing of one file.

Using the **−d***dest* (*destination*) option on the command line would cause your file to be printed on another available device that you name in place of *dest*. Using the **−m** option would cause mail to be sent to you indicating when the job is completed.

If you would like to cancel the request to **lp** to print the file *letters*, type **cancel laser-6885**<CR>, where laser-6885 is the request id. The **lpstat** command gives the status and request id of the line printer jobs.

A command recap follows that summarizes what you can expect of the **lp** command.

Command Recap

lp - request paper copy of file from a line printer

command	*options*	*arguments*
lp	**−d, −m,** and others*	**file(s)**

Description:	**lp** requests that specified files be printed by a line printer, thus providing paper copies of the contents.
Options:	**−d***dest* Allows you to choose *dest* as the printer or type of printer that is to produce the paper copy. If you do not use this option, the **lp** program specifies the printer for you.
	−m Sends a message to you via **mail** after the printing is complete.
Remarks:	You can cancel a request to the line printer by typing **cancel** and the request **id** furnished to you by the system when the request was acknowledged.
	Check with the system administrator for information on additional and/or different commands for printers that may be available at your location.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Making a Duplicate Copy of a File *(cp)*

When using the ICON/UXV system, you may wish to make a copy of a file. For example, you might want to revise a file while leaving the original version intact. The ICON/UXV system provides you with the **cp** command, short for **copy**, which copies the complete contents of one file into another. The **cp** command also allows you to copy one or more files from one directory into a different directory while leaving the original file or files in place.

To copy the file named *outline* to a file named *new.outline* in the sample directory, simply type **cp outline new.outline<CR>**. The system returns the **$** prompt when the copy is made. To verify the existence of the new file, you can type **ls<CR>**, which lists the names of all files and directories in the current directory, in this case *draft.* The following screen summarizes the activity.

```
$ cp outline new.outline<CR>
$ ls<CR>
new.outline
outline
table
$
```

You know from looking at the sample file system that the file *new.outline* did not exist before the **cp** command to copy *outline* to *new.outline* was given. However, if it had, it would have been replaced by a copy of the file *outline* and the previous version of *new.outline* would have been deleted.

If you had tried to copy the file *outline* to another file named *outline* in the same directory, the system would have told you that the file names were identical and returned the $ prompt to you. If you listed the contents of the directory to determine exactly how many copies of *outline* exist, the terminal monitor would look something like the following:

```
$ cp outline outline<CR>
cp: outline and outline are identical
$ ls<CR>
outline
table
$
```

As you can see, the ICON/UXV system does not allow you to have two files with the same name in a directory.

You could, however, copy the file named *outline* from the directory *draft* to another file named *outline* in the directory named *letters* by using any of the following command lines assuming you are currently in *draft*:

```
cp outline ../letters/outline<CR>
cp outline ../letters<CR>
cp outline /user1/starship/letters/outline<CR>
cp outline /user1/starship/letters<CR>
```

A copy of the file *outline* would reside in both directories *draft* and *letters* after using one of these commands since each of them contains a legal path name to the file *outline*. From this example, you can see that the ICON/UXV system allows you to have files with identical names as long as they are in different directories.

If you would like to copy the file *outline* in the directory *draft* to a file named *outline.vers2* in the directory *letters*, you could use either of the following command lines:

```
cp outline ../letters/outline.vers2<CR>
cp outline /user1/starship/letters/outline.vers2<CR>
```

Keep in mind the conventions for naming directories and files given in the section entitled *Creating Directories* in this chapter.

The following recap summarizes how the **cp** command works.

Command Recap

cp - make a copy of a file

command	*options*	*arguments*
cp	none	**file1 file2** **file(s) directory**

Description: **cp** allows you to make a copy of *file1* and call it *file2* leaving *file1* intact, or to copy one or more files into a different directory.

Remarks: When copying *file1* to *file2* and *file2* already exists, the **cp** command will overwrite the first version of *file2* with a copy of *file1* calling it *file2*. The first version of *file2* is deleted.

You cannot copy directories with the **cp** command.

Moving and Renaming a File *(mv)*

The **mv** command allows you to rename a file in the same directory or to move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name.

To rename a file in a directory, use the following command:

```
mv file1 file2<CR>
```

The **mv** command changes a file's name from *file1* to *file2*. Remember that the names *file1* and *file2* can be any valid names, including path names.

For example, if you are in the directory *draft* in the sample file system and you would like to rename the file *table* as *new.table*, simply type **mv table new.table<CR>**. You should receive the $ command prompt if the command executed successfully. To verify that the file *new.table* exists, you can list the contents of the directory by typing **ls<CR>**. In turn, the terminal should read:

```
$ mv table new.table<CR>
$ ls<CR>
new.table
outline
$
```

You can also move a file from one directory to another keeping the file's name the same or changing it to a different one. To do so, use the following command line.

mv file(s) directory<CR>

where the file and directory names can be any valid names, including path names.

To move the file *table* from the current directory named *draft* (whose full path name is */user1/starship/draft*) to a file with the same name in the directory **letters** (whose relative path name from *draft* is *../letters* and whose full path name is */user1/starship/letters*), any one of several command lines can be used, including the following:

```
mv table /user1/starship/letters<CR>
mv table /user1/starship/letters/table<CR>
mv table ../letters<CR>
mv table ../letters/table<CR>
mv /user1/starship/draft/table /user1/starship/letters/table<CR>
```

The file *table* could have been renamed *table2* when moving it to the directory *letters* using any of the following:

```
mv table /user1/starship/letters/table2<CR>
mv table ../letters/table2<CR>
mv /user1/starship/draft/table2 /user1/starship/letters/table2<CR>
```

You can verify that the command worked by listing the contents of the directory with the **ls** command.

Refer to the recap that follows for a summary of how the **mv** command works.

Command Recap

mv - move or rename files

command	*options*	*arguments*
mv	none	**file1 file2** **file(s) directory**

Description:	**mv** allows you to change the name of a file or to move a file(s) into another directory.
Remarks:	When changing the name of *file1* to *file2* and *file2* already exists, the **mv** command will overwrite the first version of *file2* with *file1* and rename it *file2*. The first version of *file2* is deleted.

Removing a File *(rm)*

When you no longer need a file, you can get rid of it by using the **rm** command, which is short for **r**e**m**ove.

To remove one or more files, use the format:

rm file(s)<CR>

After the command executes, the file(s) you specified are removed permanently.

To remove a file named *new.outline* in the current directory type **rm new.outline<CR>** and list the contents of the directory with the **ls** command to verify that the file *new.outline* no longer exists.

To remove more than one file, such as the files *outline* and *table*, type **rm outline table<CR>** and list the contents of the directory by typing **ls<CR>**.

```
$ rm outline table<CR>
$ ls<CR>
$
```

The $ response indicates that the files named *outline* and *table* were removed permanently.

The following recap summarizes how the **rm** command works.

Command Recap

rm - remove a file

command	*options*	*arguments*
rm	available*	**file(s)**

Description:	**rm** allows you to remove one or more files.
Remarks:	Files specified as arguments to the **rm** command are removed permanently.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Counting Lines, Words, and Characters in a File *(wc)*

The **wc** command, which stands for **w**ord **c**ount, reports the number of lines, words, and characters there are in a file that you specify by name on the command line. If you name more than one file, the **wc** program counts the number of lines, words, and characters in each specified file and then totals the counts. In addition, you can direct the **wc** program to give you only a line, a word, or a character count by using the **−l**, **−w**, or **−c** options, respectively.

To determine the number of lines, words, and characters in a file, use the following format on the command line:

```
wc file1<CR>
```

When you do, the system responds with a line in the format:

```
l    w    c    file1
```

where

l = Number of lines in *file1*,

w = Number of words in *file1*, and

c = Number of characters in *file1*.

For example, to count the lines, words, and characters in the file *johnson* in the current directory *letters*, type **wc johnson<CR>**. The terminal monitor would show the following output:

```
$ wc johnson<CR>
     3    14     78  johnson
$
```

The system response displays the line count (*3*), the word count (*14*), and the character count (*78*) for the file *johnson.*

To determine the number of lines, words, and characters in more than one file, use the following format:

wc file1 file2<CR>

In turn, the system responds with the following format:

```
l    w    c    file1
l    w    c    file2
l    w    c    total
```

where line, word, and character counts are displayed for *file1* and *file2* on separate lines and the combined counts appear on the last line called *total.*

If you request that the **wc** program count lines, words, and characters in the files *johnson* and *sanders* in the current directory, the system would respond as follows:

```
$ wc johnson sanders<CR>
     3    14     78  johnson
     4    16     95  sanders
     7    30    173  total
$
```

In this case, the first line of the system response shows the line, word, and character counts for the file *johnson.* The second line of output gives line, word, and character counts for *sanders.* The last line of output shows combined line, word, and character counts for both files in the line labeled *total.*

If you prefer to get only a line, a word, or a character count, select the appropriate format from the following lines:

```
wc   -l   file1<CR>   (line count)
wc   -w   file1<CR>   (word count)
wc   -c   file1<CR>   (character count)
```

For instance, by typing **wc –l sanders<CR>** on the command line you would obtain the following output:

```
$ wc -l sanders<CR>
    4  sanders
$
```

The system tells you that the number of lines in the file *sanders* is 4 in answer to specifying –l. If the **–w** or **–c** option was specified for that file, the ICON/UXV system would have responded with the number of words or number of characters, respectively, in the file.

The command recap that follows summarizes how the **wc** command works.

Command Recap

wc - count lines, words, and characters in a file

command	*options*	*arguments*
wc	**–l, –w, –c**	**file(s)**

Description: **wc** counts lines, words, and characters in the file(s) named keeping a total count of all tallies when more than one file is specified.

Options

–l Counts the number of lines in the specified file(s).

–w Counts the number of words in the specified file(s).

–c Counts the number of characters in a specified file(s).

Remarks: When a file name is specified in the command line, it is printed with the count(s) requested.

Protecting Your Files *(chmod)*

The **chmod** command, short for **change mode**, allows you to decide who can read, alter, and use your files and who cannot. Because the ICON/UXV operating system is a multiuser system, you are not working alone in the file system: you and other system users can follow path names and run system commands to move to various directories and to read and use files belonging to one another if you have permission to do so.

If you own a file, then you are able to determine who has the right to read that file, to make changes to or write the file, and to run or execute the file if it is a program. These permissions are defined as:

r = Allows system users to read a file or to copy its contents,

w = Allows system users to write changes into a file or copy of a file, and

x = Permits system users to run an executable file.

Specifically, you can determine who in the population of ICON/UXV users is entitled to these various permissions and who is not according to the following classifications:

u = You, the user and login owner of your files and directories,

g = Members of the group to which you belong (the group could consist of team members working on a project, members of a department, or a group arbitrarily designated by the person who set up your ICON/UXV account), and

o = All other system users.

When you create a file or a directory, the system automatically grants or denies permission specifically to you, members of your group, and other system users. You can alter this automatic action to some extent by modifying your environment, which is discussed in *Chapter 7*. Regardless of how the permissions are granted when a file is created, as the owner of the file or directory it is up to you to allow current permissions to remain in effect or to change them to suit your purposes and the situation. For example, you may wish to keep certain files private and for your use only. Or you may wish to grant permission to read and to write changes into a file to members of your group and all other system users as well. Or you may share a program with members of your group by granting them permission to execute it.

How to Determine Existing Permissions. You can determine what permissions are currently in effect on a file or a directory by using the command that produces a long listing of a directory's content, which is **ls –l**. For example, typing **ls –l<CR>** while in the directory named ***starship/bin*** in the sample file system would produce the following output:

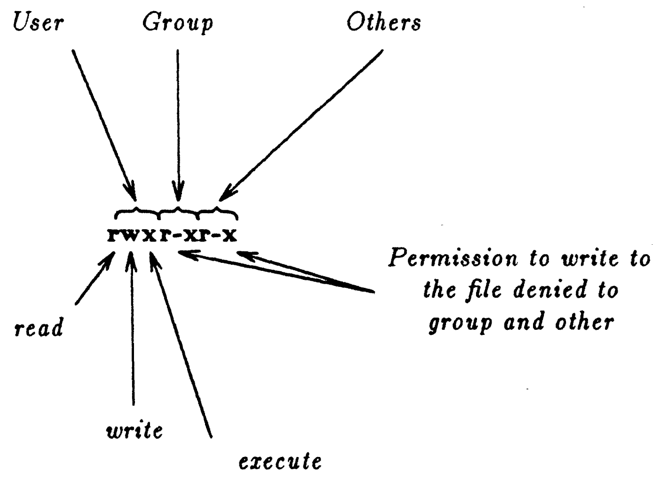
```
$ ls -l<CR>
total 35
-rwxr-xr-x        1      starship project9346Nov 108:06display
-rwx--x--x        1      starship project6428Dec 210:24list
drwx--x--x        2      starship project  32Nov 815:32tools
$
```

Permissions for the files *display* and *list* and the directory *tools* are shown on the left of the terminal monitor under the line *total 35*, and look like:

rwxr-xr-x	(file *display*)
rwx--x--x	(file *list*)
rwx--x--x	(directory *tools*)

These nine characters represent three groups of three characters. The first set of three characters refers to your (or the user's/owner's) permissions, the second set to members of the group, the last set to all other system users. Within each set of characters, the *r*, *w*, and *x* indicate the permission currently enabled for the groups. If a dash appears instead of an *r*, *w*, or *x*, permission to read, write, or execute is denied.

The following diagram summarizes this breakdown for the file named *display*.



As you can see, the owner has *r*, *w*, and *x* permissions and members of the group and other system users have *r* and *x* permissions.

How to Change Existing Permissions. After you have determined what permissions are in effect, you can change them using the following format:

chmod who + (or −) permission file(s)<CR>

where:

chmod	= Name of program,
who	= One of three user groups *u*, *g*, *o*: *u* = User, *g* = Group, and *o* = Other.
+ −	= Instruction that grants (+) or denies (−) permission.
permission	= Authorization to *r*, *w*, or *x*: *r* = Read, *w* = Write, and *x* = Execute.
file(s)	= File (or directory) name(s) listed; assumed to be branches from your working directory, unless you use full path (names).

This may sound a bit confusing. But, a few examples on how to use the **chmod** command should help to make permission possibilities clear.

Let's use the permissions for the file *display* to experiment with **chmod**. You can see from the permissions that as the user and owner of *display* you can read, write, and run this executable file. You can protect the file against accidentally changing it by denying yourself write (*w*) permission by typing the command line **chmod u−w display<CR>**. After receiving the **$** prompt, type in **ls −l<CR>** to verify the permission has changed.

```
$ chmod u−w display<CR>
$ ls −l<CR>
total 35
-r-xr-xr-x          1      starship project9346Nov 108:06display
-rwx--x--x          1      starship project6428Dec 210:24list
drwx--x--x          2      starship project  32Nov 815:32tools
$
```

From this output, you can see that you no longer have permission to write changes into the file, that is, unless you change the mode back to include write permission.

Now, let's consider another example. Notice that permission to write into the file *display* has been denied to members of your group and other system users. These users, however, have read permission, which means that any of these users can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you could deny them read permission by typing **chmod go−r display<CR>**. The *g* and *o* stand for group members and all other system users, respectively, and the **−r** denies them permission to read or copy the file. Check the results with the **ls −l** command.

```
$ chmod go−r display<CR>
$ ls −l<CR>
total 35
-rwx--x--x          1      starship project9346Nov 108:06display
-rwx--x--x          1      starship project6428Dec 210:24list
drwx--x--x          2      starship project  32Nov 815:32tools
$
```

A Note on Permissions and Directories. If you read the preceding pages describing the **chmod** command, you might have gathered that you can use this command to grant or deny permission for directories as well as files. It is true, you can. To do so, simply use the directory name instead of a file name on the command line.

The impact, however, of granting or denying permissions for directories to various system users is worth considering. For example, if you grant read permission for a directory to yourself (u), members of your group (*g*), and other system users (*o*), every user who has access to the system can read the names of the files that directory contains by using the **ls −l** command. Similarly, granting write permission allows the designated users to create new files in the directory and change and remove existing ones. And granting permission to execute the directory allows the designated users the ability to move to that directory (and make it their working directory) by using the **cd** command.

An Alternate Method. The **chmod** method described in the preceding pages is one of two ways to change permissions to read, write, and execute files and directories. The method previously described uses symbols, such as *r*, *w*, *x* and *u*, *g*, *o*, to specify instructions to **chmod**. Hence, it is called the symbolic method.

The alternate method uses a number system called octal that is different than the decimal number system we typically use on a day-to-day basis. This method uses three octal numbers ranging from 0 through 7 to assign permissions. If you wish to use the octal method when changing permission, see the description of **chmod** in the *ICON/UXV User Reference Manual.*

Summary. The command recap that follows provides a quick reference on how **chmod** works.

Command Recap

chmod - change permission modes for files (and directories)

command	*instruction*	*arguments*
chmod	**who + − permission**	**filename(s)** **directoryname(s)**

Description:	**chmod** gives (+) or removes (−) *read*, *write*, and *execute* permissions for three types of system users: *user* (you), *group* (members of your group), and *other* (all other users able to access the system on which you are working).
Remarks:	The instruction set can be represented in either octal or symbolic terms.

Advanced Commands

You will become more and more familiar with the file system as you use the commands thus far discussed in this chapter. As this familiarity increases so might your need or interest for more sophisticated information processing techniques when working with files. This section introduces you to three commands that give you just that. These commands and their capabilities are listed as follows:

diff --	Finds difference between two files,
grep --	Searches a file for a pattern, and
sort --	Sorts and merges files.

The following discussion only scratches the surface on information processing techniques available with the ICON/UXV system. You may refer to the *ICON/UXV User Reference Manual* for additional information.

Identifying Differences Between Files *(diff)*

The **diff** command locates all the differences between two files and proceeds to tell you how to change the first file to be a carbon copy of the second. It reports all differences between the files.

The basic format for the command is:

```
diff file1 file2<CR>
```

If *file1* and *file2* are identical, the system returns the **$** prompt to you. If not, the **diff** command instructs you on how to bring the first file into agreement with the second by using line editor (**ed**) commands. (See *Chapter 5* for details on the line editor.) The ICON/UXV flags lines in *file1* with the < symbol and *file2* with the > symbol.

For example, if you use the **diff** command to identify differences between the files *johnson* and *sanders*, the system would respond as follows:

```
$ diff johnson sanders<CR>
2,3c2,4
< to Mr. Johnson on the topic of
< office automation.
---
> to Mrs. Sanders inviting her to
> speak at your departmental
> meeting.
$
```

The first line of the system response is

2,3c2,4

which means lines 2 through 3 in the file *johnson* must be changed (designated by *c*) to lines 2 through 4 in the file *sanders*. The system then displays lines 2 through 3 in the file *johnson* as follows:

```
< to Mr. Johnson on the topic of
< office automation.
```

and lines 2 through 4 in the file *sanders*

```
> to Mrs. Sanders inviting her to
> speak at our departmental
> meeting.
```

If you make these changes (using the **ed** or the **vi** text editing program), the file *johnson* will be identical to the file *sanders*. Remember, the **diff** command tells you exactly what the differences are between the named files. If you simply want an identical copy of a file, use the **cp** command. Refer to the recap that follows for a summary of what you can expect the **diff** command to do when no options are specified. See the reference to the *ICON/UXV User Reference Manual* for details on available options.

Command Recap

diff - finds differences between two files

command	*options*	*arguments*
diff	available*	**file1 file2**

Description:	**diff** reports what lines are different in two files and what you must do to make the first file identical with the second.
Remarks:	Instructions on how to change a file to bring it into agreement with another file are line editor (**ed**) commands: *a* (append), *c* (change), or *d* (delete). Numbers given with *a, c,* or *d* indicate the lines to be modified. Also used are the symbols < (indicating a line from the first file) and > (indicating a line from the second file).

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Searching a File for a Pattern *(grep)*

You can request the ICON/UXV system to search through files for a specific word, phrase, or group of characters by using the **grep** command. Technically, **grep** means **g**lobally search through a file or files to locate a **r**egular **e**xpression and **p**rint the lines that contain the regular expression. A regular expression is the pattern of characters--be it a word, a phrase, or an equation--that you stipulate.

The basic format for the command line is:

grep pattern file(s)<CR>

Thus, to locate the line containing the word *automation* in the file *johnson*, you would type:

grep automation johnson<CR>

and the system would respond as follows:

```
$ grep automation johnson<CR>
office automation
$
```

The output gives you all the lines in the file *johnson* that contain the pattern for which you were searching, which is the word *automation*.

If the pattern contains multiple words or any characters that have a special meaning to the ICON/UXV operating system, such as $, ¦, *, ?, and so on, the entire pattern must be enclosed in single quotes. (For an explanation of the special meaning for these and other characters see the section entitled *Metacharacters* in *Chapter 7, Shell Tutorial.*) For example, if you are interested in locating the lines containing the pattern *office automation*, the command line and system response would read:

```
$ grep `office automation´ johnson<CR>
office automation.
$
```

But what if you could not recall to whom you sent a letter on the topic of office automation in the first place--Mr. Johnson or Mrs. Sanders? You could type:

grep `office automation´ johnson sanders<CR>

If you did, the system would respond in the following manner:

```
$ grep `office automation´ johnson sanders<CR>
johnson:office automation.
$
```

The output tells you that the pattern *office automation* is found once in the file *johnson*.

In addition to the capabilities of the **grep** command that are summarized in the recap that follows, the ICON/UXV system provides variations to the basic **grep** command, called **egrep** and **fgrep**, along with several options that further enhance the searching powers of the command. See the *ICON/UXV User Reference Manual* if you are interested in learning more.

Command Recap

grep - searches a file for a pattern

command	*options*	*arguments*
grep	available*	**pattern file(s)**

Description:	**grep** searches the file or files you name for lines containing a pattern and then prints the lines that match. If you name more than one file, the name of the file containing the pattern is given also.
Remarks:	If the pattern you give contains multiple words or special characters, enclose the pattern in single quotes on the command line.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Sorting and Merging Files *(sort)*

The ICON/UXV system provides you with an efficient tool called **sort** for sorting and merging files. The basic form of the command line is:

sort file(s)<CR>

which causes lines in the specified files to be sorted and merged in the order defined by the ASCII representations of the characters in the lines.

- Lines beginning with numbers are sorted by digit and listed before letters in the output,
- Lines beginning with uppercase letters are listed before lines beginning with lowercase letters, and
- Lines beginning with symbols, such as *, %, or @, are sorted on the basis of the symbol's ASCII representation.

To get an idea of how the **sort** command works, let's say that you have two files, named *phase1* and *phase2*, each containing a list of names that you wish to sort alphabetically and finally interfile into one list. First, display the contents of each file using the **cat** command.

```
$ cat phase1<CR>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
$ cat phase2<CR>
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
$
```

(Note: we could have used the command line **cat phase1 phase2<CR>** instead of listing the contents of each file separately.)

Now, sort and merge the contents of the two files using the **sort** command. Note that the output of the **sort** program will print on the terminal monitor unless you specify otherwise.

```
$ sort phase1 phase2<CR>
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
$
```

In addition to putting together simple listings as in the previous examples, the **sort** command can rearrange the lines and parts of lines (called fields) according to a number of other specifications you can designate on the command line. The possible specifications are complex and are not within the scope of this text. You should consult the *ICON/UXV User Reference Manual* for a full rundown on the available options.

However, the following command recap summarizes the capabilities of the **sort** program.

Command Recap

sort - sorts and merges files

command	*options*	*arguments*
sort	**available***	**file(s)**

Description: **sort** sorts and merges lines from the file or files you name and displays the result on your terminal.

Remarks: If no options are specified on the command line, lines are sorted and merged in the order defined by the ASCII representations of the characters in the lines.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

SUMMARY

This chapter described the structure of the file system and presented ways to use and to navigate through the file system via ICON/UXV commands. The next chapter gives you an overview of a variety of ICON/UXV capabilities, such as text editing, using the shell as a command language, communicating electronically with other system users, and programming and developing software.

Chapter 4

ICON/UXV SYSTEM CAPABILITIES

PAGE

Chapter 4

ICON/UXV SYSTEM CAPABILITIES

INTRODUCTION

This chapter serves as a transition between the first three chapters in the overview part of this guide and the four tutorials that follow. The material in this chapter combines basic, fundamental concepts about the ICON/UXV system covered in the first three chapters of this guide with information about system capabilities that you may use to do your computing work efficiently and effectively.

This chapter provides an overview of the following ICON/UXV capabilities: text editing, working in the shell, communicating electronically, and programming in the ICON/UXV environment. In addition, it serves as an introduction to chapters 5, 6, 7, and 8--*Line Editor Tutorial*, *Screen Editor Tutorial*, *Shell Tutorial*, and *Communication Tutorial*, respectively.

TEXT EDITING

You have read a good deal about files up to this point simply because using the file is a way of life in a ICON/UXV environment. The information in this section will enhance your knowledge about manipulating files by introducing you to a software tool called a text editor. A text editor provides you with the ability to create and modify files: it will help you to fare well in the ICON/UXV system since a considerable amount of your computing time may be spent writing and revising letters, memos, reports, or source code for programs that will be stored in files.

This section contains information that tells you what a text editor is and how it works. In addition, this section acquaints you with two types of text editors supported on the ICON/UXV system: the line editor and the visual, or screen, editor. Since you will probably come to prefer one of these editing programs over the other--even if you learn to use them equally well--the line editor and the screen editor are briefly compared to help you to assess their capabilities. For detailed information on the line editor and the screen editor, see *Chapter 5* and *Chapter 6.*

What Is a Text Editor?

When you write or type letters, memos, and reports and then decide to change what you have written or typed, you will use skills required in text editing. These skills include inserting new or additional material, deleting unneeded material, transposing material (sometimes called cutting and pasting), and finally preparing a clean, corrected copy. Text editors perform these tasks at your direction making writing and revising text much easier and quicker than if done by hand or on a typewriter.

In the ICON/UXV system, a text editor is much like the ICON/UXV shell. Both a text editor and the shell are programs that accept your commands and then perform the requested functions--essentially, they are both interactive programs. A major difference between a text editor and the shell, however, is the set of commands that each recognizes. All the commands you have learned up to this point belong to the shell's command set. A text editor, on the other hand, has its own distinct set of commands that allow you to create, move, add, and delete text in files, as well as acquire text from other files.

How Does a Text Editor Work?

To understand how a text editor works you need information about the environment created when you use an editing program and the modes of operation understood by a text editor.

Text Editing Buffers

To create a new file, you must ask the shell to put the editor in control of your computing session. When you do, a temporary work space is allocated to you by the editor. This work space is called the editing buffer, in it you can enter information you want the file to hold and modify it if you wish.

Because you are in a temporary work space when using a text editor, the file you are creating along with the changes you make to it are also temporary. This work space allotment and what it is holding will exist only as long as you work in the editing program. If you wish to save the file, you must tell the text editor to write the contents of the buffer into a storage area. If you do not tell the editor to write or record what you have done during the editing session, the buffer's contents will disappear when you leave the editing program. If you forget to write a new file or update an existing one, the text editors remind you to do so when you attempt to leave the editing environment.

To modify an existing file, the procedure is almost identical to the one you follow when creating a new file. First, call the editor and give it the name of the file you wish to change. In turn, the editor makes a copy of the file that is in the storage area and places it in the buffer so you can work on it.

When you finish editing the file, you can write the buffer's contents into storage and leave the editing program knowing the file is updated and ready to be recalled when you need it again. Or you can chose to leave the editor without writing the file if you have made a critical mistake or you are unhappy with the edited version. This step leaves the original file intact and the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text you put in the buffer is organized into lines. A line of text is simply the series of characters that appears horizontally across a row of typing that is ended by pressing the **<CR>** key. Occasionally, files may contain a line of text that is too long to fit on the terminal monitor. Some terminals will automatically display the continuation of the line on the next row of the monitor, whereas others will not.

Modes of Operation

Text editors are capable of understanding two modes of operation: the command mode and the text input mode.

When you begin an editing session, you will automatically be placed in command mode. In command mode, all your input is interpreted as a command. Typical editing commands allow you to move about in a file, search for patterns in the file's contents, or print a portion of a file on the terminal monitor. The input mode is entered when you use a command to create text. Once in input mode, what you type on the keyboard is placed into the buffer as part of the text file until you send the appropriate instruction to the editor that returns you to command mode.

You may occasionally lose track of the mode in which you are working by attempting to enter text while in command mode or by trying to enter a command while in input mode. This is something even experienced users do from time to time. It will not take long to recognize the mistake and it will be apparent what to do to remedy these situations as you work through the tutorials in *Chapter 5* and *Chapter 6.*

Line Editor

The line editor, accessed by the **ed** command, is a fast, versatile program for preparing text files. This editor gets its name because it operates on the lines of text a file holds. For example, to change a single character in a file, you specify the line of the file that contains the character you wish to change and then specify the change.

Put simply, you manipulate text on a line-by-line basis with the line editor. Commands for this text editor can change lines, print lines, read and write files, and initiate text entry. In addition, you can specify the line editor to run from a shell program; something you cannot do with the screen editor. (See *Chapter 7* for information on basic shell programming techniques.)

The line editor works equally well on paper printing terminals and video display terminals. It will also obligingly accommodate you if you are using a slow-speed telephone line.

Refer to *Chapter 5, Line Editor Tutorial,* for instructions on how to use this editing tool. Also see *Appendix D* for a summary of line editor commands. If you are interested in a comparison of line editor (**ed**) and screen editor (**vi**) features, see *Table 4-1.*

Screen Editor

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. When you use the screen editor, your terminal acts as a window to let you view the file you are editing a screenful or page at a time. This editor works most efficiently and effectively when used on a video display terminal operating at 1,200 or higher baud.

For the most part, modifications to a file (such as, additions, deletions, and changes) are accomplished by positioning the cursor at the point in the window where the modification is to be made and then making the change. In other words, the screen editor displays the effects of editing

TABLE 4-1

Comparison of Line *(ed)* and Screen *(vi)* Editors

Feature	Line Editor *(ed)*	Screen Editor *(vi)*
Recommended terminal type	Paper-printing or VDT*	VDT
Speed	Accommodates high- and low-speed data transmission lines.	Works best via high-speed data transmission lines (1,200+ baud).
Versatility	Can be specified to run from shell scripts as well as used during editing sessions.	Must be used interactively during editing sessions.
Sophistication	Changes text quickly. Uses comparatively small amounts of processing time.	Changes text easily. However, can make heavy demands on computer resources.
Power	Provides a full set of editing commands. Standard ICON/UXV text editor.	Provides its own editing commands and recognizes all line editor commands as well.

* VDT = video display terminal

changes in the context in which you make them. Because of this feature, the screen editor in considered to be much more sophisticated than the line editor.

Furthermore, the screen editor offers a replete collection of commands. For example, a number of screen editor commands allow you to move the cursor around within the window to a file. Other commands move the window up or down through a page or more of the file. Still other commands allow you to change existing text or to create new text. In addition to its own set of commands, the screen editor has access to all the commands offered by the line editor. This arsenal of commands accounts for the screen editor's tremendous power.

There is, however, a trade-off for the screen editor's speed, visual appeal, efficiency, and power, which is the heavy demand that it places on the computer's processing time. For example, a simple change might cause an entire screen to need updating. Moreover, if simple changes lead to long delays while you wait for a screen to be updated, the pleasant experience of using a visual-oriented editor can be somewhat diminished.

Refer to *Chapter 6, Screen Editor Tutorial*, for instructions on how to use this software. And see *Appendix E*, which contains a summary of screen editor commands. If you wish to compare the features of the line editor (**ed**) and the screen editor (**vi**) see *Table 4-1*.

WORKING IN THE SHELL

Every time you log into the ICON/UXV system you will be communicating directly with a program called the shell. You will continue to interact with the shell until you log off the system, unless you use a program, such as a text editor, that temporarily suspends your dealings with the shell until you are finished using that particular program.

The shell is much like other programs, except that instead of performing one job, as **cat** or **ls** does, it is central to most of your interactions with the ICON/UXV system. This is because the shell's primary function is to act as an interpreter between you and the computer on which the ICON/UXV system is running. As an interpreter, the shell translates your requests into language the computer understands, calls requested programs into memory, and executes them.

This section acquaints you with some of the ways you can use the shell as the command language interpreter to simplify a computing session and to enhance your ability to use system features. In addition to running a single program for you, you can also use the shell to:

- interpret the name of a file or a directory you input in an abbreviated way using a type of "shell shorthand,"
- redirect the flow of input and output of the programs you run,
- execute multiple programs, and
- tailor your computing environment to meet your individual needs and preferences.

In addition to being the command language interpreter, the shell is also a programming language. If you would like an overview of shell programming capabilities, see the section entitled *Programming in the System* at the end of this chapter. Or refer to *Chapter 7, Shell Tutorial,* for detailed information on how to use the shell as a command language interpreter and as a programming language. For complete, unabridged information on shell programming *Part 3* of this manual should be consulted.

Using Shell Shorthand

Many ICON/UXV commands require that you name a file or a directory as an argument to it on a command line, such as **mkdir directory name(s)<CR>** or **rm filename(s)<CR>**. Easy enough! But suppose you have 12 files to remove corresponding to monthly reports for 1983 named *rept1*, *rept2*, *rept3*, *rept4*, and so on? Or suppose you need to move 24 files corresponding to file names *sect1*, *sect2*, ... *sect24* to a different directory?

Typing the file name for each monthly report after the **rm** command or the file name for each section after the **mv** command is still easy, but all the repetition gets tedious after inputting four or five names.

In instances like these, you should consider using shorthand notation when specifying file or directory names. If the file or directory names have some part in common, you can use a type of shorthand to tell the shell that you are referring to all of them on the basis of the similarity without specifying each one individually. Or, if a file has a unique character or sequence of characters within a group of similarly named files, you can use this shorthand notation to locate the file on the basis of the difference.

The ICON/UXV operating system recognizes several characters as having special meanings when they are used in place of a directory name or when they appear as part of a file or directory name on a command line. These characters allow you to specify the names of files and directories in a rapid, abbreviated way. Some of the characters are referred to as metacharacters because of their special meanings to the shell.

The special characters are . .. ? * [ ] – \ and their meanings are summarized in *Table 4-2*. When you specify file or directory names, you can substitute various characters within them with the appropriate shorthand abbreviation. Any part of the name that is not a special character is taken at its literal value.

TABLE 4-2

Shorthand Notation for File and Directory Names

Special Character	Meaning	Detailed Reference
.	Current directory	*Chapter 3*
..	Parent directory	*Chapter 3*
?	Match any single character	*Chapter 7*
*	Match any number of characters	*Chapter 7*
[ ]	Designate a sequence of characters to be matched, such as [abc] or [628]	*Chapter 7*
–	Specify a character range within [ ], such as A-Z	*Chapter 7*
\	Remove meaning of special characters	*Chapters 2, 7*

For example, for the possibilities described at the beginning of this section, typing **rm rept*<CR>** would remove all the files in the current directory starting with the characters *rept* followed by any other characters corresponding to monthly reports for 1983, and typing **mv sect* ../chapter3<CR>** would move all the files from the current directory beginning with the letters *sect* and followed by any other characters to another directory named *chapter3* belonging to its parent directory.

Details on how to use the special characters appear in other chapters of this guide as indicated in *Table 4-2*. Refer to that chapter for the information you need.

Redirecting the Flow of Input and Output

Up to this point in the *ICON/UXV User Guide*, any request to ask the shell to execute a command was done by inputting the command and the necessary argument(s) on the terminal keyboard. In turn, the output, if any, was displayed on the terminal monitor. This pattern illustrates the idea of standard input and standard output.

In general, the place from which a program expects to receive its input is called the standard input. A ICON/UXV command called **mail**, which you will learn more about in *Chapter 8*, provides a good example of this and warrants mentioning here. For example, to use **mail**, you would simply type **mail jmrs<CR>** and the **mail** command takes everything you type on your keyboard after <CR> until you type <d> as input. After you type <d>, mail sends your input to the person with the login name *jmrs*. The place to which a program writes its results, in this case the login name *jmrs*, is referred to as the standard output.

In the ICON/UXV system, most commands expect to receive their input from the keyboard and then display output on the terminal monitor. By default, then, the standard input is the keyboard and the standard output is the terminal monitor (*Figure 4-1*).

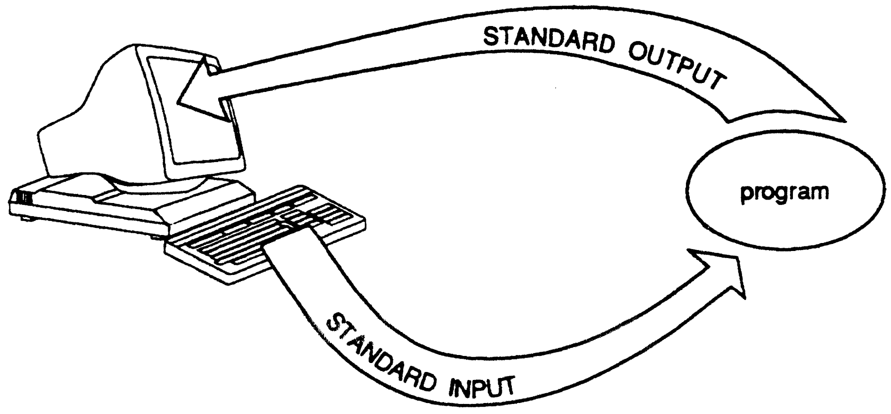


Figure 4-1. Standard input/output flow. A program's standard input and standard output are usually assigned to your terminal.

You can, if you wish, use a feature called redirection to change these defaults. Put simply, redirection is a ICON/UXV feature that allows you to request the shell to reassign standard input and/or standard output to other files or devices.

With the redirection feature, you can request the shell to do the following:

- reassign to a file any output that a program would ordinarily send to your terminal,
- have a program take its input from a file rather than from your terminal keyboard, or
- use a program as the source of data for another program.

You request the shell to redirect input and output using a set of operators, which are > (greater than sign), >> (two greater than signs), < (less than sign), and ¦ (a pipe). Now let's take a look at what each of these operators can do for you.

Redirecting the Standard Output *(>)*

The > operator allows you to redirect the output of a command (or program) into a file (*Figure 4-2*).

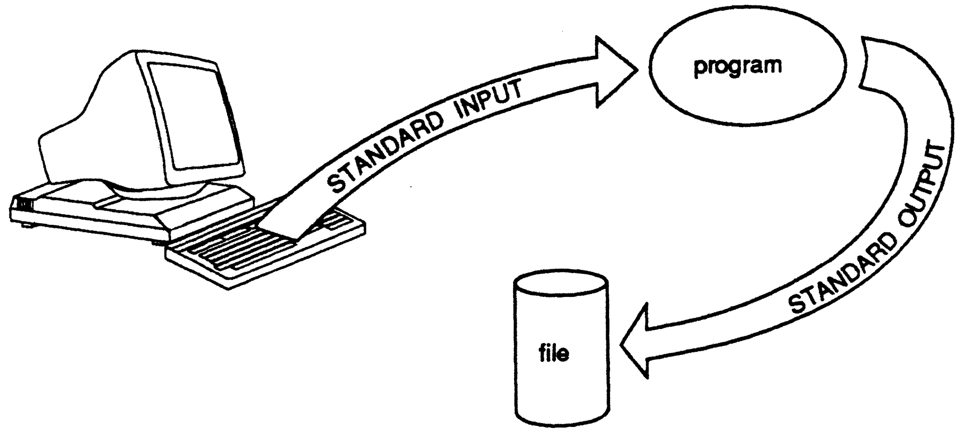


Figure 4-2. Standard output can be redirected from your terminal to a file.

To use the > operator, follow the command line format:

command > newfile<CR>

in which you can choose to surround the > operator with spaces as indicated in the command line or leave the spaces out (**command>newfile<CR>**); either method is correct.

For example, if you have two files, named *group1* and *group2* each containing a list of names with telephone extension numbers that you would like to sort alphabetically and then interfile into a separate file called *members*, you would type:

sort group1 group2 > members<CR>

When you do, the ICON/UXV system first alphabetically sorts and then interfiles the contents of the files *group1* and *group2* and redirects the output into the file called *members* rather than displaying it on your terminal. If you wish to read the contents of the *members* file, you could use the **cat** or **pg** command.

Therefore, if the contents of the file *group1* is:

```
Smith, Allyn     101
Jones, Barbara   203
Cook, Karen      521
Moore, Peter     180
Wolf, Robert     125
```

and the contents of the file *group2* is:

```
Frank, M. Jay     118
Nelson, James     210
West, Donna       333
Hill, Charles     256
Morgan, Kristine  175
```

then the file *members* would appear as follows on your terminal when displayed with the **cat** command.

```
$ sort phase1 phase2 > members<CR>
$ cat members<CR>
Cook, Karen    521
Frank, M. Jay  118
Hill, Charles  256
Jones, Barbara 203
Moore, Peter   180
Morgan, Kristine175
Nelson, James  210
Smith, Allyn   101
West, Donna    333
Wolf, Robert   125
$
```

Keep in mind that if the file to which you are redirecting the standard output already exists, its contents will be replaced with the output of the redirection command.

Redirecting and Appending the Standard Output *(>>)*

Occasionally, you might like to add information to the end of an existing file. You can use the >> operator to do so. Simply input the following command line:

command >> file<CR>

For example, if the file *members* that was created in the previous section was subject to additions and deletions, it might be a good idea to date the list so you know at a glance what version of the list you are using. You could do so by typing

date >> members<CR>

on the command line and the date and time would be printed at the end of the file *members*. Or instead of adding the date to the end of the file *members*, you could have appended another file containing even more names.

Redirecting the Standard Input *(<)*

Standard input can be redirected as well as standard output with the < operator. The general command line format for input redirection is:

command < file<CR>

in which the < operator informs the command (or program) to take input from the file you specify rather than from the terminal keyboard (*Figure 4-3*).

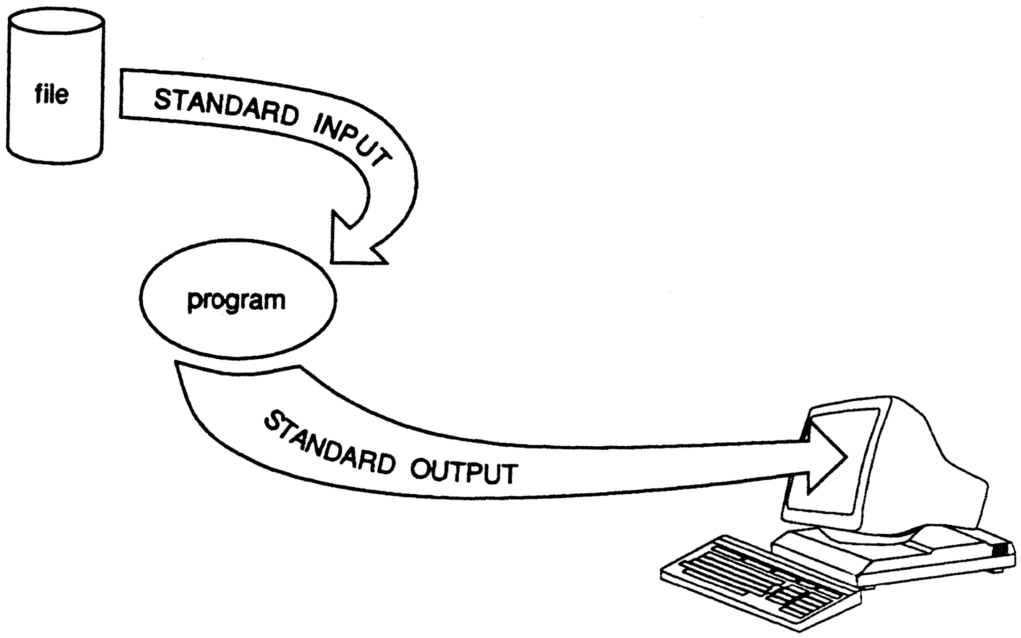


Figure 4-3. You can ask the shell to take a program's input from a file rather than from your terminal.

For example, if you would like to send a copy of the file *members* to co-workers who work on your ICON/UXV system and who have the login names *mary2* and *jmrs*, typing

mail mary2 jmrs < members<CR>

will accomplish the task. The **mail** command, however, does not know whether it received its input from the file *members* (which it did) or from your keyboard. Rather, input/output redirection is a service provided by the ICON/UXV shell and is available to every program. (You will learn more about the **mail** command in *Chapter 8*.)

Connecting Commands with the Pipe *(|)*

The pipe operator is a powerful, yet flexible, mechanism for doing computing tasks quickly and without the need to develop special purpose tools. You can use it to redirect the standard output of one program to be the standard input of another (*Figure 4-4*). Generally, the format for using the pipe is:

command | command<CR>

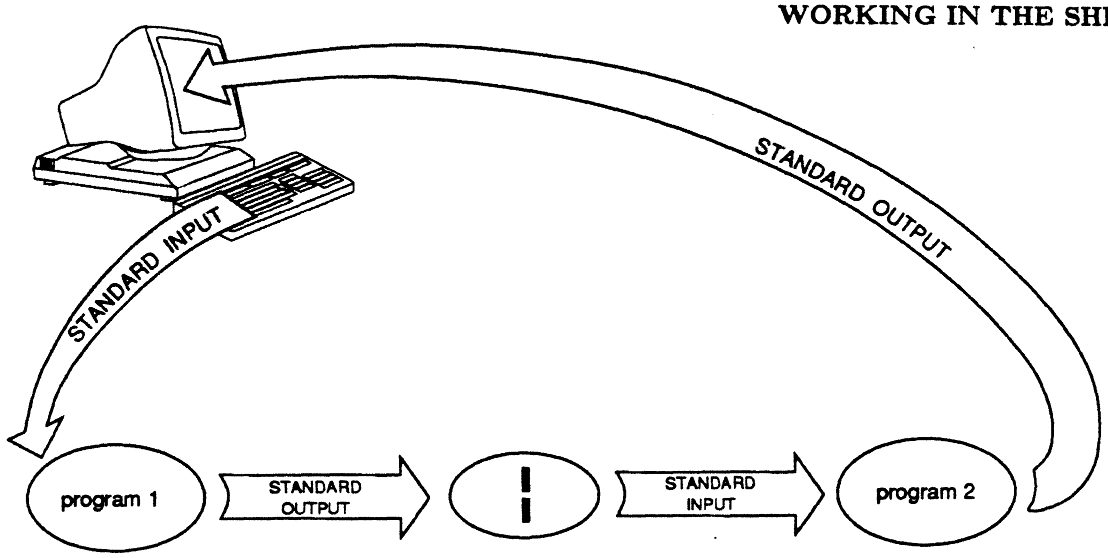


Figure 4-4. You can use the output from one program to be the input for another.

A popular example of this is taking the output of the **who** command (which you were introduced to in *Chapter 2*) and using it as input to the **wc** command (which counts lines, words, and/or characters) as follows:

```
who | wc -l<CR>
```

This example shows that the standard output of the **who** command was passed to the **wc -l** command (**-l** is the option that counts the number of lines output by the **who** command, each corresponding to a user who is logged into your ICON/UXV system.)

Summary

Table 4-3 summarizes which operator performs which redirection task and what general format should be followed in using it. Refer to the section on redirection in *Chapter 7* for details on how to use them.

Running Multiple Programs

There are two methods for running multiple programs: you can specify more than one command to execute in sequence from a single command line or you can run commands simultaneously.

Executing Commands in Sequence

Up to this point, the command lines to which you have been introduced and examples for using them have dealt with asking the shell to run a single request or program. For example, each of the command lines **cat filename<CR>**, **date<CR>**, and **ls -l directoryname<CR>** requests the shell to perform one task. You can, however, ask the shell to execute more than one request per command line. Sequential execution allows you to enter as many commands as you wish on one command line and have them execute in the order in which you input them.

TABLE 4-3

Options for Redirecting Input and/or Output†

Action	Operator	General Format
Redirecting output to a file	>	command > filename
Redirecting and appending output to a file	>>	command >> filename
Redirecting input from a file	<	command < filename
Redirecting output of first command to be input for second	¦	command ¦ command

* *See Chapter 7 for complete details on how to use these options.*

† *Blank spaces immediately before and after redirection operators are optional.*

To do so, you should first be familiar with the general rules for command line syntax given in *Chapter 3.* Briefly, command line syntax orders elements in the command line so that the command name, any options you wish to specify, and the data on which the command is to operate (usually the name of a file or directory) follow one another.

To execute more than one command on a line, simply separate the request sequences with semicolons (;) as follows:

```
command option(s) argument(s); command option(s) argument(s); ...<CR>
```

For example, to determine where you are in the file system and then list the contents of the directory in which you are working, you can type **pwd; ls<CR>** and the terminal monitor might read:

```
$pwd; ls<CR>
/user1/starship/bin
dir
list
tools
$
```

As you can see, the output of the multiple commands is ordered the same way the input is: first, the current working directory is given (in response to **pwd**) and, then, the names of the files and/or directories it holds follow (in response to **ls**).

You could just as easily type **who am i; date; who<CR>** or **mkdir directoryabc; cd directoryabc; pwd<CR>** or any combination of commands that you wish to use.

Executing Commands Simultaneously

In addition to running programs sequentially, you can choose to run them simultaneously. To do so, you need to know the difference between foreground and background commands. When a program runs in the background, the computer is executing that program concurrently with the commands that you enter or with the program that you run in the foreground. However, the computer considers your foreground work more important, in a sense, than your background program. This difference has no perceivable effect on the execution of most programs, but running a job in the background is a useful technique when you wish to run a lengthy or time-consuming job without tying up your terminal.

All the command lines used in this guide until now have been examples of foreground commands. This means that they were initiated and run to completion before other commands could be executed and before the shell would return the **$** prompt for you to continue. However, you also have the option of running a command in the background so you can continue to work in the foreground.

You can run a command in the background by placing an ampersand (&) at the end of the command line as follows:

command option(s) argument(s) &<CR>

When the shell reads the **&**, it starts running the program, prints an identification number, and displays the $ prompt so you can use the terminal immediately for other work.

To save the output from the job you are running in the background, you must redirect the results of the execution into another file so you can look at or use the output when you are ready. For example, if you input the command **cat file1 file2 > file3 &<CR>**, the shell would first give you an identification number, and then the prompt. It will also save the results of **cat file1 file2** in a file named *file3*. When you are ready to peruse *file3*, simply use **cat** or **pg**. If you do not redirect the output, then no output is saved.

When a program is running in the background, it ignores interrupt and break signals, but if you log off, the shell terminates the background program along with the computing session. If you would like to stop a background command while you are still logged into the ICON/UXV system, type **kill id<CR>**, where id is the identification number of the command. On the other hand, to have a program continue to run after you log off, you can use the **nohup** command (which stands for "no hang up") in the following way

nohup command &<CR>

When you do, the command will continue to run until completion and its output is saved in a file called *nohup.out* (which stands for **nohup** output).

Customizing Your Computing Environment

The information in this section deals with another dimension of control provided to you by the shell called your environment. When you log into the ICON/UXV system, the shell automatically sets up a computing environment for you. You can choose to use it as supplied by the system or you can tailor it to meet your needs.

By default, the environment set up by the shell includes the variables:

HOME = your login directory,

PATH = route the shell takes to search for executable files or commands (typically *PATH=:/bin:/usr/bin*), and

LOGNAME = your login name.

If you find the default environment satisfactory, simply leave it as it is and go on with your work. However, if you would like to modify it, you must have a file in your login directory named *.profile*. If you do not, you can create one with a text editor like **ed** or **vi**.

To determine if you have a *.profile*, move to your login directory and type **cat .profile<CR>** and its contents should appear on the terminal monitor. Typically, the *.profile* tests for mail and sets data parameters, system variables, and terminal settings.

Possible modifications to your login environment might include changing your login prompt, setting tab stops, and changing erase and kill characters. If you would like to customize your *.profile*, see the section entitled *Modifying Your Login Environment*, in *Chapter 7*.

COMMUNICATING ELECTRONICALLY

Before the days of office automation, you would probably have thought of relaying a message or information to someone either personally or by way of a letter, note, or telephone conversation. Now as a ICON/UXV user, you can choose to communicate electronically with other ICON/UXV users by way of the computer.

You can send messages or transmit information stored in files to other users who work on your system or on another ICON/UXV system. To do so, your ICON/UXV system must be able to communicate with the ICON/UXV system to which you wish to send information. In addition, the command you use to send information depends on what you are sending.

This guide introduces you to these communication programs:

mail -- This command is typically used for sending messages to others and reading the messages sent to you. You can use **mail** to send messages or files to other ICON/UXV users using their login names as addresses. And, at your convenience, you can use the **mail** command to read messages sent to you by other users. With **mail**, the recipient can choose when to read it.

uuto/uupick -- These commands are used to send and retrieve files. You use the **uuto** command to send a file(s) to a public directory; when its available to the recipient, the person is sent mail telling him/her that the file(s) has arrived. The recipient then can use the **uupick** command to copy the file(s) from the public directory to the directory of choice.

mailx -- This command is a sophisticated, more powerful spin-off of **mail**. It offers a number of options for managing the electronic mail you send and receive.

Chapter 8 teaches you how to use the **mail**, **uuto**, and **uupick** commands. It also introduces you to the **mailx** command so you can begin to use it.

PROGRAMMING IN THE SYSTEM

The ICON/UXV system provides an efficient, effective, and convenient environment for programming and software development. This section briefly describes the environment and your programming options when working in it.

If you are not a programmer, your immediate reaction might be to skip this section. But you need not be a programmer or software developer to enjoy some of the capabilities that fall under the realm of programming.

For example, you can use the shell as a command level programming language as well as the command line interpreter. Shell programming capabilities are useful and usable techniques that allow you to take simple, existing programs and make them more powerful. So why not read on.

On the other hand, if you're interested in sophisticated programming and software development capabilities, this section can serve as a springboard to using them.

What you can expect to find in the next few pages is an overview of shell and C language programming and a mention of other languages that can be used on the ICON/UXV system. In addition, an overview of some ICON/UXV tools for software development is included.

Programming in the Shell

Most interactive users of the ICON/UXV system think of the shell solely as the command language interpreter. The shell, however, is also a command level programming language. What this means is that you can let the shell continue to act as your liaison with the computer or you can program the shell to repeat sequences of instructions and to test certain considerations for you automatically. When you program the shell to perform a task, you use the shell to read and to execute commands that you place in an executable file. These files are sometimes called shell scripts or shell procedures.

When you use the shell in this manner, it provides you with features, like variables, control structures, subroutines, and parameter passing that are very similar to those offered by programming languages. These features provide you with the ability to create your own tools by linking together system commands.

For example, you can write a simple shell procedure from existing ICON/UXV system programs that tells you the date and time along with the number of users working on your ICON/UXV system. One way to do so is illustrated in the following screen:

```
$ cat > users<CR>
date; who | wc -l<CR>
<^d>
$ chmod u+x users<CR>
$
```

A file called *users* is created using the > redirection operator. In the example, **cat** is taking as input everything you type after <CR> on the command line and placing it in a file named *users*. Then the file is made executable with the **chmod** command. If you type the command **users<CR>**, your terminal monitor would look something like the next screen.

```
$ users<CR>
Tues May 22 10:29:09 CDT 1984
   7
$
```

The output tells you that seven users were logged into the system when you typed the command at approximately 10:30 A.M. on Tuesday, May 22.

For additional information on shell procedures and for more sophisticated shell programming techniques, see *Chapter 7, Shell Tutorial,* for step-by-step instructions.

Programming in the C Language

C is a general purpose programming language. It is a relatively "low level" language, which means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the usual arithmetic and logic operators.

C is closely associated with the ICON/UXV system because it was developed on the ICON/UXV system and because ICON/UXV system software is largely written in C.

Although the C programming language is implemented on many computers, it is independent of any particular machine architecture. With a little care, it is easy to write portable programs, that is, programs that can be run without change on a variety of computers if the machine supports a C compiler.

The C programming language comprises the following main elements:

- *Types, operators, and expressions*--Constants and variables are the basic data objects manipulated in a program. Constants are data objects that do not change during the execution of a program, while variables are assigned new values throughout execution. Declarations list variables, state type, and perhaps initial values. Operators specify what is to be done on them. Expressions combine variables and constants to produce new values.

- *Control flow*--Control flow statements of a language specify the order in which computations are done. In C, these include *if-else*, *else-if*, and *switch* statements, and *while*, *for*, and *do-while* loops. In addition, *break*, *continue*, and *goto* statements can be used. Labels can be used as well.

- *Functions and program structure*--C programs generally consist of numerous small functions rather than a few big ones. These functions break large computing tasks into smaller ones and enable you to build on what others have done.

- *Pointers and arrays*--A pointer is a variable that contains the address of another variable. Pointers are frequently used when programming in C because oftentimes they provide the only way to express a computation and partly because their use typically leads to more compact and efficient code than can be obtained in other ways.

- *Structures*--A structure is a collection of one or more variables, possibly of different types, that are grouped together under a single name for convenient handling. Structures help to organize complicated data because they permit a group of related variables to be treated as a unit instead of separate entities.

- *Input and output*--A standard I/O library containing a set of functions designed to provide a standard input and output system is available for C programs. This library is a ICON/UXV feature available for programming in C.

These elements are covered in detail in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Additional information is also available in the *ICON/UXV Programming Guide.*

Other Programming Languages

In addition to C, other programming languages are available for use on the ICON/UXV system, such as FORTRAN-77, BASIC, Pascal, COBOL, APL, LISP, and SNOBOL.

You can obtain details on FORTRAN and its variations in the *ICON/UXV Programming Guide.* Or contact your AT&T Technologies Account Representative for document availability and ordering information on the others.

Tools to Aid Software Development

This section highlights some sophisticated software development tools available on the ICON/UXV system. The tools are designed to make development of software easier and to provide you with a systematic approach to programming.

There are numerous software development aids provided by the ICON/UXV operating system. This section introduces you to five of them to give you an idea of what you can expect development utilities to do. They are:

SCCS --	Source Code Control System,
RJE --	Remote job entry,
make --	Maintaining programs,
lex --	Generating programs for simple lexical tasks, and
yacc --	Generating parser programs.

Refer to the *ICON/UXV Support Tools Guide* and the *ICON/UXV Programming Guide* for more information.

Source Code Control System *(SCCS)*

The Source Code Control System (SCCS) is a collection of ICON/UXV system commands that helps you to control and report changes to source code files or text files. SCCS allows you to access different versions of the same file while maintaining only one file. The way this works is that SCCS stores the original file on a disk. Whenever modifications are made to the file SCCS stores only those changes as a set in something called a *delta.* Each delta or set of changes is numbered to reflect the different versions of a file. You can then choose to retrieve either the original file or a version of the original file.

By allowing SCCS to store and control all iterations of a file, space allocations for storage are minimized and administration of different versions of the same program or document is efficient and simplified. Updates to files can be made quickly and the original version of a program or document is retained if you should need to recall it later.

For additional information, see the *ICON/UXV Support Tools Guide.* Most of the commands needed to use SCCS are documented in the *ICON/UXV User Reference Manual.*

Remote Job Entry *(RJE)*

Remote job entry (RJE) is a software package designed to facilitate communication between a ICON/UXV operating system and an IBM System/360 or an IBM System/370 computer. The RJE software allows the ICON/UXV operating system to communicate with the IBM Job Entry Subsystem by mimicking an IBM System/360 remote multileaving work station. A set of background processes support RJE, and the ICON/UXV system uses these processes to submit jobs for remote execution on the networked IBM system.

When RJE software runs, it does so in the background. It transmits jobs (consisting of job control statements [JCL] and input data) that you queue with the **send** command and status reports you request with the **rjestat** command. In turn, the RJE software subsystem receives print and punch data sets and message output from the IBM system.

For more information on RJE software, see the *ICON/UXV Support Tools Guide.* Commands to be used with RJE are covered in the *ICON/UXV User Reference Manual* and the *ICON/UXV Administrator Reference Manual.*

Maintaining Programs *(make)*

The **make** command is a tool for maintaining, supporting, and regenerating large programs or documents on the basis of smaller ones. Since it is easier to handle and modify small programs, it is recommended that if you wish to develop a large program, you start by creating a series of smaller ones that work together to produce the large one.

The **make** command provides you with a method to store all the information you need to assemble small programs or modules into a large, more sophisticated one. A file called a **makefile** holds the file names of the small programs, the steps necessary to generate the large program, and specifies the dependencies among the files.

When **make** executes the **makefile**, the date and time you last modified any of the small programs are checked and the operations needed to update them are performed in sequence. Then, **make** goes on to create the overall large program.

For details on the operation of **make**, see the *ICON/UXV Support Tools Guide*. Or, for a quick reference, see the entry for **make** in the *ICON/UXV User Reference Manual*.

Generating Programs for Lexical Tasks *(lex)*

The **lex** utility generates programs to be used in simple lexical analysis of text. Lexical analysis is done by evaluating a stream of characters and constructing the forms that are acceptable to the language. Proper forms are defined in the **lex** program and usable forms can be defined by **lex** defaults or by you. **Lex** produces a subroutine as output that must be compiled and combined with other programs to use the lexical analyzer.

The processing done by the **lex** command can be the first step in creating a compiler-type program. In addition, it can be useful as a preprocessing tool for many different software generation functions.

For additional information on the **lex** command, see the *ICON/UXV Support Tools Guide*. A brief description of how **lex** operates and an explanation of its options can be found in the *ICON/UXV User Reference Manual*.

Generating Parser Programs *(yacc)*

The **yacc** program, short for **y**et **a**nother **c**ompiler **c**ompiler, is primarily used in the generation of software compilers. Essentially, **yacc** is a utility for creating parser subroutines. The way this works is that first **yacc** uses specified syntax and produces source code for a parser subroutine. Then, the parser subroutine is compiled, and finally used with a program that calls it to parse input. In this way, structure can be imposed on the input to a program and the desired language can be created from defined rules.

See the *ICON/UXV Support Tools Guide* for details on the **yacc** command. Or refer to the *ICON/UXV User Reference Manual* for some general guidelines on how to use it.

Chapter 5

LINE EDITOR TUTORIAL (ed)

PAGE

Chapter 5

LINE EDITOR TUTORIAL (ed)

INTRODUCING THE LINE EDITOR

This tutorial is an introduction to the line editor, **ed**. The advantages of the line editor are speed and versatility. **ed** requires very little computer time to perform editing tasks. The line editor commands can be typed in by you at a terminal, or they can be used in a shell program. (See *Chapter 7, Shell Tutorial.*)

When you enter **ed**, you are placed in a temporary buffer. The buffer is like a piece of scratch paper for you to work on until you have finished creating or correcting your text in this scratch pad buffer. If you are creating a new file, you enter commands from your terminal that tell **ed** how to create or modify your text in this scratch pad buffer. If you are editing an existing file, a copy of that file is placed in the buffer. Changes are made to the copy of the file. The changes have no effect on the original file until you instruct **ed**, using the "write command", to move the contents of the scratch pad buffer into the file.

You can create text in a file line by line, just as you would on a typewriter. However, **ed** is easier to use than a typewriter because it gives you commands that allow you to change, delete, or add text on several lines in the file, and then display those lines of text on your terminal. You can also add text from another file.

After you have read through this tutorial and have done the examples and exercises, you will have a good working knowledge of **ed**. The following basics will be covered:

- A brief introduction to **ed**, accessing the line editor, creating some text, displaying the lines of text, deleting lines, writing the text to a ICON/UXB file, and quitting **ed**,
- How to address those lines of the file that you want to work on,
- How to display lines of text,
- How to create text,
- How to delete text,
- How to substitute new text for old text,
- How to use special characters as shortcuts for search and substitute patterns,
- How to move text around in the file, and
- Some other useful commands and information.

HOW TO READ THIS TUTORIAL

In this tutorial, commands printed in **bold** should be typed into the system exactly as shown. The system responses to those commands are shown in *italic*. Text that you type into a file is not shown in bold. You should assume that each line you type in at your terminal ends in a carriage return unless the text directs you to do something else. The carriage return is denoted by **<CR>**. As you read the text, you may want to glance back to this section for a quick recap of these conventions.

bold command(Type in exactly as shown.)

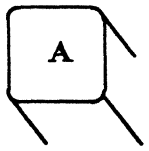
italic response(The system's response to the command.)

roman (Text that is being typed into a file.)

<CR> (Carriage return.)

A display screen or partial screen, like the one above, will be used to illustrate the commands. Because **ed** is versatile and can be used on any type of terminal, you may not be working on a video display terminal. However, the lines you type in, and the system responses are the same whether you are working with a video display terminal or a paper printing terminal.

The **ed** commands are introduced by depicting the corresponding key on your keyboard. The key will appear as shown below in the example of the "a" key.



Notice that the letter on the key appears as it does on your keyboard. However, when you press the key, the letter will appear in lowercase on your terminal. If you need an uppercase letter, the example will include the SHIFT key.

The commands discussed in each section are reviewed at the end of that section. A summary of the **ed** commands discussed in this chapter is found in *Appendix D*, where they are listed in alphabetical order, as well as by topic.

At the end of some sections, exercises are given so you can experiment with the commands. The answers to all of the exercises are at the end of this chapter.

GETTING STARTED

Let's get started. The best way to learn **ed** is to log into the ICON/UXB system and try the examples as you read this tutorial, do the exercises, and do not be afraid to experiment with the **ed** commands. The more you experiment with **ed** commands, the sooner these commands will become second nature to you, and you will have a fast and versatile method of editing text.

In this section, you will learn the bare essentials on how to:

- Access **ed**,
- Append some text,
- Move up or down in the file to display a line of text,
- Delete a line of text,
- Write the buffer to a file, and
- Quit **ed**.

How to Access ed

To access the line editor, type in **ed** and then a file name. The general format for the **ed** command line is:

ed filename<CR>

Choose a file name that reflects what will be in the file. The system will respond with a question mark if this is a new file.

```
$ ed new-file<CR>
? new-file
```

If you are going to edit an existing file, **ed** will respond with the number of characters in the file.

```
$ ed old-file<CR>
235
```

In the above example, the existing file, *old-file*, has 235 characters.

How to Create Text

If you have just accessed **ed**, you are in the command mode of the line editor. **ed** is waiting for your commands. How do you tell **ed** to create some text? Press the "a" key and then a carriage return.

A **Append text.**

If **a** is the only character on a line, it tells the editor that the next characters typed in from the terminal are text for the file. You are now in the text input mode of **ed**. After you have added all the text that you want to the file, type in a period on the line by itself. This takes you out of the text input mode and returns you to the command mode of **ed**, so that you can give **ed** other commands.

The next example shows how to enter **ed** and begin creating text in the new file, *try-me*. The text input mode is then ended with a period.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is a second line,<CR>
and this is the third line.<CR>
.<CR>
```

Notice that **ed** does not give you a response to the period. It just waits for you to enter a new command. If **ed** is not responding to your commands, you may have forgotten to type in the period. Even experienced users sometimes forget to end the text input mode with a period. Type in a period at the beginning of the line. Now **ed** should respond to your commands. If you have added some unwanted characters or lines to your text, you can delete them once you are back in the command mode.

How to Display a Line of Text

How can you display what is in the file? Type in **p**, for print, on a line by itself.

P **Display text.**

Since you have not specified any line number, or line address, **p** will display the current line, that is, the line that was last touched or worked on by **ed**.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is a second line,<CR>
and this is the third line.<CR>
.<CR>
p<CR>
and this is the third line.
```

If you want to see all the lines of text in the file, type in **1,$p**. The **1** and the $ are the line addresses for the first line and the last line of the file. These will be discussed in detail in the section on *Line Addressing.*

```
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
```

Problem:
If you forgot to end the text input mode with the period, you would have added a line of text that you did not want. Try to make this mistake. Add another line of text to your *try-me* file and then try the **p** command without ending the text input mode. Now, end the text input mode and press "p". What did you get? How do you get rid of that line?

```
p<CR>
and this is the third line.
a<CR>
This is the fourth line.<CR>
p<CR>
.<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
p
```

How to Delete a Line of Text

If you are in the command mode of **ed**, press **d** to delete the current line.

D **Delete text.**

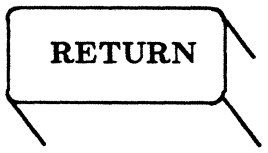
To get rid of the line with the "p" on it, in the last example, delete the line with the **d** command. The next example displays the current line, deletes the current line, and then displays all the lines in the file.

```
p<CR>
p
d<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

After you press **d**, **ed** deletes the current line, but it does so quickly and quietly. It is not evident to you that anything has happened unless you press **p** and find that the current line has been deleted.

How to Move Up or Down a Line in the File

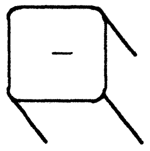
To display the line below the current line, press <CR>.



Display the next line of text.

If there is no line below the current line, **ed** will respond with a *?* and the current line will remain the last line of file. Pressing <CR> is a good way to move down through the buffer.

How do you display the line above the current line? Use the minus key, – .



Display the line of text above the current line.

The next screen demonstrates how to display a line of text, above or below the current line in the file.

```
p<CR>
This is the fourth line.
-<CR>
and this is the third line.
-<CR>
This is a second line,
-<CR>
This is the first line of text.
<CR>
This is a second line,
<CR>
and this is the third line.
```

If you pressed the –<CR> or <CR>, you noticed that the line was displayed without having to press the "p" key. You were addressing a line. If you give a line address and do not follow it with a command, **ed** assumes you want the **p** command, which is the default command for a line address.

Experiment with these commands, create some text, delete a line, and display your file.

How to Save the Buffer Contents in a File

If you have finished editing your text, how do you move it from the buffer, your scratch pad, into a file? To save your text, write the contents of the buffer into a file with the **w** command.

W **Write the contents of the buffer to a file.**

ed will remember the file name you gave when you accessed **ed**, and will write the contents of the buffer to a file with that name. If the file did not already exist, **ed** will create it and then write the contents of the buffer into it.

```
w<CR>
107
```

If the write command is successful, the character count is displayed. In the last example, there are 107 characters of text. When you write a file, you copy the contents of the buffer into the file. The text in the buffer is not disturbed. You can add more text to it. It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to your terminal), you may lose the material in the buffer, but you will not lose the copy written to your file. You can also write to another file name that is different from the one you entered in the **ed** command line. The file name will be a parameter to the **w** command. In the following example, the new file name is *stuff*.

```
w stuff <CR>
107
```

When you return to the shell command mode, display the contents of *stuff* and *try-me*. Are they the same file?

How to Quit the Editor

You have completed editing your file, and have written the editing buffer to the file. To leave the editor and return to the shell command mode, type in the quit command, **q**.

Q **Quit the editing buffer.**

```
w<CR>
107
q<CR>
$
```

The system responds with your shell prompt. At this point, the editing buffer vanishes. Unless you have used the write command, your text in the buffer has also vanished. Since this could be a serious problem, **ed** warns you with a *?* the first time you type in **q** without having written any new changes to a file.

```
q<CR>
?
w<CR>
107
q<CR>
$
```

If you insist on typing in a second **q**, **ed** assumes you do not want to write the changes to the buffer into your file, and returns you to the shell command mode. Your file is left unchanged and the buffer contents are wiped out.

You now know the basic commands to create and edit a file.

SUMMARY OF COMMANDS FOR GETTING STARTED

ed filename	Enter **ed** to edit the file called *filename*.
a	Append text after the current line.
.	End the text input mode, and return to the command mode of **ed**.
p	Display text on your terminal.
d	Delete text.
<CR>	Display the next line in the buffer.
—	Display the line above the current line in the buffer.
w	Write the buffer to the file.
q	Quit **ed** and return to shell command mode.

EXERCISE 1

The answers to all the exercises throughout this chapter are found at the end of this chapter. However, if your method works, if it performs the task even though it does not match the answer given, it is a correct answer.

1-1. Enter **ed** with the file named *junk*. Create a line of text "Hello World", write to the file and quit **ed**.

1-2. Reenter **ed** with the file named *junk*. What was the system response? Was it the same character count as the response to the **w** command in Exercise 1-1.?

Display the contents of the file. Is that your file *junk*?

How do you get back to the shell command mode? Try **q** without writing the file. Why do you think the editor allowed you to quit without writing to the buffer?

1-3. Enter **ed** with the file *junk*. Add a line:

This is not Mr. Ed, there is no horsing around.

Since you did not specify a line address, where do you think the line was added to the buffer? Display the contents of the buffer. Try quitting the buffer without writing to the file. Try writing the buffer to a different file *stuff*. Notice that **ed** does not warn you that the file *stuff* already exists. You have erased the contents of *stuff* and replaced it with new text.

GENERAL FORMAT OF ed COMMANDS

The commands in **ed** have a simple and regular format. Commands are of the form:

[address1,address2]command[parameter]<CR>

The brackets around the addresses and parameter denote that these are optional. The brackets are not part of the command line.

address1,address2
The addresses give the position of lines in the buffer. Address1 through address2 gives you a range of lines that will be affected by the command.

command
The command is one character and tells the editor what task to perform.

parameter
The parameters to a command are those parts of the text that will be modified, or a file name, or another line address.

This general format will become clearer to you when you begin to experiment with the commands in **ed**.

LINE ADDRESSING

Line addresses are very important to **ed**. To add text before or after a line, to delete, move, or change a line, **ed** must know the line address.

[address1,address2]command<CR>

Address2 is given only if you are specifying a range of lines. If address1 is not given, **ed** assumes that the line address is the current line.

A line address is a character or group of characters that identify a line of text. The most common ways to address a line in **ed** are:

- Line numbers, 1 being the first line of the file,
- Special symbols for the current line, last line, and a range of lines,
- Adding or subtracting a number of lines from the current line, and
- A character string or word on that line.

You can access one line, a range of lines, or make a global search for all lines containing a specified character string. A character string is a group of successive characters, such as a word.

Number Line Addresses

ed gives a number address to each line in the buffer. The first line of the buffer is 1, the second line of the buffer is 2 and so on for each line in the buffer. Each line can be accessed by **ed** with the line address number. If you want to see how line numbers address a line, enter **ed** with the file *try-me* and type in a number of a line.

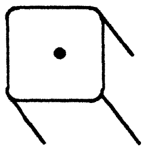
```
$ ed try-me<CR>
107
1<CR>
This is the first line of text.
3<CR>
and this is the third line.
```

Remember that **p** is the default command for **ed**. Since you gave a line address, **ed** assumes you wanted that line displayed on your terminal.

Problem:
Later in this tutorial you will create lines in the middle of the text, or delete lines, or move a line to a different position. This will change the address number of a line. The number of a specific line is always the current position of that line in the editing buffer. If you add five lines of text between line 5 and line 6, once the lines have been added, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Special Symbols Addresses

Current Line Address Character



The address of the current line.

The current line is the line that was most recently acted upon by **ed**, either displayed, created, or moved. If you have just accessed **ed** with an existing file, the current line is the last line of the buffer. The address for the current line is a period. If you want to display the current line, type in: .

If you access **ed** with your file *try-me*, you will find that the current line is the last line. Try it.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
```

The "." is the address. Since no command is given, **ed** assumes the default command **p** and displays the line addressed by " . ".

If you want to know the line number of the current line, you can type in the command:
.=

ed will respond with the line number. In the last example the current line is 4.

```
.<CR>
This is the fourth line.
.=<CR>
4
```

Last Line Address Character

$ **The address of the last line.**

The last line of the file can be addressed by $. It does not matter how many lines are in the file, the last line can always be addressed by $. If you access **ed** with the *try-me* file, you can see that when you first enter **ed** the current line is the last line.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
$<CR>
This is the fourth line.
```

Remember that the $ address within **ed** is not the same as the $ prompt of the shell. If this gets confusing and you want to change your prompt, see *Changing Your Environment* in *Chapter 7, Shell Tutorial.*

Address for the First Line Through the Last Line

The , used as an address will refer to all lines of the file, the first line through the last line.

, **Address all lines of the file.**

If you wanted to display all lines of the file, you could use , as a shortcut address for **1,$**.

```
,p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

Address for the Current Line Through the Last Line

The ; addresses the current line through the last line of the file.

; **Address the range of lines from the current line through the last line.**

The ; is the same as addressing **.,$**.

```
.<CR>
This is a second line,
;p<CR>
This is a second line,
and this is the third line.
This is the fourth line.
```

Relative Addressing, Adding or Subtracting Lines from the Current Line

If you are in a long file, you may want to address lines with respect to the current line. You can do this by adding or subtracting the number of lines from the current line, thus giving a relative line address.

$\boxed{+}$ **Add a number of lines to the current line address.**

$\boxed{-}$ **Subtract a number of lines from the current line address.**

To see relative line addressing, add several more lines to your file *try-me*. Each line should contain the number of the line.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
a<CR>
five
six
seven
eight
nine
ten
.<CR>
```

Now try adding and subtracting line numbers from the current line.

```
4<CR>
This is the fourth line.
+3<CR>
seven
-5<CR>
This is a second line,
```

What happens if you ask for a line address that is greater than the last line, or you try to subtract a number greater than the current line number? Experiment with a relative line addressing. See what happens.

```
5<CR>
five
-6<CR>
?
.=<CR>
5
+7<CR>
?
```

Notice in the above example that the current line remains at line 5 of the buffer. The current line only changes if you give **ed** a correct address. The *?* response indicates an error. The section on *Other Useful Commands and Information* at the end of this chapter, will discuss getting a help message which describes the error.

Character String Addresses

You can search forward or backward in the file for a line containing a specified character string. The line address is the search delimiter and the character string.

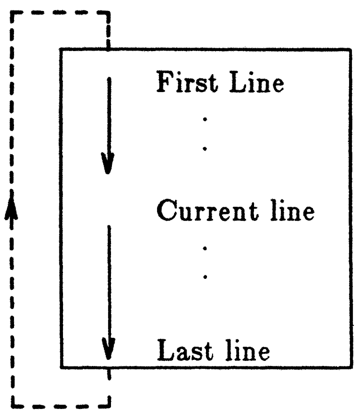
A delimiter gives the boundaries of the character string. Delimiters tell **ed** where a character string starts and ends. The most common delimiter is /. You may also use **?**. If / is used at the beginning of an address **ed** will search forward or down the buffer for the next line containing the specified character string.

Search down or forward in the buffer and address the first line with a specified pattern of characters.

Type in: **/pattern**

ed will search the current line and then down the buffer for the first line that contains the characters **pattern**. If the search reaches the last line of the buffer, **ed** will then wrap around and start searching down the buffer from line 1.

The rectangle below represents the editing buffer. The path of the arrows shows the search initiated by / .

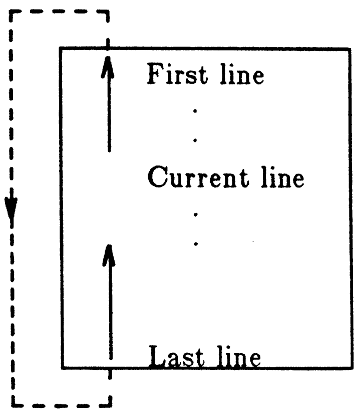


If ? is used at the beginning of an address, **ed** will search backward or up in the buffer for the specified character string.

? **Search up or backward in the buffer and address the first line containing a specified pattern of characters.**

Type in: **?pattern**

ed searches backward from the current line for the first line containing the characters **pattern**. If the search reaches the first line of the file, it will wrap around and continue searching upward from the last line of the file. The next rectangle represents the editing buffer. The path of the arrows shows the search initiated by ? .



Experiment with these two search address requests on the file *try-me*. What happens if **ed** does not find the search pattern?

```
$ ed try-me<CR>
107
.<CR>
ten
?first<CR>
This is the first line of text.
/fourth<CR>
This is the fourth line.
/junk<CR>
?
```

Once again, since no command was given, **ed** assumes it is the **p** command and displays the line. In the above example when **ed** was asked to search for the pattern **junk**, it could not find **junk** and responded with a *?* .

Try the following sequence of commands.

Type in: **/line<CR>**
/<CR>

What happened?

```
.<CR>
This is the first line of text.
/line<CR>
This is the second line,
/<CR>
and this is the third line.
/<CR>
This is the fourth line.
```

ed remembers the pattern of the last search and looks for that pattern until it is given a new pattern.

Specifying a Range of Lines

There are two ways to address a range of lines. You can specify a range of lines such as address1 through address2, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line number of the first line through the line number of last line of the range. These numbers are separated by a comma and placed before the command. If you want to display lines four through ten of the editing buffer, you would give address1 as 4 and address2 as 10.

Type in: **4,10p<CR>** If you are editing the file *try-me*, how would you display lines one through five?

```
1,5p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
five
```

Did you try typing in **1,5** without the **p**? What happened? If you do not add the **p** command, **ed** only prints out address2, the last line of the range of addresses.

You can also use relative line addressing for a range of lines. Be careful, address1 must come before address2 in the buffer. The relative addresses are calculated from the current line.

```
.<CR>
This is the fourth line
-2,+3p<CR>
This is a second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Specifying a Global Search

There are two commands that do not follow the general format of the **ed** commands. They are the global search commands that specify the addresses with a character string.

G

The global search command searches the entire file for lines that contain a specified pattern of characters.

V

The global search command searches the entire file for lines that do NOT contain a specified pattern of characters.

The general format for these two commands gives the command, a delimiter, the search pattern, a delimiter, and a command.

```
g/pattern/command<CR>
v/pattern/command<CR>
```

Try out these commands on *try-me.*

```
g/line/p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line
```

```
v/line/p<CR>
five
six
seven
eight
nine
ten
```

p will act as a default command for the lines addressed by **g** or **v**. If you just want to display the lines, you do not need the last delimiter or **p**.

```
g/line<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line
```

If the lines are used as addresses for other **ed** commands, you will need the beginning and ending delimiters. All of these methods of addressing a line can be used as addresses for **ed** commands.

SUMMARY OF LINE ADDRESSING

1,2...	The number of the line in the buffer.
.	The current line, the last line **ed** touched.
.=	The command that gives the line number of the current line.
$	The last line of the file.
,	Addresses lines 1 through the last line.
;	Addresses the current line through the last line.
+n	Add a number of lines *n* to the current line address.
−n	Subtract a number of lines *n* from the current line address.
/abc/	Search forward in the buffer and address the first line containing the pattern of characters *abc*.
?abc?	Search backward in the buffer and address the first line containing the pattern of characters *abc*.
g/abc/	Address all lines containing the pattern *abc*.
v/abc/	Address all lines that do NOT contain the pattern *abc*.

EXERCISE 2

2-1. Create a file *towns* with the following lines:

```
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
I lost my heart in
San Francisco
I lost $$ in
Las Vegas
```

2-2. Display line 3.

2-3. What lines are displayed for the relative address range −2,+3p ?

2-4. The current line number is? Display the current line.

2-5. The last line says?

2-6. What line is displayed by the search:

?town<CR>

Now type in:

?<CR>

alone on a line. What happened?

2-7. Address all lines that contain the pattern "in". Then address all lines that do NOT contain the pattern "in".

DISPLAY LINES IN A FILE

The two commands that display lines of text in the editing buffer are **p** and **n**.

Display Lines of Text

P — **Print or display lines of text in the editing buffer on your terminal.**

You have already used the **p** command in several examples.

The general form of the print command is:

address1,address2p<CR>

p does not have parameters. However, it can be combined with the substitute command line. This will be discussed later in this chapter.

Experiment with different line addresses and the **p** command on a file in your directory. Try out the following types of addresses.

Type in: **1,$p<CR>**

The entire file should have been displayed on your terminal.

Type in: **–5p<CR>**

The editor should have subtracted 5 from the current line and displayed that line.

Type in: **+2p<CR>**

The editor should have added 2 to the current line and displayed that line.

Type in: **1,/a/p<CR>**

Did you figure out what happened? The editor searched for the next "a" from the current line, and then displayed lines 1 through the first line that contained "a" after the current line.

It is very important to delimit the search pattern to avoid errors in **ed**. You have to delimit the search pattern "a" (enclose "a" between slashes) so that **ed** can tell the difference between the search pattern address "a" and an **ed** command **a**.

Display Lines of Text Preceded by the Line Address Number

N **Display the line address number with the line of text.**

The **n** command is a convenient command when you are deleting, creating, or changing lines. Besides displaying the lines of text, **n** precedes each line with the line address number.

The general format for **n** is the same as **p**.

[address1,address2]n<CR>

Also, like **p**, **n** does not have parameters, but it can be combined with the substitute command. Try out **n** on your test file *try-me*.

```
$ ed try-me<CR>
187
1,$n<CR>
1 This is the first line of text.
2 This is a second line,
3 and this is the third line.
4 This is the fourth line.
5 five
6 six
7 seven
8 eight
9 nine
10 ten
```

Experiment with **n** using different line addresses. In the next example, the relative line addresses **−5** and **+2** are used. Also, the range of lines addressed from line 1 through the first line after the current line that contains an "ne" is also displayed.

```
-5n<CR>
5         five
+2n<CR>
7         seven
1,/ne/n<CR>
1         This is the first line of text.
2         This is a second line,
3         and this is the third line
4         This is the fourth line.
5         five
6         six
7         seven
8         eight
9         nine
```

SUMMARY OF DISPLAY COMMANDS

p	Displays on your terminal the specified lines of text in the editing buffer.
n	Displays on your terminal the line address numbers with the specified lines of text in the editing buffer.

CREATING TEXT

ed has three basic commands for creating new lines of text:

a	Append text,
i	Insert text, and
c	Change text.

Appending Text

Create text after the specified line in the buffer.

You have already used the append command in the *Getting Started* section of this tutorial. The general format for the append command is:

[address1]a<CR>

The default for address1 is the current line. If you do not give **a** an address, **ed** will make address1 the current line.

You have used the default address for **a**, now try using different line numbers for address1. In the next example, a new file called *new-file* is created. The first append command uses the default address. The second append command uses address1 as **1**. The lines are displayed with **n** so that you can see the line addresses.

```
$ ed new-file<CR>
?new-file
a<CR>
Create some lines
of text in
this file.
.<CR>
1,$n<CR>
1  Create some lines
2  of text in
3  this file.
1a<CR>
This will be line 2<CR>
This will be line 3<CR>
.<CR>
1,$n<CR>
1  Create some lines
2  This will be line 2
3  This will be line 3
4  of text in
5  this file.
```

Notice that the address of the line "of text in" changes from two to four after you append the two new lines.

Try out the following special addresses.

.a<CR>	Append after the current line.
$a<CR>	Append after the last line of the file.
0a<CR>	Append text before the first line of the file.

Each of these addresses is used to append text in the following examples.

```
.<CR>
This is the current line
.a<CR>
This line is after the current line.<CR>
.<CR>
-1,.p<CR>
This is the current line.
This line is after the current line.
```

```
$a<CR>
This is the last line now.<CR>
.<CR>
$<CR>
This is the last line now.
```

```
0a<CR>
This is the first line now.<CR>
This is the second line now.<CR>
The line numbers change<CR>
as lines are added.<CR>
.<CR>
1,4n<CR>
1   This is the first line now.
2   This is the second line now.
3   The line numbers change
4   as lines are added.
```

The **0a** command can be replaced by the next command, the insert command.

Inserting Text

The insert command creates text before a specified line in the editing buffer.

I **Insert text before the specified line.**

The general format for **i** is the same as for **a**.

[address1]i<CR>

As with the append command, you can insert one or more lines of text. The text input mode is always ended with a period alone on a line.

The example that follows inserts a line of text above line two; inserts a line of text above the first line; and displays all the lines of the buffer with **n**.

```
2i<CR>
Now this is line 2.<CR>
.<CR>
1,$n<CR>
1  Line 1
2  Now this is line 2
3  Line 2
4  Line 3
5  Line 4
1i<CR>
In the beginning<CR>
1,$n<CR>
1  In the beginning
2  Line 1
3  Now this is line 2
4  Line 2
5  Line 3
6  Line 4
```

Take a few minutes to experiment with the insert command. Try out the special line addresses.

Type in: **.i<CR>**

or

Type in: **$i<CR>**

Changing Text

The change text command erases all of the specified lines and creates new text beginning at address1. You can create one or more lines of text. The change command puts you in the text input mode, so you must end the text input mode by a period alone on a line.

c **Erase specified lines and create new text.**

Since **c** can erase a range of lines, the general format for the change command gives both address1 and address2.

[address1,address2]c<CR>

Addressl is the first line to be erased, and address2 is the last line of the range of lines to be replaced by new text. If you only want to erase one line of text, you would use only addressl. If you do not type in addressl, **ed** assumes the current line is addressl.

The next example changes a range of lines. The first five lines are displayed with **n**. Then lines one through four (**1,4c**) are changed. The lines in the buffer are displayed after the change.

```
1,5n<CR>
1   Line 1
2   Line 2
3   Line 3
4   Line 4
5   Line 5
1,4c<CR>
Change line 1<CR>
and line 2 through 4<CR>
.<CR>
1,$n<CR>
1   Change line 1
2   and line 2 through 4
3   Line 5
```

Now experiment with **c**. Try changing the current line.

```
.<CR>
This is the current line.
c<CR>
I am changing the current line.<CR>
.<CR>
.<CR>
I am changing the current line.
```

If you are not sure you have left the text input mode, it is a good idea to type in the period a second time. If the current line is displayed, you know you are in the command mode of **ed**.

SUMMARY OF CREATE COMMANDS

a	Append text after the specified line in the buffer.
i	Insert text before the specified line in the buffer.
c	Change the text on the specified lines to new text
.	End the text input mode with a period alone on a line, and return to **ed** command mode.

EXERCISE 3

3-1. As an experiment, create a new file *ex3*. Instead of using the append command to create new text in the empty buffer, try the insert command. What happened?

3-2. Enter the file *towns* into **ed**. What is the current line?

Insert above the third line:

Illinois<CR>

Insert above the current line:

or<CR>
Naperville<CR>

Insert before the last line:

hotels in<CR>

Display the text in the buffer preceded by line numbers.

3-3. In the file *towns*, display lines one through five and replace lines two through five with:

London<CR>

Display lines one through three.

3-4. After you have completed exercise 3-3, what is the current line?

Find the line of text containing:

Toledo

Replace:

Toledo

with:

Peoria

Display the current line.

3-5. With one command line search for and replace:

New York

with:

Iron City

DELETING TEXT

This section of the tutorial discusses the delete commands:

d	Delete lines in the command mode;
u	Undo the last command;
# **or <BACK SPACE>**	Delete characters in the text input mode; and
@	Delete a line of text in the text input mode or delete the current command line.

Deleting Lines of Text

You have already deleted lines of text with the delete command **d** in the section of *Getting Started.*

d **Delete one or more lines of text.**

The general format for **d** is:

[address1,address2]d<CR>

You can delete a range of lines, addressl through address2, or you can delete one line using only addressl. If no address is given, **ed** assumes you want to delete the current line.

The next example displays lines one through five and then deletes the range of lines two through four.

```
1,5n<CR>
1  1 horse
2  2 chickens
3  3 ham tacos
4  4 cans of mustard
5  5 bails of hay
2,4d<CR>
1,$n<CR>
1  1 horse
2  5 bails of hay
```

How would you delete only the last line of a file?

```
$d<CR>
```

How would you delete the current line? One of the most common errors in **ed** is forgetting to end the create mode with a period. A line or two of text that you do not want may be added to the buffer. In the next example, the print command is accidentally added to the text before the create mode is ended. Then the current line, the print command, is deleted.

```
a<CR>
Last line of text<CR>
1,$p<CR>
.<CR>
p<CR>
1,$p
.d<CR>
p<CR>
Last line of text.
```

Remember that **1,$p** prints every line of the buffer.

Before you do much experimenting with the delete command, you may first want to learn about the **u** command.

Undo the Last Command

The undo command will erase the effect of the last command and restore any text that had been added, changed, or deleted by that command.

U **Undo the last command.**

If you create new text, change lines of text, delete lines of text, or read new lines into the file, **u** undoes the effect of these commands. (The read command will be discussed in the section on *Moving Text*). Since **u** undoes the last command, it does not have any addresses or arguments. The general form is:

u<CR>

u does not undo the write command or the quit command. However, **u** will undo an undo command.

One example of the **u** command is restoring deleted lines. If you delete all the lines in the file and then type in **p, ed** will respond with a *?* since there are no more lines in the file. Type in **u** and all lines of the file will be restored.

```
1,$d<CR>
p<CR>
?
u<CR>
p<CR>
This is the last line
```

Now try **u** on the append command.

```
.<CR>
This is the only line of text
a<CR>
Add this line<CR>
.<CR>
1,$p<CR>
This is the only line of text
Add this line
u<CR>
1,$p<CR>
This is the only line of text
```

Deleting Commands in the Text Input Mode

Deleting the Current Line

The @ will delete the current line of typing. The line will not be erased from your terminal, but will end with an @ sign and the cursor will move to the next line. When you end the create mode and display the lines of text, the deleted line will not appear.

@ **Delete the current line in the text input mode.**

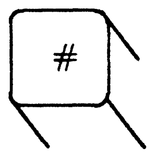
```
a<CR>
I don't want to add this @
a new line of text<CR>
.<CR>
1,$p<CR>
a new line of text
```

The above example begins creating a new file. The first line is deleted in the text input mode, therefore, only the second line is displayed by the **1,$p** command. @ will also delete the current command line. If you make an error typing in a command, type in @ instead of <CR> and **ed** will ignore the command. In the next example, an incorrect address is given, so the command line is cancelled with @.

```
1,$d@
1d<CR>
```

Deleting the Last Characters Typed

If you only made a mistake in typing the last few characters, the # or <BACK SPACE> can delete those characters if you have not pressed <CR>.



Delete the last character just typed into the buffer.

BACK SPACE

Delete the last character just typed into the buffer.

The **<BACK SPACE>** key will delete characters if you have changed your environment to include this command. (See *Chapter 7*, *Shell Tutorial* for changing your environment.)

```
a<CR>
This is a typoo#<CR>
.<CR>
.<CR>
This is a typo
```

In the above example, the extra o in typo was deleted by #. When the line is displayed the error is gone.

You must enter a # for each character that needs to be erased or retyped. In the following example, the error is corrected and new characters follow the last #. (The **<BACK SPACE>** will back up over the characters.)

```
a<CR>
To the IRS, I mail a check<CR>
for one hun###thousand dollars.<CR>
.<CR>
.<CR>
for one thousand dollars.
```

If you press **<CR>** before you correct the error, it is too late to correct the error in the text input mode. However, once you have left the text input mode, the substitute command, discussed in the next section, can solve your problem.

Create a junk file and practice each of these four commands until you are comfortable with them.

SUMMARY OF DELETE COMMANDS

In the command mode:	
d	Delete one or more lines of text.
u	Undo the last command.
@	Delete the current command line.
In the text input mode:	
@	Delete the current line.
# or **<BACK SPACE>**	Delete the last character typed in.

SUBSTITUTING TEXT

You can modify your text with the substitute command **s**.

S **Replace a pattern of characters with new text.**

The substitute command replaces the first occurrence of a string of characters with new text. The general format is:

[address1,address2]s/old text/new text/[command]<CR>

Since this is a more complicated format than the preceding commands, let's look at it piece by piece.

address 1 and address2
The range of lines being addressed by **s**. The address can be one line, address1, a range of lines address1 through address2 or the global search address. If no address is given, **ed** will make the substitution on the current line.

s
: The substitute command, which is positioned right after the line address.

/old text/
: The text to be replaced. It is usually delimited by backslashes, however, it can be delimited by other symbols such as **?** or a period. The **old text** matches the first occurrence of the words or characters to be replaced.

/new text/
: The text that replaces the **old text**. It is placed between the second and third delimiters and replaces the **old text** between the first and second delimiters.

command
: This may be one of four commands that can be placed after the last delimiter. The commands are:

- **g** Change all occurrences of **old text** on the specified lines.
- **l** Display the last line of substituted text including nonprinting characters. (See last section of this chapter entitled *Other Useful Commands and Information.*)
- **n** Display the last line of the substituted text preceded by the line number.
- **p** Display the last line of substituted text.

Substituting on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give the line address for the current line.

s/old text/new text/<CR>

In the next example, a typing error was made on the current line. The example displays the current line, then makes the substitution to correct the error. The **old text** is the **ai** of **airor**, the **new text** is **er**.

```
.p<CR>
In the beginning, I made an airor
s/ai/er/<CR>
.p<CR>
In the beginning, I made an error
```

Did you try out the example? Did you notice **ed** was quiet and gave no response to the substitute command? You either have to display the line with **p** or **n**, or place **p** or **n** on the substitute line. The example below substitutes file for toad.

```
.p<CR>
This is a test toad
s/toad/file/n<CR>
1           This is a test file
```

ed has a short cut for you. If you leave off the last delimiter of the substitute command, the line will automatically be displayed.

```
.p<CR>
This is a test file
s/file/frog<CR>
This is a test frog
```

Substituting on One Line

To substitute on a line that is not the current line, use address1.

[address1]s/old text/new text/<CR>

In this example, the current line is line three. Line one will be corrected.

```
1,3p<CR>
This is a pest toad
testing testing
come in toad
.<CR>
come in toad
1s/pest/test<CR>
This is a test toad
```

Notice that the last delimiter was omitted and **ed** printed out the line.

Substituting on a Range of Lines

If you want to make a substitution on a range of lines, you can specify the first address, address1, through the last address, address2.

[address1,address2]s/old text/new text/<CR>

If **ed** does not find the pattern to be replaced on one of the lines, no changes are made to that line. In the next example, all the lines in the file are addressed for the substitute command. However, only the lines that contain the old text, **es**, are changed.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<CR>

4         tESting 1, 2, 3
```

When you specify a range of lines, **p** or **n** on the substitute line only prints out the last line changed.

To display all the text that was changed use **n** or **p** alone in a command line.

```
1,$n<CR>
1         This is a tESt toad
2         tESting testing
3         come in toad
4         tESting 1, 2, 3
```

Notice only the first occurrence of "es" is changed on line 2. How do you change every occurrence?

Global Substitution

One of the most versatile tools in **ed** is global substitution.

G **Global substitution or search.**

If you place the **g** command after the last delimiter of the substitute command, you will change every occurrence on the specified lines. Try changing every occurrence of **es** in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

```
u<CR>
1,$p<CR>
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another way to do the above example is to use the global search as an address instead of the range of lines one through the last line (**1,$**).

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique, and is the same as the **old text** to be replaced, you can use an **ed** shortcut. You do not need to repeat the pattern for the **old text**. **ed** remembers the search pattern and uses it again as the pattern to be replaced.

g/old text/s//new text/g<CR>

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with the other search pattern addresses:

```
/pattern<CR>
?pattern<CR>
v/pattern<CR>
```

See how they react with the substitute command. In the example below, the **v/pattern** is used to locate the characters **in** that are NOT in the word **testing**.

```
v/testing/s/in/out<CR>
This is a test toad
come out toad
```

If you leave off the last delimiter all search addresses will print out including the ones where no substitution occurs.

```
g/testing/s//jumping<CR>
jumping testing
jumping 1, 2, 3
```

Notice that the global search substitutes for only the first occurrence of testing in each line. The lines are displayed on your terminal because the last delimiter is missing.

EXERCISE 4

4-1. In your file *towns* change **town** to **city** on all lines but the line with **little town** on it.

The file should read:

```
My kind of city is
London
Like being no where at all in
Peoria
I lost those little town blues in
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

4-2. Try using **?** as a delimiter. Change the current line

```
Las Vegas
```

to

```
Toledo
```

You could also use the change command **c**, since you were changing the whole line.

4-3. Try searching backward in the file for the word

```
lost
```

and substitute

```
found
```

using the **?** as the delimiter. Did it work? (The last line of the file is the current line.)

4-4. Search forward in the file for

```
no
```

and substitute

```
NO
```

for it. What happens if you try to use **?** as a delimiter?

Experiment with the various combinations of addressing a range of lines and global searches.

What happens if you try to substitute for the $$? Try to substitute for the $ on line nine of your file.

Type in: **9s/$/Big $<CR>**

What happened?

```
9s/$/Big$<CR>
I found $$ in Big $
```

The substitution did not work correctly because $ is a special character in **ed**. It will be discussed next in the section on special characters.

SPECIAL CHARACTERS

If you tried to substitute for the $ in the line

I lost my $ in Las Vegas

you would find that instead of replacing the $, the new text was placed at the end of the line. The $ is a special character meaning the end of the line.

ed has several special characters that give you a shorthand for search patterns and substitution patterns. The characters act as wild cards. If you have tried to type in any of these characters, the result was probably different than what you had expected.

The special characters are:

.	Match any one character.
*	Match zero or more occurrences of the preceding character.
.*	Match zero or more occurrences of any character following the period.
^	Match the beginning of the line.
$	Match the end of the line.
\	Take away the special meaning of the special character that follows.
&	Repeat the old text to be replaced in the new text of the replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

$\boxed{\bullet}$ **Match any one character.**

The period will represent any one character in a search or substitute pattern. In the next example, a list of animals is searched for the pattern of any letter followed by **at**.

```
1,$p<CR>
rat
cat
turtle
cow
goat
g/.at<CR>
rat
cat
goat
```

Notice that the characters **oat** in goat match **.at**.

The combination of the period and the * is a very potent wild card for the substitution pattern. (See below)

* **Match zero or more occurrences of the preceding character.**

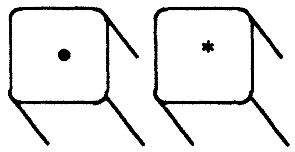
The * is shorthand for a character that is repeated several times in a row in a search or substitute pattern. For example, if you were creating some text and held down a key a little too long, the character would be entered several times into your text. The * is an easy way to substitute one character for those extra characters.

```
p<CR>
brrroke
s/br*/br<CR>
broke
```

It is important to include the **b** in the substitute pattern since * will substitute for *zero* or more occurrences of **r**. Below is an example of using only **r***.

```
p<CR>
brrroke
s/r*/r<CR>
rbrrroke
```

The first zero or more occurrences of r is at the beginning of the line where there are no occurrences of r.



Match zero or more occurrences of any character after the period.

If you combine the period and the *, the combination will match all characters after the period. With this combination you can replace all characters on the last part of a line.

```
p<CR>
Toads are slimy, cold creatures
s/are.*/are wonderful and warm<CR>
Toads are wonderful and warm
```

The .* can also replace all characters between two patterns.

```
p<CR>
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre<CR>
Toads are wonderful and warm creatures
```

^ **Match the beginning of a line.**

If you want to insert a word at the beginning of a line, use the ^ for the old text to be substituted. This is very helpful when you want to insert the same pattern in the front of several lines. The next example places the word **all** at the beginning of each line.

```
1,$p<CR>
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all /<CR>
1,$p<CR>
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```

$ **Match the end of the line.**

This character is useful for adding characters at the end of a line or a range of lines.

```
1,$p<CR>
I love
I need
I use
The IRS wants my
1,$s/$/ money.<CR>
1,$p<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

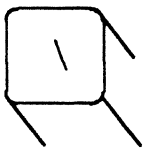
Did you try out the last two examples? Did you remember to put a space after the **all** or before **many**? **ed** adds the characters to the very beginning or the very end of the sentence. If you forgot the space before **money**, your file looks like the following:

```
1,$s/$/money/<CR>
1,$p<CR>
I lovemoney
I needmoney
I usemoney
The IRS wants mymoney
```

The $ is a good way to add punctuation to the end of the line.

```
1,$p<CR>
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./<CR>
1,$pfl/<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

Since . is not matching a character, but replacing a character, it does not have a special meaning in this case. How could you change a period in the middle of a line to another punctuation? You must take away the special meaning of the period in the old text.



Take away the special meaning of the following special character.

If you want to substitute or search for some of the special characters, you must precede them by a \ . To change a **period**, precede the . with a \ .

```
p<CR>
Way to go.  Wow!
s/\. /!<CR>
Way to go!  Wow!
```

Because the backslash is a special character, it too must be preceded by a \ if it is used in the old text.

```
p<CR>
Way to go\  Wow!
s/ \\ /!<CR>
Way to go!  Wow!
```

&

Repeat the old text to be replaced in the new text of the replacement pattern.

If you want to add text without changing the rest of the line, the **&** is a useful shortcut. The **&** repeats the old text in the replacement pattern, so you do not have to worry about typing the correct pattern twice. The next screen shows an example of this.

```
p<CR>
The neanderthal skeletal remains
s/thal/& man's/<CR>
The neanderthal man's skeletal remains
```

%

Repeat the last replacement pattern.

ed automatically remembers the last pattern of characters in a search pattern or the old text in a substitution. But, you must tell **ed** to repeat the replacement characters in a substitution with the **%**. The **%** pattern is very useful if you do not want to make a global change, but you do want to make the same substitution on several different lines. If you want to change money into gold for yourself, but not the IRS, you would repeat the last substitution from line one on line three, but not on line four.

```
1,$p<CR>
I love money
I need food
I use money
The IRS wants my money
1s/money/gold<CR>
I love gold
3s//%<CR>
I use gold
1,$p<CR>
I love gold
I need food
I use gold
The IRS wants my money
```

ed automatically remembers **money**, the old text to be replaced, so it does not have to be repeated between the first two delimiters. The % tells **ed** to use the last replacement pattern, gold.

[. . .] **Match the first occurrence of a character in the bracket.**

ed will try to match one of the characters enclosed in the brackets and substitute the specified old text with new text. The brackets can occur anywhere in the pattern to be replaced.

To conceal the large appetite of the anteater, the zoo keeper quietly altered his file on the animal's dietary habits as shown in the following screen.

```
1,$p<CR>
Monday33,000 ants
Tuesday75,000 ants
Wednesday88,000 ants
Thursday62,000 ants
1,$s/[6789]/4<CR>
Monday33,000 ants
Tuesday45,000 ants
Wednesday48,000 ants
Thursday42,000 ants
```

In the example above, the first occurrence of 6, 7, 8, or 9 was changed to 4 on each line that **ed** found a match.

The next example deletes the Mr or Ms from a list of names.

```
1,$p<CR>
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] //<CR>
1,$p<CR>
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```

[^ . . .]

Match the first occurrence of a character that is not in the brackets.

If the caret is placed as first character in the brackets it tells **ed** to replace characters that are NOT one of these characters. However, if the caret is placed at any other position other than the first character, it will stand for the character ^.

If a copy of John's grades were sent to him as a file in his login, he could enter the file into **ed** and make the following changes to correspond with his own evaluation of his performance.

```
1,$p<CR>
grade  A  Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  D   Jogging
grade  C   Tennis
1,$s/grade [^AB]/grade A<CR>
1,$p<CR>
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  A   Jogging
grade  A   Tennis
```

Whenever you use special characters as wild cards in the old text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only

```
1,$s/[^AB]/A<CR>
```

you would have changed the **g** in **grade** to A. Try it.

As with all commands in **ed**, experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

SUMMARY OF SPECIAL CHARACTERS

.	Match any one character in a search or substitute pattern.
*	Match zero or more occurrences of the preceding character in a search or substitute pattern.
.*	Match zero or more occurrences of any characters following the period.
^	Match the beginning of the line in the substitute pattern to be replaced or in a search pattern.
$	Match the end of the line in the substitute pattern to be replaced.
\	Take away the special meaning of the special character that follows in the substitute or search pattern.
&	Repeat the old text to be replaced in the new text replacement pattern.
%	Repeat the last replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

EXERCISE 5

5-1. Create a file that contains the following lines of text.

```
A   Computer Science
D   Jogging
C   Tennis
```

What happens if you try the command line:

1,$s/[^AB]/A/<CR>

Undo the above command. How would you make the C and D unique? (Hint: they are at the beginning of the line ^.) Do not be afraid to experiment!

5-2. Insert the following line above line 2:

```
These are not really my grades
```

Using brackets and the beginning of the line character ^, create a search pattern that you could use to locate the line you inserted. There are several ways to address a line. When you edit text, use the way that is quickest and easiest for you.

5-3. With one command, change the next three lines

```
I love money
I need money
The IRS wants my money
```

to the following lines:

```
It's my money
It's my money
The IRS wants my money
```

Using two command lines: change the first line from money to gold, change the last two lines from money to gold without using the characters **money** or **gold**.

5-4. How would you change the line

```
1020231020
```

to

```
10202031020
```

without repeating the old digits in the replacement pattern?

5-5. Create a line of characters

```
* . \ & % ^ *
```

Substitute a letter for each character. Did you need to use the backslash for every substitution?

MOVING TEXT

You have now learned to address lines, create and delete text, and make substitutions. **ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. You can also read in text from a file that is not in the editing buffer, or write lines of the file in the buffer to another file in the current directory. The commands that move text are:

m	Move lines of text.
t	Copy lines of text.
j	Join contiguous lines of text.
w	Write lines of text to a file.
r	Read in the contents of a file.

Move Lines of Text

You can move paragraphs of text to another place in the file, or you can move an entire subroutine of a program to another place in the computer program you are creating in **ed**.

M **Move one or more lines of text.**

The general format for the move command is:

[address1,address2]m[address3]<CR>

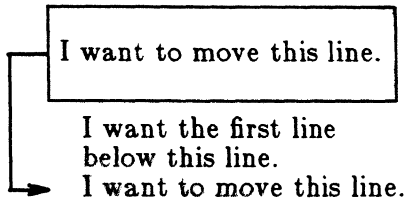
address1,address2	The range of lines to be moved. If only one line is moved, only address1 is given. If no address is given, the current line is moved.
m	The move command.
address3	Place the text after this line.

The following lines are in a file.

```
I want to move this line.
I want the first line
below this line.
```

Type in: **1m3<CR>**

ed will move line 1 below line 3.

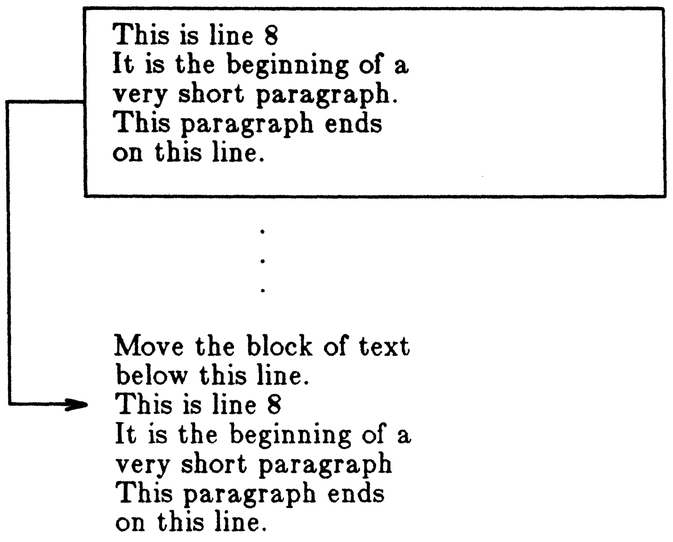


The next screen shows how this will appear on your terminal.

```
1,$p<CR>
I want to move this line.
I want the first line
below this line.
1m3<CR>
1,$p<CR>
I want the first line
below this line.
I want to move this line.
```

If you want to move a paragraph of text, address1 and address2 would be the range of lines of the paragraph.

The following example depicts moving a block of text. Line 8 through line 12 are moved below line 65.



```
This is line 8
It is the beginning of a
very short paragraph.
This paragraph ends
on this line.

      .
      .
      .

Move the block of text
below this line.
This is line 8
It is the beginning of a
very short paragraph
This paragraph ends
on this line.
```

The next screen shows how the command would appear on your terminal. The **n** command is used so that you can see how the line numbers change.

```
8,12n<CR>
8  This is line 8.
9  It is the beginning of a
10 very short paragraph.
11 This paragraph ends
12 on this line.
64,65n<CR>
64 Move the block of text
65 below this line.
8,12m65<CR>
59,65n<CR>
59 Move the block of text
60 below this line.
61 This is line 8.
62 It is the beginning of a
63 very short paragraph.
64 This paragraph ends
65 on this line.
```

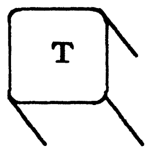
How do you think you would move lines above the first line of the file? Try the following command.

Type in: **3,4m0<CR>**

When address3 is 0, the lines are placed at the beginning of the file.

Copy Lines of Text

The copy command **t** acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text.



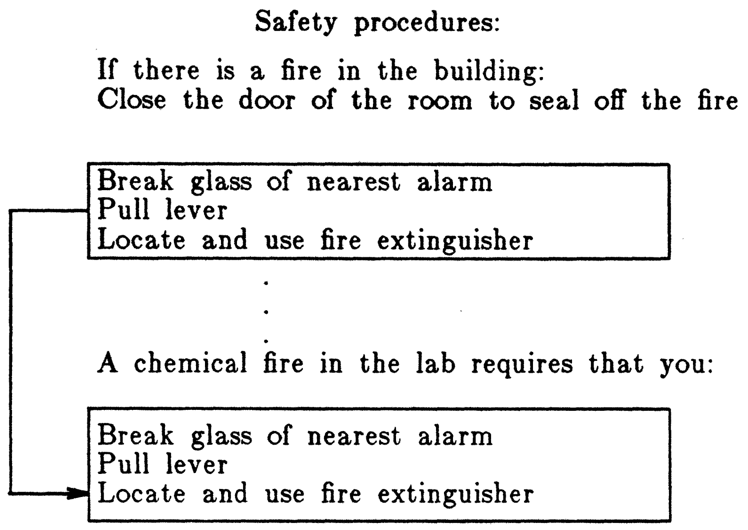
Copy lines of text and place them after a specified line.

The general format of the **t** command also looks like the **m** command.

[address1,address2]t[address3]<CR>

address1,address2	The range of lines to be copied. If only one line is copied, only address1 is given. If no address is given, the current line is copied.
t	The copy command.
address3	Place the copy of the text after this line.

You may want to reiterate a set of directions. You can place a copy of those lines of text below another line in the file. In the next example you want to copy three lines of text below the last line.



Safety procedures:

If there is a fire in the building:
Close the door of the room to seal off the fire

Break glass of nearest alarm
Pull lever
Locate and use fire extinguisher

.
.
.

A chemical fire in the lab requires that you:

Break glass of nearest alarm
Pull lever
Locate and use fire extinguisher

The commands and **ed**'s responses to those commands are displayed in the next screen. The **n** command displays the line numbers.

```
5,8n<CR>
5  Close the door of the room, to seal off the fire.
6  Break glass of nearest alarm
7  Pull lever
8  Locate and use fire extinguisher
30n<CR>
30 A chemical fire in the lab requires that you:
6,8t30<CR>
30,$n<CR>
30 A chemical fire in the lab requires that you:
31 Break glass of nearest alarm
32 Pull lever
33 Locate and use fire extinguisher
6,8n<CR>
6  Break glass of nearest alarm
7  Pull lever
8  Locate and use fire extinguisher
```

The text in lines six through eight remain in place. A copy of those three lines is placed after line 50.

Experiment with **m** and **t** on one of your files.

Joining Contiguous Lines

The **j** command joins the line below the current line with the current line.

J

Join the line below the current line with the current line.

The **j** command does not accept an address, so the general format for the **j** command is:

j<CR>

If the current line is not the line you want joined, the easiest way to make it the current line is to display it with **p** or **n**.

```
1,2p<CR>
Now is the time to join
the team.
p<CR>
the team.
1p<CR>
Now is the time to join
j<CR>
p<CR>
Now is the time to jointhe team.
```

Notice that there is no space inserted between the last word **join** and the first word of the next line **the**. You will have to place the space between them with the **s** command.

Write Lines of Text to a File

If you are writing the same letter to several different people, you may want to keep the body of the text in a special file to use over again. Those lines of text can be written to the special file with the **w** command.

W **Write a copy of the contents of the editing buffer to a file.**

The general format for the **w** command is:

[address1,address2]w [filename]<CR>

address1,address2
The range of lines to be placed into another file. If you do not use address1 or address2, the entire file is written into a new file.

w The write command.

filename The name of the new file that contains a copy of the block of text.

In the next example the body of the letter is saved in a file called *memo*, so that it can also be sent to other people.

```
1,$n<CR>
1         March 17, 1985
2  Dear Kelly,
3  There is a meeting in the
4  green room at 4:30 P.M.
5  today.  Refreshments will
6  be served.
3,6w  memo<CR>
87
```

The **w** command has placed a copy of lines three through six into a new file *memo*. **ed** responds to the **w** command with the number of characters in the new file.

Problem:
If there was a file called *memo* in the current directory, it has been erased. The **w** command will overwrite, that is, erase the current file called *memo*, and put the new block of text in the file without giving any warning. In the next section of this tutorial on *Special Commands*, you will learn how to execute shell commands from **ed**. Then, you can list the file names in the directory to make sure that you are not overwriting a file.

Problem:
You cannot write other lines to the file *memo*. If you tried to add lines 13 through 16, the existing lines (3 through 6) would be erased and the file would only contain the new lines 13 through 16.

Read in the Contents of a File

The body of your memo is in a file called *memo*. How do you copy it from that file into the editing buffer?

R

Read in a copy of the contents of another file into the current editing buffer.

The general format for the **read** command is:

[address1]r filename<CR>

address1	The text will be placed after the line address1. If address1 is not given, the file is added to the end of the buffer.
r	The read command.
filename	The name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen depicts editing a new letter and then reading in the contents of the file *memo*.

```
1,$n<CR>
1         March 17, 1985
2  Dear Michael,
3  Are you free later today?
4  Hope to see you there.
3r memo<CR>
87
3,$n<CR>
3  Are you free later today?
4  There is a meeting in the
5  green room at 4:30 P.M.
6  today. Refreshments will
7  be served.
8  Hope to see you there.
```

ed responds to the read command with the number of characters in the file *memo* that are now added to the editing buffer.

It is always a good idea to display new or changed lines of text to be sure that they are correct.

SUMMARY OF COMMANDS TO MOVE TEXT

m	Move lines of text.
t	Copy lines of text.
j	Join contiguous lines.
w	Write text into a new file.
r	Read in text from another file.

EXERCISE 6

6-1. There are two ways to copy lines of text in the buffer, one is the copy command, the other is writing the lines of text to a file and then reading the file into the buffer. Writing to a file and then reading the file into the buffer is a longer process. Can you think of an example where this would be more practical? What commands would copy lines 10 through 17 of file *exer* into the file *exer6* at line 7?

6-2. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command does this task?

6-3. If you are on line 10 of a file and you want to join lines 13 and 14, what commands would you issue?

OTHER USEFUL COMMANDS AND INFORMATION

There are four other commands and a special file that will be useful to you when you are editing your files. They are the following:

h,H	The help commands that give error messages.
l	Display characters that are not normally displayed.
f	Display the current file name.
!	Temporarily escape **ed** to execute a shell command.
ed.hup	When a system interrupt occurs, the **ed** buffer is saved in a special file named *ed.hup*.

Help Commands

You may have noticed when you were editing a file that **ed** responds to some of your commands with a *?*. The *?* is a diagnostic message indicating there is an error. The help commands give you a short message to explain the reason for the most recent diagnostic.

H **Display a short error message to explain the ? diagnostic.**

There are two help commands.

h	Display a short error message that explains the reason for the most recent *?*.
H	Place **ed** in a help mode that displays the short error message each time *?* is displayed. The next **H** turns off the help mode.

Let's look at an example of **h** first. At the beginning of this tutorial, you learned that if you tried to quit **ed** without writing the changes in the buffer to a file, you would get a *?*. Try it now using **h** to find out what the problem is. When the *?* is displayed, type in **h**.

```
q<CR>
?
h<CR>
warning: expecting 'w'
```

The ? is displayed when you give a new file name to the **ed** command line. Examine that ? with **h** to see what the error message is.

```
ed newfile<CR>
? newfile
h<CR>
cannot open input file
```

This error message is telling you there is no file called *newfile*, or if there is a file named *newfile* **ed** is not allowed to read the file.

Now let's examine the **H** command. This command will respond to the ? and then turn on the help mode of **ed**, so that **ed** will give you an explanation each time the ? is displayed until you turn off the help mode with a second **H**. The next screen shows the help mode turned on by **H**. The various error messages are displayed in response to some common mistakes.

```
e newfile<CR>
? newfile
H<CR>
cannot open input file
/hello<CR>
?
search string not found
1,22p<CR>
?
line out of range
a<CR>
This is line one.
.<CR>
s/$ end of line<CR>
?
illegal or missing delimiter
,$s/$/ end of line<CR>
?
unknown command
H<CR>
q<CR>
?
h<CR>
warning expecting 'w'
```

In the preceding example, the help mode is turned on by **H** and displays the error message for *? newfile.* Then it displays some of the error messages you may encounter in an editing session.

/hello<CR> There is no search pattern **hello** since the buffer is empty.

search string not found

1,22p<CR> There are no lines in the buffer so **ed** cannot print the lines.

line out of range

A line of text is appended to the buffer to show you some error messages associated with the **s** command.

s/$ end of line<CR>
The delimiter between the old text to be replaced and the new text is missing.

illegal or missing delimiter

,$s/$/end of line<CR>
address1 was not typed in before the comma, **ed** does not recognize ,$.

unknown command

The help mode is then turned off and **h** was used to discover the meaning of the last *?* . While you are learning **ed**, you may want to leave the help mode turned on so you will use **H**. However, once you become more adept at editing in **ed**, you will only need to see the error message occasionally and so you will use **h**.

Display Nonprinting Characters

If you are typing in a tab character, normally the terminal will display up to eight spaces to the next tab setting. (Your tab setting may be more or less than eight spaces. See *Chapter 7, Shell Tutorial*, on setting **stty-tabs**.)

If you want to see how many tabs you have inserted into your text, you would use the **l** command.

L **Display nonprinting characters.**

The general format for the **l** command is the same as for **n** and **p**.

[address1,address2]l<CR>

address1,address2
: The range of lines to be displayed. If no address is given, the current line will be displayed. If only address1 is given, only that line will be displayed.

l
: The command that displays the nonprinting characters along with the text.

The **l** command denotes tabs with a > character. **l** displays some control characters. These characters are typed in by holding down the CTRL key and pressing another character key. The key that sounds the bell is control g. It is displayed as \07 which is the ASCII hexadecimal representation (the computer's code) for control g.

Type in two lines of text that contain a control g, denoted in the text by <**^g**>, and a tab denoted by <**tab**>. Then use the **l** command to display the lines of text on your terminal as shown below.

```
a<CR>
Type in <^g> control g.<CR>
Type in a <tab> tab.<CR>
.<CR>
1,2l<CR>
Type in \07 control g
Type in a > tab.
```

Did the bell sound when you typed in <**^g**>?

The Current File Name

In a long editing session, you may forget the file name. The **f** command will remind you which file is currently in the buffer.

Or, you may want to preserve the original file that you entered into the editing buffer and write the contents of the buffer to a new file. In a long editing session, you may forget, and accidentally overwrite the original file with the customary **w** and **q** command sequence. You can prevent this by telling the editor to associate the contents of the buffer with a new file name while you are in the middle of the editing session. This is done with the **f** command and a new file name.

F **Displays or changes the current file name.**

The general format to display the current file name is just **f** alone on a line.

f<CR>

To see how **f** works, enter a file into **ed** and then use the **f** command. The file *oldfile* is entered into **ed** in the example.

```
ed oldfile<CR>
323
f<CR>
oldfile
```

The general format to associate the contents of the editing buffer with a new file name is:

f newfile<CR>

If no file name is given to the write command, **ed** remembers the file name given at the beginning of the editing session and writes to that file. If you do not want to overwrite the original file, you must either use a new file name with the write command, or change the current file name using the **f** command followed by the new file name. Since you can use **f** at any point in the editing session, you can immediately change the currently remembered file name, thus protecting the original file. You can then continue with the editing session without worrying about overwriting the original file.

The next screen shows the commands for entering the editor with *oldfile* and then changing the current file name to *newfile*. A line of text is added to the buffer and then the write and quit commands are given.

```
ed oldfile<CR>
323
f<CR>
oldfile
f newfile<CR>
newfile
a<CR>
Add a line of text.<CR>
.<CR>
w<CR>
343
q<CR>
```

Once you have returned to the shell command mode, you can list your files and see that there is a new file named *newfile*. *newfile* should contain a copy of the contents of *oldfile* plus the new line of text.

Escape to the Shell

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new file name? You need to return to the shell command mode and list your files. The **!** allows you to temporarily return to the shell and execute a shell command line and then return to the current line of the editor.

$\boxed{!}$ **Temporarily escape to the shell.**

The general format for the escape sequence is:

```
!shell command line<CR>
 shell response to the command line
 !
```

When you type in the **!** as the first character on a line, the shell command must follow on that same line. The response to the shell command line will be displayed. When the shell command is finished executing, the **!** will be displayed alone on a line. This tells you that you are back in the editor at the current line.

If you want to return to the shell to find out the correct date, you could type in **!** and the shell command **date**.

```
p<CR>
This is the current line
! date<CR>
mon Apr 1  14:24:22  CST  1988
!
p<CR>
This is the current line.
```

The screen first displays the current line. Then, the command is given to temporarily leave the editor and display the date. After the date is displayed, you are returned to the current line of the editor.

If you want to execute more than one command on the shell command line, see the ; in the section on *Special Characters* in *Chapter 7, Shell Tutorial.*

Recover From a System Interrupt

What happens if you are creating text in **ed** and there is an interrupt to the system, you accidentally hung up on the system, or your terminal was unplugged? Is all lost? When there is an interrupt to the system, the ICON/UXB system trys to save the contents of the editing buffer in a special file named *ed.hup*. You can either use the shell command to move *ed.hup* to another file name, or you can put *ed.hup* back into **ed** and use the **f** command to associate the contents of the editing buffer with a new file name. The next screen shows placing *ed.hup* in **ed** and giving it a new file name.

```
ed ed.hup<CR>
928
f myfile<CR>
myfile
```

Conclusion

You now are familiar with many useful commands in **ed**. The commands that were not discussed in this tutorial, such as G, P, Q and the use of ( ) and { }, are discussed in the *Editing Guide*. Their functions are also listed under the **ed** command in the *ICON/UXB User Reference Manual.* (See *Appendix A.*) You can experiment with these commands and try them out to see what tasks they perform.

SUMMARY OF OTHER USEFUL COMMANDS AND INFORMATION

h	Display a short error message for the preceding diagnostic *?*.
H	Turn on the help mode. An error message will be given with each diagnostic *?*. The second H turns off the help mode.
l	Display nonprinting characters in the text.
f	Display the current file name.
f newfile	Change the current file name associated with the editing buffer to *newfile*.
! cmd	Temporarily escape to the shell to execute a shell command **cmd**.
ed.hup	The editing buffer is saved in *ed.hup* if the terminal is hung up before a write command.

EXERCISE 7

7-1. Create a new file *newfile1*. Once you have entered **ed**, change the current file name to *current1*. Create some text and write and quit **ed**. If you do the shell command **ls** you will see the directory does not contain a file called *newfile1*.

7-2. Create a file named *file1*. Append some lines of text to the file. Leave the append mode. Do not write the file. Turn off your terminal. Turn on your terminal and log in again. Do an **ls** in the shell. Is there a new file *ed.hup*? Place *ed.hup* in **ed**. How do you change the current file name to *file1*? Display the contents of the file. Are the lines the same lines you created before you turned off your terminal?

7-3. While you are in **ed**, temporarily escape to the shell and send a mail message to yourself.

ANSWERS TO EXERCISES

Exercise 1

1-1.

```
$ ed junk<CR>
? junk
a<CR>
Hello world.<CR>
.<CR>
w<CR>
12
q<CR>
$
```

1-2.

```
$ ed junk<CR>
12
1,$p<CR>
Hello world.<CR>
q<CR>
$
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
$ ed junk<CR>
12
a<CR>
This is not Mr. Ed, there is no horsing around<CR>
.<CR>
1,$p<CR>
Hello world.
This is not Mr. Ed, there is no horsing around
q<CR>
?
w stuff<CR>
60
q<CR>
$
```

Exercise 2

2-1.

```
$ ed towns<CR>
? towns
a<CR>
My kind of town is<CR>
Chicago<CR>
Like being no where at all in<CR>
Toledo<CR>
I lost those little town blues in<CR>
New York<CR>
I lost my heart in<CR>
San Francisco<CR>
I lost $$ in<CR>
Las Vegas<CR>
.<CR>
w<CR>
164
```

2-2.

```
3<CR>
Like being no where at all in
```

2-3.

```
-2,+3p<CR>
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
```

2-4.

```
.=<CR>
6
6<CR>
New York
```

2-5.

```
$<CR>
Las Vegas
```

2-6.

```
?town<CR>
I lost those little town blues in
?<CR>
My kind of town is
```

2-7.

```
g/in<CR>
My kind of town is
Like being no where at all in
I lost those little town blues in
I lost my heart in
I lost $$ in

v/in<CR>
Chicago
Toledo
New York
San Francisco
Las Vegas
```

Exercise 3

3-1.

```
$ ed ex3<CR>
?ex3
i<CR>
?
q<CR>
```

The *?* after the **i** indicates there is an error in the command. There is no current line to insert text before that line.

3-2.

```
$ ed towns<CR>
164
.n<CR>
10 Las Vegas
3i<CR>
Illinois<CR>
.<CR>
.i<CR>
or<CR>
Naperville<CR>
.<CR>
$i<CR>
hotels in<CR>
.<CR>
1,$n<CR>
1  my kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
6  Like being no where at all in
7  Toledo
8  I lost those little town blues in
9  New York
10 I lost my heart in
11 San Francisco
12 I lost $$ in
13 hotels in
14 Las Vegas
```

3-3.

```
1,5n<CR>
1  My kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
2,5c<CR>
London<CR>
.<CR>
1,3n<CR>
1  My kind of town is
2  London
3  Like being no where at all
```

3-4.

```
.<CR>
Like being no where at all
/Tol<CR>
Toledo
c<CR>
Peoria<CR>
.<CR>
.<CR>
Peoria
```

3-5.

```
.<CR>
/New Y/c<CR>
Iron City<CR>
.<CR>
.<CR>
Iron City
```

Your search string need not be the entire word or line. It only needs to be unique.

Exercise 4

4-1.

```
v/little/s/town/city<CR>
My kind of city is
London
Like being no where at all in
Peoria
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

The line

I lost those little town blues in

was not printed because it was NOT addressed by the v command.

4-2.

```
.<CR>
Las Vegas
s?Las Vegas?Toledo<CR>
Toledo
```

4-3.

```
?lost?s??found<CR>
I found $$ in
```

4-4.

```
/no?s??NO<CR>
?
/no/s//NO<CR>
Like being NO where at all in
```

You can not mix delimiters such as / and ? in a command line.

Exercise 5

5-1.

```
$ ed file1<CR>
? file1
a<CR>
A  Computer Science<CR>
D  Jogging<CR>
C  Tennis<CR>
.<CR>
1,$s/[^AB]/A/<CR>
1,$p<CR>
AA Computer Science
A  Jogging
A  Tennis
u<CR>
```

```
1,$s/^[^AB]/A<CR>
1,$p<CR>
A  Computer Science
A  Jogging
A  Tennis
```

5-2.

```
2i<CR>
These are not really my grades.<CR>
.<CR>
1,$p<CR>
A   Computer Science
These are not really my grades.
A   Tennis
A   Jogging
/^[^A]<CR>
These are not really my grades
?^[T]<CR>
These are not really my grades
```

5-3.

```
1,$p<CR>
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m<CR>
It's my money
It's my money
```

```
/s/money/gold<CR>
It's my gold
2,$s//%<CR>
The IRS wants my gold
```

5-4.

```
s/10202/&0<CR>
10202031020
```

5-5.

```
a<CR>
* . \ & % ^ *<CR>
.<CR>
s/*/a<CR>
a . \ & % ^ *
s/*/b<CR>
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/\./c<CR>
a c \ & % ^ b
s/\\/d<CR>
a c d & % ^ b
s/&/e<CR>
a c d e % ^ b
s/%/f<CR>
a c d e f ^ b
```

The & and % are only special characters in the replacement text.

```
s/\^/g<CR>
a c d e f g b
```

Exercise 6

6-1. Any time you have lines of text that you may want to have repeated several times, it may be easier to write those lines to a file and read in the file at those points in the text.

If you want to copy the lines into other files you must write them to a file and then read in that file into the buffer containing another file.

```
ed exer<CR>
725
10,17 w temp<CR>
210
q<CR>
ed exer6<CR>
305
7r temp<CR>
210
```

The file *temp* can be called any file name.

6-2.

```
33,46m3<CR>
```

6-3.

```
.=<CR>
10
13p<CR>
This is line 13.
j<CR>
.p<CR>
This is line 13 and line 14.
```

Remember the **.=** will give you the current line.

Exercise 7

7-1.

```
ed newfile1<CR>
? newfile1
f current1<CR>
current1
a<CR>
This is a line of text<CR>
Will it go into newfile1<CR>
or into current1<CR>
.<CR>
w<CR>
66
q<CR>
ls<CR>
bin
current1
rje
```

7-2.

```
ed file1<CR>
? file1
a<CR>
I am adding text to this file.<CR>
Will it show up in ed.hup?<CR>
.<CR>
```

Turn off your terminal.

Log in again.

```
ed ed.hup<CR>
58
f filel<CR>
filel
1,$p<CR>
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

```
ed filel<CR>
58
! mail mylogin<CR>
You will get mail when<CR>
you are done editing!<CR>
.<CR>
!<CR>
```

Chapter 6

SCREEN EDITOR TUTORIAL (vi)

PAGE

Chapter 6

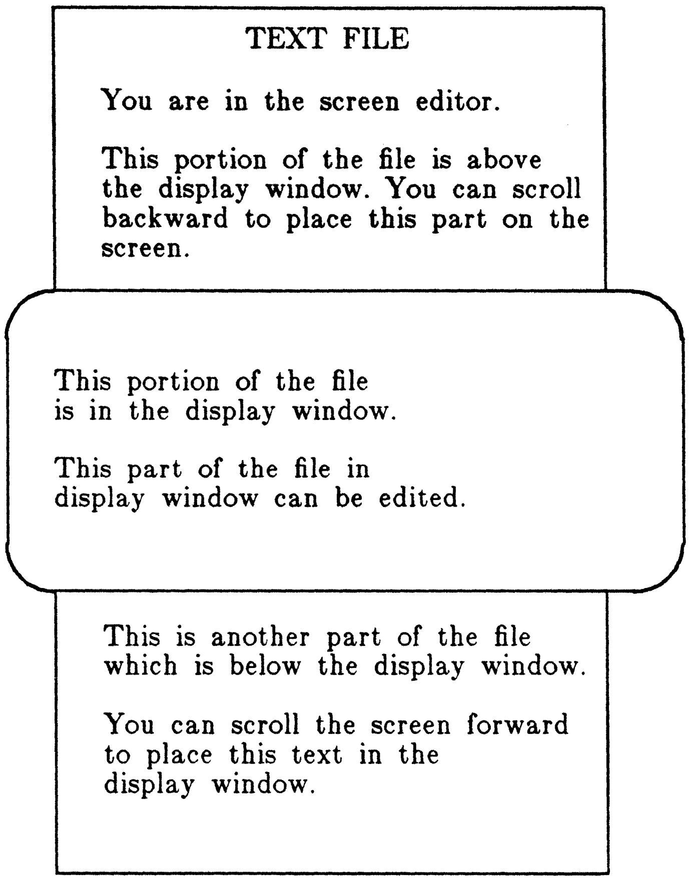
SCREEN EDITOR TUTORIAL (vi)

GETTING ACQUAINTED WITH vi

The screen editor, accessed by the **vi** command, is a powerful and sophisticated tool for creating and editing files. The video display terminal is used as a window to view the text of a file. Within this window, you can add, delete, or change text in much the same way as you would on a typewriter or with paper and pencil. However, making corrections in **vi** does not involve white out, correction tape, or cutting and pasting. A few simple commands change the text, and these changes are quickly reflected in the text on the screen.

The **vi** editor displays from 1 to several lines of text. The cursor can be moved to any point on the screen and text can be created, changed, or deleted from that point. The text in the file can be scrolled forward to reveal the lines below the current window, the window that is on the screen now. Or, the file can be scrolled backward to reveal lines above the current window. (See the display on *page 6-2.*) Other commands can place you at the beginning or end of the file, paragraph, line, or word.

Besides the convenience of editing portions of text within the window, **vi** also gives you the advantage of some line editor commands, such as the powerful global commands that make the same change throughout the whole file.


TEXT FILE
You are in the screen editor.
This portion of the file is above the display window. You can scroll backward to place this part on the screen.
This portion of the file is in the display window.
This part of the file in display window can be edited.
This is another part of the file which is below the display window.
You can scroll the screen forward to place this text in the display window.

Editing window of vi displaying part of a file

HOW TO READ THIS TUTORIAL

This chapter is a tutorial on how to access and use **vi**. Although there are more than 100 commands within **vi**, this tutorial covers only the basic commands that will enable you to effectively use **vi**. The following basics will be covered:

- How to set up your particular type of terminal so you can access **vi**,
- How to get started creating a file, deleting some of your mistakes, writing the text into a ICON/UXV file, and then leaving **vi** to go back to the shell command mode,
- How to move around within the file, so that you can create, delete, or change text,
- How to electronically cut and paste your text,
- How to use some special commands and shortcuts,

- How to temporarily escape to the shell to perform some shell commands and then return to edit the current window of text,
- How to use some line editing commands within **vi**,
- How to quit vi,
- How to edit several files in the same session,
- How to recover a file lost by an interruption to an editing session, and
- How to change your shell environment to automatically set your terminal configuration, and set an automatic carriage return.

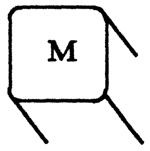
In this tutorial, commands printed in **bold** should be typed into the system exactly as shown. ICON/UXV system responses to those commands are printed in *italic*. The **vi** editor commands that do not print out on the screen will be enclosed in <>. For example, **<CR>** denotes carriage return, meaning press the RETURN key.

The **vi** editor has several commands executed by holding down the "control" or CTRL key while you press another key. These are called control characters. A **^** and a letter denote a control character in the text. For example, **^d** means hold down the control key and press the "d" key. Since **^d** is a command that does not appear on the screen, it will appear in the text as <**^d**>, meaning you should execute **vi** command <**^d**>. As you read the text you may want to glance back for a quick review of these conventions, which are summarized next.

bold command	(Type in exactly as shown.)
italic response	(The system's response to a command.)
roman	(Text that is being typed in a file.)
<CR>	(Commands that are typed in, but not reflected on the screen are enclosed in < >.)
^g	(A control character. Hold down the control key, CTRL, while you press "g".)

In the following sections, a full or partial screen may be used to display the examples showing how the commands are executed. An arrow will point to the letter that is over the cursor. Cursor movements on the screen are depicted by arrows pointing in the direction that the cursor will move.

The keys on your keyboard may be depicted as shown in the example of the "m" key.



Notice that the letter on the key appears as it does on your keyboard. However, when you press the key it will appear in lowercase in your text. If you need an uppercase letter, the example will include the SHIFT key.

The commands discussed in each section are reviewed at the end of the section. A summary of all the **vi** commands is found in *Appendix E*, where they are listed in alphabetical order, as well as by topic.

At the end of some sections, exercises are given for you to experiment with those commands covered in the section. The answers to all of the exercises are at the end of this chapter.

GETTING STARTED

The best way to learn **vi** is to log into the ICON/UXV system and do the examples and the exercises as you read the tutorial. If you experiment with the commands, they will become familiar to you and you will soon be adept at editing in **vi**.

You should be logged into the ICON/UXV system, and ready to create a file in your current directory, the directory you are in now.

How to Set Terminal Configuration

Before you access **vi**, you must set your terminal configuration. That is, you must tell the system what kind of terminal will display the editing window of your file. Each type of terminal has a code name that can be recognized by the system. The code for your terminal is in the ICON/UXV file */etc/termcap*. The *termcap* file contains information about different terminals. You only need to know the code for your terminal, which is the first two letters of the line containing information about your terminal.

To find the code for your type of terminal, use the **grep** command to search the */etc/termcap* file for your terminal type. For example, if you have a TELETYPE 5420 terminal, type in the following from your login directory:

```
$ grep "teletype 5420" /etc/termcap<CR>
T7|5420|tty5420|teletype 5420 80 columns:
$
```

The code for a Teletype 5420 is T7.

To set the terminal configuration, type in:

```
TERM=code<CR>
export TERM<CR>
```

TERM must be typed in uppercase and there are no spaces on either side of the equal sign. "code" will be the first two letters on the line for your terminal from the *termcap* file. In this command sequence, the **export** command assigns the terminal type to your login environment for this session while you are logged in to the ICON/UXV System. You can learn more about exporting variables such as TERM in *Chapter 7, Shell Tutorial* and in *Part 3 Shell Commands.* (See *Appendix A.*)

In the example below, you have logged into the ICON/UXV system and have gotten your $ prompt from the system. Then, you set your terminal configuration for the Teletype 5420.

```
$ TERM=T7<CR>
$ export TERM<CR>
$
```

Look up your terminal code in the *termcap* file, or ask your system administrator for the code. If you set your terminal configuration now, you can do the examples as you read the text.

Do not experiment typing in terminal configurations that do not match your terminal, since you may confuse the ICON/UXV system, and you will either have to log off, hang up, or get the help of the system administrator to restore your login environment.

Later in this chapter, you will learn how to set your shell environment so that you do not have to set the terminal configuration each time that you log in to the ICON/UXV system.

How to Access vi

Now you are ready to access vi.

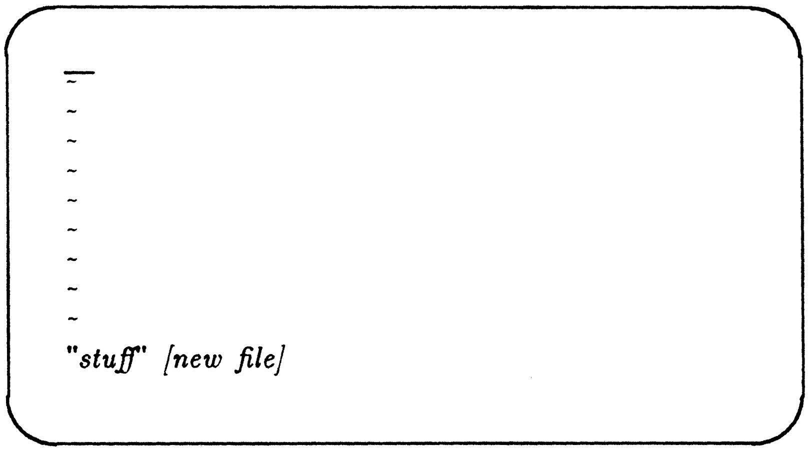
Type in: **vi filename<CR>**

where *filename* is the name of the file you wish to edit, or the name of the file you are about to create.

After you have set your terminal configuration, you want to create a file called *stuff*. For the purpose of this example, **TERM** is set to **T7**.

```
$ TERM=T7<CR>
$ export TERM<CR>
$ vi stuff<CR>
```

The **vi** command will clear the screen and display the window for the screen editor. It should look like this:



```
_
~
~
~
~
~
~
~
~
"stuff" [new file]
```

The **vi** editor window initially displays some lines of text. In this example there are no lines of text. The screen editor displays a ~ on each line to indicate the file is empty. The cursor is at the beginning of the file waiting for the first command. In this example, the cursor appears as a short line. Your video display terminal may indicate the cursor by a blinking line or a reverse color block.

Problem:
If you access **vi** and get the following message you have forgotten to set the terminal configuration.

```
$ vi stuff<CR>
I don't know what kind of terminal you are on - all I have is unknown
[Using open mode]
"stuff" [New file]
```

Type in: **:q<CR>**

This returns you to the shell command mode, now you can set your terminal configuration.

How to Create Text

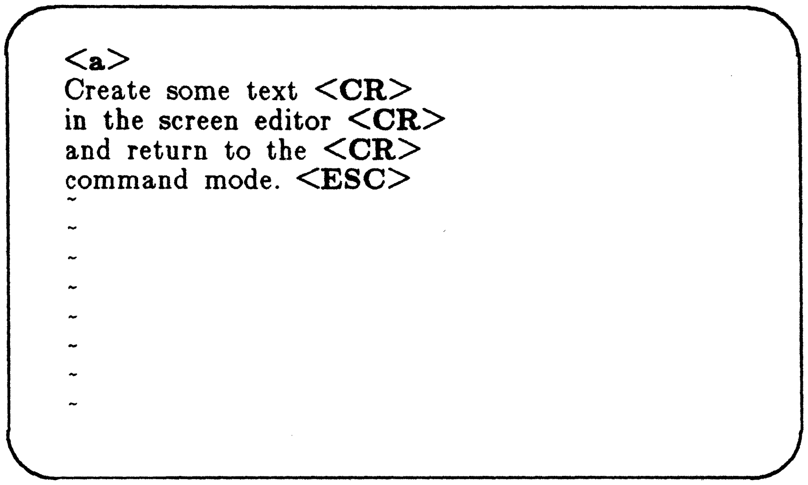
If you have successfully accessed **vi**, you are in the command mode of the screen editor, and **vi** is waiting for your commands. How do you create some text?

- Press the "a" key, **<a>**. Now you are in the append mode of **vi**. You can add text to the file. The **a** does not print out on the screen.
- Start typing in some text.
- To begin a new line press the carriage return key **<CR>**.
- Notice as you get close to the right margin a bell sounds to remind you to press the carriage return. Terminals which do not have a bell, may warn you another way, such as flashing the screen.

It is possible to set the carriage return so that it is automatic; this is discussed later in this chapter in the section on changing your environment.

How to Leave the Append Mode

If you are finished creating text, you need to leave the append mode and return to the command mode of **vi** to edit any text you have created, or to write the text into a ICON/UXV file. Press the escape key, ESC or DEL, denoted by **<ESC>**. You are now back in the command mode.



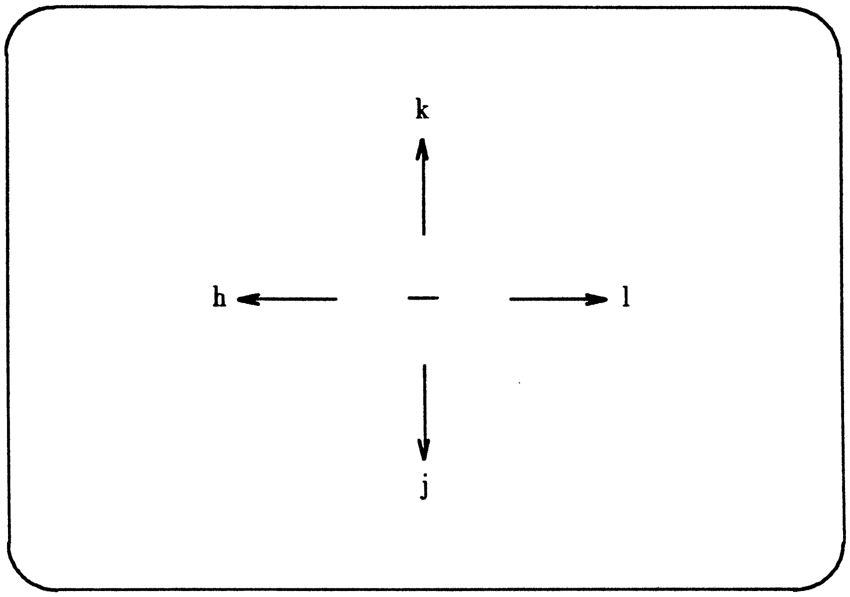
```
<a>
Create some text <CR>
in the screen editor <CR>
and return to the <CR>
command mode. <ESC>
~
~
~
~
~
~
~
~
```

Problem:
If you press <ESC> and a bell sounds, **vi** is telling you that you are already in command mode. It will not affect the text in the file if you press <ESC> several times. The **vi** editor will only sound a bell each time that you press <ESC>.

How to Move the Cursor

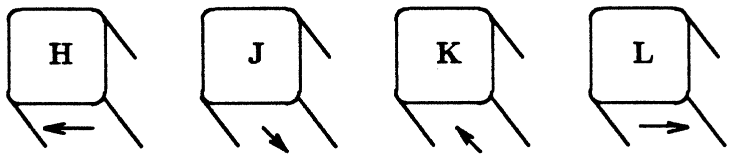
To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are next to each other on the keyboard, "h, j, k, l".

<h>	Moves the cursor one character to the left.
<j>	Moves the cursor down one line.
<k>	Moves the cursor up one line.
<l>	Moves the cursor to the right one character.



Right now try moving the cursor around. Watch the cursor on the screen while you press the keys <**h**>, <**j**>, <**k**>, and <**l**>. If you want to move two spaces to the right, press <**l**> twice. If you want to move up four lines, press <**k**> four times. If you cannot go any farther in the direction you have indicated, **vi** will sound a bell.

Many people who use **vi** find it helpful to mark these four keys with arrows indicating the direction that each key moves the cursor. Mark an arrow on each of four small pieces of white correction tape and place a left arrow on the front of the "h" key, a down arrow on the "j" key, an up arrow on the "k" key, and a right arrow on the "l" key.

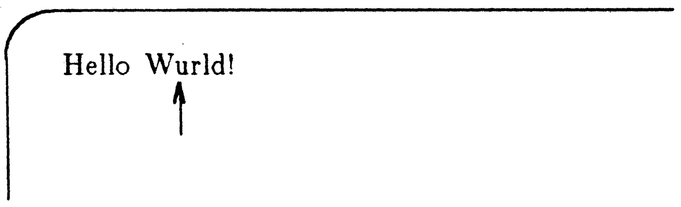


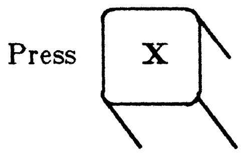
Some terminals have special cursor control keys that are marked with arrows. These may be used as "h, j, k, and l" keys are used.

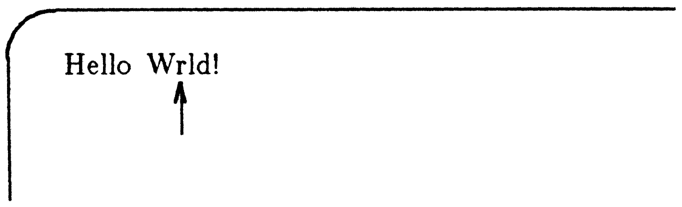
Problem:
If you are trying to move the cursor around on the screen and the letters h, j, k, and l print out on the screen, you are still in the append mode of **vi**. Press **<ESC>**. Most of the commands in the screen editor are silent, that is they do not print out. If the screen editor commands are printing out on the screen you are still in append mode. Press **<ESC>** and try the commands again.

How to Delete Text

If you have put in an extra character in the text, you will want to delete that character. Move the cursor to that character, and press the "x" key. Watch the screen. The letter will disappear and the line will readjust to the change. If you want to erase three letters in a row, press **<x>** three times. In the examples below, the position of cursor is depicted by the arrow under the letter.

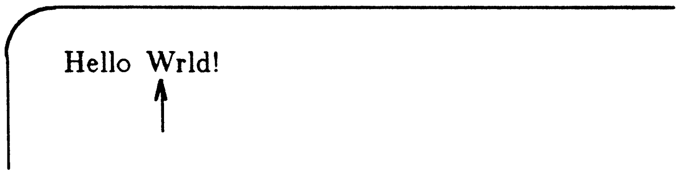




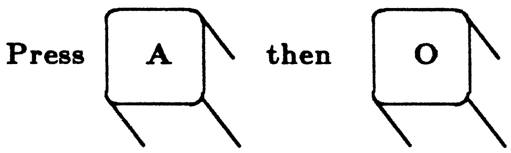


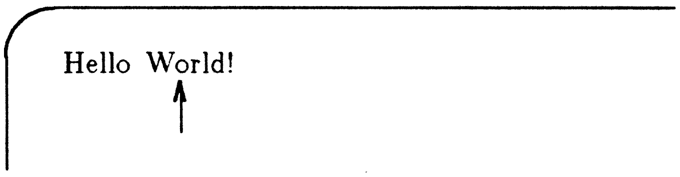
How to Add Text

If you need to add text at a certain point in the text that is in the window, move the cursor to that point using **<h>**, **<j>**, **<k>**, and **<l>**. Then, press **<a>** and text will be created after that point. As you append text, the characters to the right will move over on the screen to make room for the new characters. The **vi** editor will continue adding all characters that you type in, until you press **<ESC>**. If necessary the characters to the right will even wrap around onto the next line.

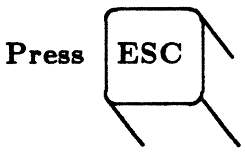


```
Hello Wrld!
```





```
Hello World!
```

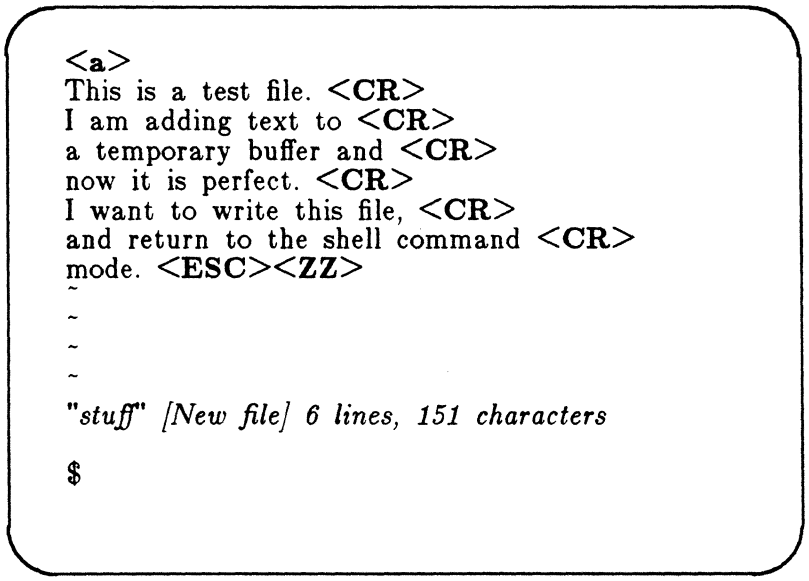


Moving around on the screen, or scrolling through the file to add or delete characters, words, or lines, is discussed in detail later in this tutorial.

How to Quit vi

The **vi** command creates a temporary buffer for you. This is equivalent to giving you a piece of scratch paper. When the text or data on the scratch pad is in the form you want for this editing session, you must write it to a ICON/UXV file. If you are done editing your test file, you will want to put this file in a file called *stuff* in the current directory and get back into the shell command mode.

Hold down the SHIFT key and press the "z" key twice, **<ZZ>**. The **vi** editor remembers the file name given to the **vi** command at the beginning of the editing session, and moves the text from the buffer of the editor to the file named *stuff*. You will get a notice at the bottom of the screen giving the file name, and the number of lines and characters in the file. Then, you are returned to the shell command level, and the ICON/UXV system displays the shell prompt $. Since *stuff* is a new file, the notice at the bottom of the screen will include this fact.



```
<a>
This is a test file. <CR>
I am adding text to <CR>
a temporary buffer and <CR>
now it is perfect. <CR>
I want to write this file, <CR>
and return to the shell command <CR>
mode. <ESC><ZZ>
~
~
~
~
"stuff" [New file] 6 lines, 151 characters

$
```

SUMMARY OF GETTING STARTED

TERM=code **export=TERM**	Set the terminal configuration.
vi filename	Enter **vi** editor to edit the file called *filename*.
<a>	Add text after the cursor.
<h>	Move one character to the left.
<j>	Move down one line.
<k>	Move up one line.
<l>	Move to the right one character.
<x>	Delete a character.
<CR>	Carriage return.
<ESC>	Leave the append mode, and return to **vi** command mode.
<ZZ>	Write to a file, and quit **vi**.
:q	Quit **vi**.

EXERCISE 1

There is often more than one way to perform a task in **vi**. If the way you tried worked, then your answer is correct. Watch the screen as you give the commands, and see how it changes or how the cursor moves.

The answers to the exercises are at the end of this chapter.

1-1. If you have not logged in yet, do so now, and set your terminal configuration.

1-2. Enter **vi** and append the following five lines of text to a new file called *exer1.*

> This is an exercise! Up, down left, right, build your
> terminal's muscles bit by bit.

1-3. Move the cursor to the first line of the file and the seventh character from the right. Notice as you move up the file, the cursor moves "in" to the last letter of the file, but it does not move "out" to the last letter of the next line.

1-4. Delete the seventh and eighth character from the right.

1-5. Move the cursor to the last line of the text, and the last character of that line.

1-6. Append a new line of text.

and byte by byte

1-7. Write the buffer to a file and quit **vi**.

1-8. Reenter **vi** and append two more lines of text to the file *exer1.*

What does the notice at the bottom of the screen say once you have reentered **vi** to edit *exer1*?

POSITIONING THE CURSOR IN THE WINDOW

Until now you have been positioning the cursor with the keys "h, j, k and, l". However, there are several commands to help you move the cursor quickly around the window.

This section on positioning the cursor in the window will look at:

- Positioning by characters on a line,
- Positioning by lines,
- Positioning by text objects
  - By words,
  - By sentences, and
  - By paragraphs, and
- Positioning in the window.

There are also several commands that position the cursor within the **vi** editing buffer. These commands will be looked at in the next section, *Positioning in the File.*

The **vi** editor provides two very helpful patterns in cursor movement.

- Instead of pressing a key such as "h" or "k" a certain number of times, you can precede the command with that number. For example, **<7h>** moves the cursor seven characters to the left.
- Many lowercase commands have an uppercase equivalent that will slightly modify or enhance the command. For example, **<a>** appends text after the cursor, but **<A>** appends text after the last character at the end of the line.

  The uppercase commands will be mentioned briefly in the text, and will be defined in the summary. As you try out the lowercase commands, experiment with the uppercase commands and see what they can do.

If you have not logged into the ICON/UXV system and have not accessed **vi** to edit a file, please do so now. You will want a file that has at least 40 lines in it. If you do not have one, create one now, because you will want to try out each of these cursor movements as you read this section of the tutorial. Remember, to execute these commands, you must be in the command mode of **vi**. Press <**ESC**> to make sure you are out of the append mode, and are in the command mode of **vi**.

Character Positioning

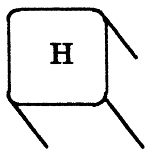
There are three ways to position the cursor by a character on a line.

- You can move the cursor right or left to a character,
- You can specify the character at either end of the line, or
- You can search for a character on a line.

Positioning the Cursor to the Right or Left

The commands, <**h**>, <**l**>, the space bar, and the BACK SPACE key move the cursor right or left to a character on the current line.

You are already familiar with the "h" and "l" keys.

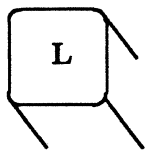


$\longleftarrow$ **Move the cursor to the left.**

<**h**>

$\leftarrow$ Move the cursor one character to the left.

<**nh**> Move the cursor "n" characters to the left.



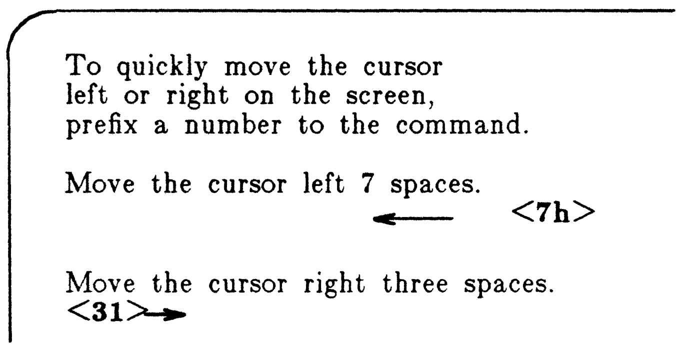
$\longrightarrow$ **Move the cursor to the right.**

<l>

→ Move the cursor one character to the right.

<nl> Move the cursor "n" characters to the right.

Try typing in a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the example below, the cursor movement is depicted by the arrows.



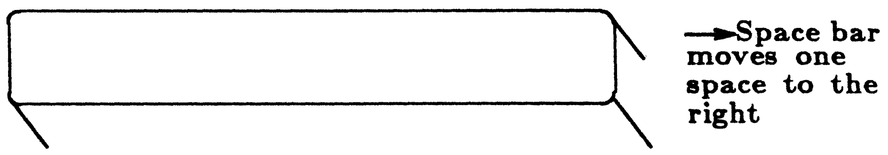
To quickly move the cursor
left or right on the screen,
prefix a number to the command.

Move the cursor left 7 spaces.
← **<7h>**

Move the cursor right three spaces.
<3l>→

Even if there are not 100 characters in a line, if you type in **<100l>**, the cursor will simply travel to the end of the line. If you type in **<100h>** the cursor will travel to the beginning of the line.

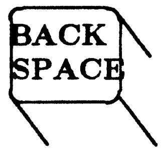
By now, you have probably accidentally discovered that you can move the cursor back and forth on a line using the space bar and the BACK SPACE key.



<space bar>

→ Move the cursor one character to the right.

<nspace bar> Move the cursor "n" characters to the right.



Move the cursor one character to the left.

<BS>

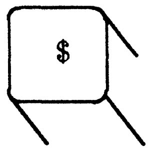
← Move the cursor one character to the left.

<nBS> Move the cursor "n" characters to the left.

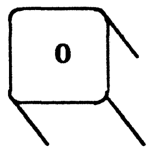
You can type in a number before the space bar or <BS>. The cursor will move that many characters to the left or right.

Positioning the Cursor at the End or Beginning of a Line

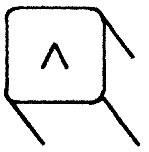
The second method of positioning the cursor on the line is shown below. These commands will place you at the first character or last character of a line.



Position the cursor on the last character of the line.

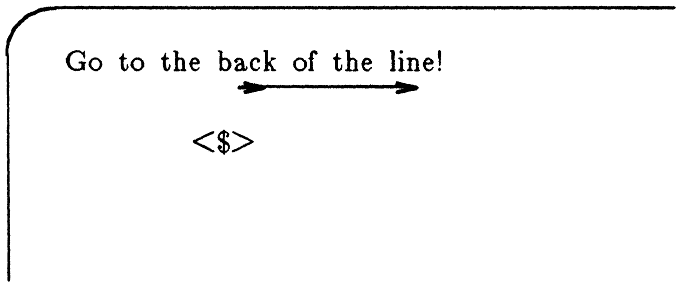


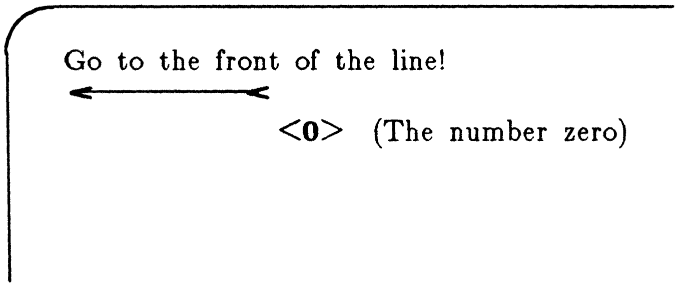
The number zero positions the cursor on the first character of the line.

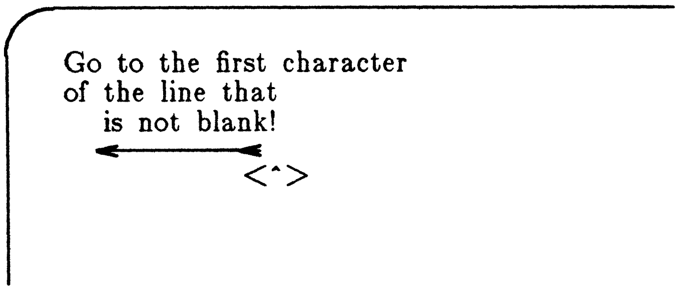


The carat key positions the cursor on the first character of the line that is not a blank. (This is not a control character.)

The next examples show the movement of the cursor for each of the three commands.







Searching for a Character on a Line

The third way to position the cursor on a line is to search for a specific character on the current line. If the character is not on the current line, a bell will sound and the cursor will not move. There is a command that will search the file for patterns. It is discussed in the next section of this tutorial.

F

Moves the cursor to the right to find the specified letter on the current line.

<fx>

→ Move the cursor to the right to the specified character *x*.

<Fx>

← Move the cursor to the left to the specified character *x*.

<;> The <;> will continue the search. It will remember the character and seek out the next occurrence of that character on the current line.

In the next example, **vi** is searching to the right for the first occurrence of the letter "A" on the current line.

Go forward to the letter A on this line.

$\longrightarrow$

<fA>

You may also find the <tx> command useful.

<tx>

→ Move the cursor to the right, to the character just before the specified character *x*.

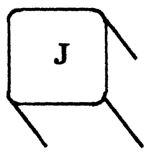
<Tx>

← Move the cursor left to the character just after the specified character *x*.

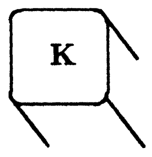
Try the search commands on one of your files. Notice the difference between the uppercase and lowercase commands.

Line Positioning

Besides the <j> and <k> commands that you have already used, the "+", "–" and RETURN keys will move the cursor line by line. The cursor will try to remain at the same position on the line. If the cursor is on the seventh character from the left in the current line, it will try to go to the seventh character on the new line. If there is no seventh character, the cursor will move to the last character.



↓ **Move the cursor down one line.**



↑ **Move the cursor up on line.**

Since you have already tried out <j> and <k> and know how they react, try adding a number of lines to the command as you did with <h> and <l>.

Type in: **7k**

The cursor will move up seven lines above the current line. If there are not seven lines above the current line, a bell will sound and the cursor will remain on the current line.

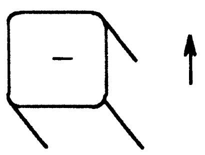
Type in: **35j**

The screen will clear and redraw. The cursor will be on the 35th line below the current line. The new line will be located in the middle of the new window. If there are not 35 lines below the current line, the bell will sound and the cursor will remain on the current line. Try the following command.

Type in: **35k**

Did the screen clear and redraw?

Now, try out the following three easy ways to move up or down in the file.



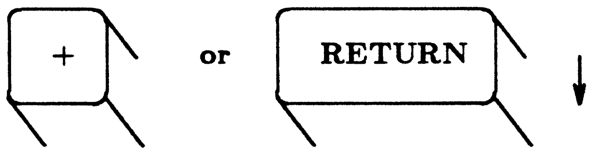
The minus sign moves the cursor up a line.

Type in: **13–**

The cursor will travel up 13 lines. If some of those 13 lines are above the current window, the window will move up to reveal those lines. This is a rapid way to move quickly up the file. Try the following command.

Type in: **100–**

What happened to the window? If there are less then 100 lines above the current line, a bell will sound telling you that you have made a mistake, and the cursor will remain on the current line.



Move the cursor down a line.

Now, try moving down the lines of the file with +.

Type in: **9+**

The cursor will move down nine lines below the current line.

Try moving down line by line in the file with the RETURN key.

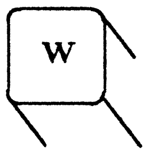
Type in: **5<CR>**

Did the RETURN key give the same response as the "+" key?

Word Positioning

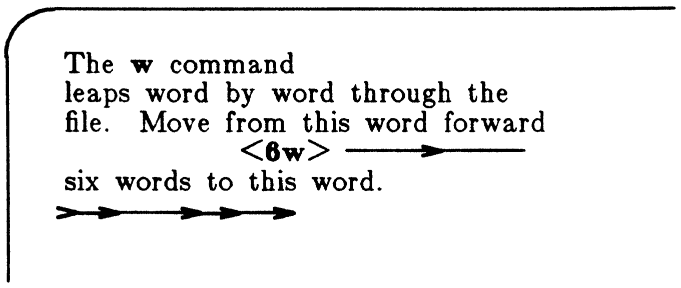
The **vi** editor considers a word a string of characters that are either numbers or letters. The word positioning commands, **<w>**, **<b>**, and **<e>**, consider that any other character is a delimiter, telling **vi** it is the beginning or end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

The uppercase word positioning commands, **<W>**, **<B>**, and **<E>**, consider that the punctuation is part of the word and define a word by all the characters within two blank spaces, that is, the word is delimited by blanks.



Move the cursor to the right by words.

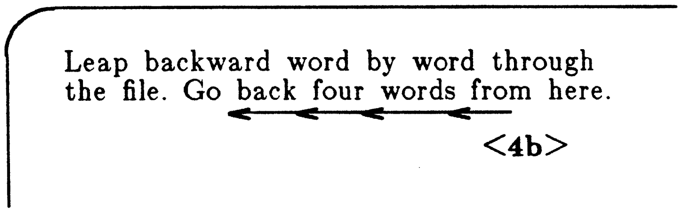
<w>	Move the cursor forward to the first character in the next word. You may press the "w" key as many times as you wish to reach the word you want, or you can prefix the number to the <w> command as shown below.
<nw>	number of words to the first character of that word. The end of the line does not stop the movement of the cursor, it will wrap around and continue to count words from the beginning of the next line.
<W>	Ignore all punctuation, and move the cursor forward to the word after the next blank.



The **w** command
leaps word by word through the
file. Move from this word forward
<6w> ⟶
six words to this word.

B **Move the cursor backwards, to the left, by words.**

<b>	Move the cursor backward one word to the first character of that word.
<nb>	Move the cursor backward "n" number of words to the first character of the nth word. The <b> command does not stop at the beginning of a line, but moves to the end of the line above and continues to move backward.
<B>	Can be used just like the <b> command, except that it delimits the word only by blank spaces. It treats all other punctuation as letters of a word.



Leap backward word by word through
the file. Go back four words from here.
<4b>

E **Move forward to the end of the word.**

The <**e**> command acts like <**w**> moving forward in the file by words, except that it moves the cursor to the end of the word. This makes it easy to add punctuation or add "s" to the end of a word.

The <**E**> command ignores all punctuation except blanks, delimiting the words only by blanks.

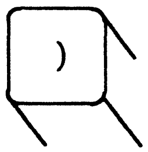
Go forward one word to the end of the next word in this line
<**e**> $\longrightarrow$

Go to the end of the third word.
<**3e**> $\rightarrow\rightarrow\longrightarrow$

Positioning the Cursor by Sentences

The **vi** editor also recognizes sentences. In **vi**, a sentence ends in "! or . or ?". If they appear in the middle of a line, they must be followed by two blanks spaces for **vi** to recognize them. You should get used to the **vi** convention of putting two spaces at the end of each sentence, because you can also delete, change, or yank whole sentences, which will be discussed later in this tutorial.

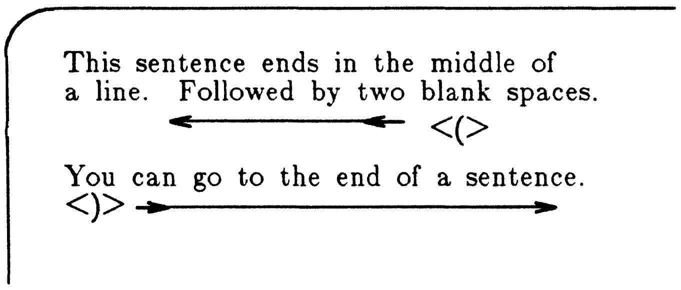
(**Move the cursor to the beginning of a sentence.**



Move the cursor to the beginning of the next sentence.

< (>	Move the cursor to the beginning of the current sentence.
< n(>	Move the cursor to the beginning of the "nth" sentence above the current sentence.
<) >	Move the cursor to the beginning of the next sentence.
< n) >	Move the cursor to the beginning of the "nth" sentence below the current sentence.

In the next example, the arrows show the movement of the cursor.


This sentence ends in the middle of
a line. Followed by two blank spaces.
<(>
You can go to the end of a sentence.
<)>

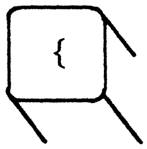
Now, precede the command with a number.

Type in: **3(** or **5)**

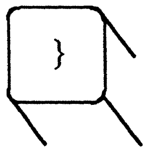
Did the cursor move the correct number of sentences?

Positioning the Cursor by Paragraphs

Paragraphs are recognized by **vi** if they begin after a blank line, or after the paragraph formatting command **.P**. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, delete or change a whole paragraph), then make sure each paragraph ends in a blank line.



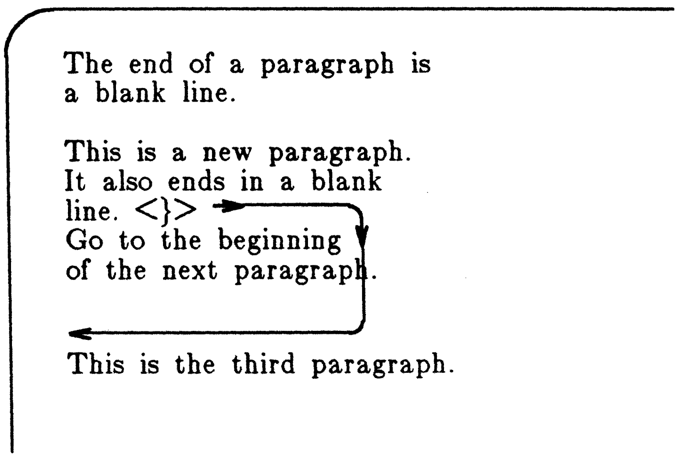
Move the cursor to the beginning of the current paragraph.



Move the cursor to the beginning of the next paragraph.

< { >	Move the cursor to the beginning of the current paragraph, which is delimited by a blank line above it.
< n{ >	Move the cursor to the beginning of the paragraph, "n" number of paragraphs above the current paragraph.
< } >	Move the cursor to the beginning of the next paragraph.
< n} >	Move the cursor to the "nth" paragraph below the current line.

The next example uses arrows to show the cursor moving down to the beginning of the paragraph.



```
The end of a paragraph is
a blank line.

This is a new paragraph.
It also ends in a blank
line. <}>
Go to the beginning
of the next paragraph.

This is the third paragraph.
```

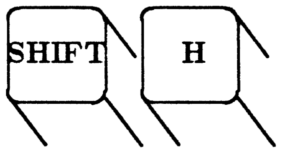
Try moving the cursor with the following commands.

```
Type in:  {
         3{
         6}
```

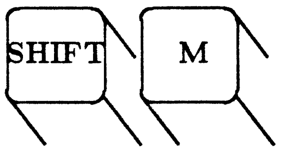
Did you have enough blank lines in your file to test out the last two commands?

Positioning in the Window

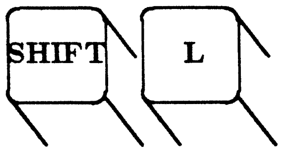
The next three commands help you quickly position yourself in the window. Try out each of the commands.



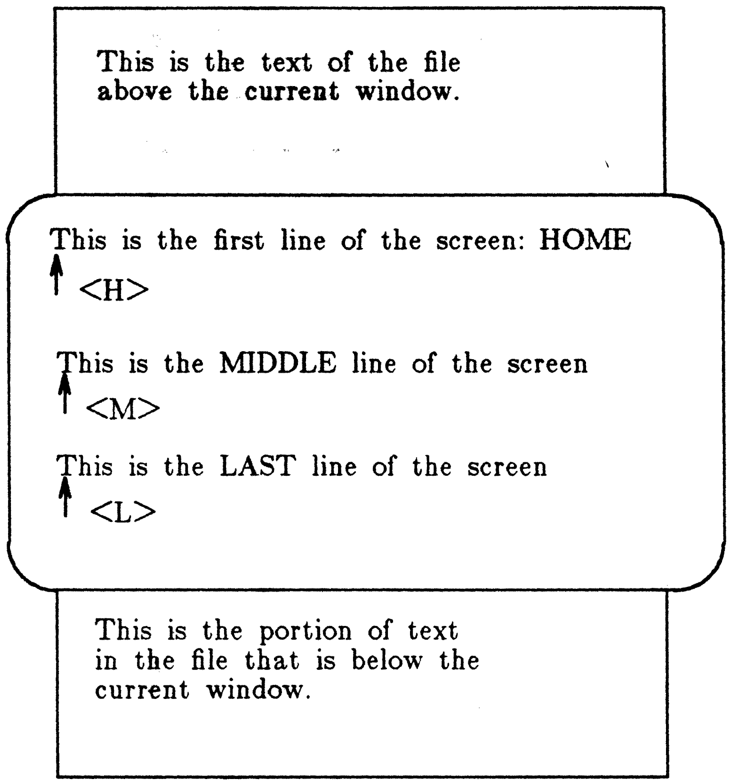
Move the cursor to the first line on the screen.



Move the cursor to the middle line on the screen.



Move the cursor to the last line on the screen.


This is the text of the file above the current window.
This is the first line of the screen: HOME
<H>
This is the MIDDLE line of the screen
<M>
This is the LAST line of the screen
<L>
This is the portion of text in the file that is below the current window.

SUMMARY OF POSITIONING IN THE WINDOW

Character Positioning Commands

<h>

$\leftarrow$ Move the cursor one character to the left.

<l>

$\rightarrow$ Move the cursor one character to the right.

<BS>

$\leftarrow$ Move the cursor one character to the left.

SUMMARY OF POSITIONING IN THE WINDOW

<space bar>	→ Move the cursor one character to the right.
<fx>	→ Move the cursor to the right to the specified character *x*.
<Fx>	← Move the cursor to the left to the specified character *x*.
<;>	Continue the search. It will remember the character and seek out the next occurrence of the character on the current line.
<tx>	→ Move the cursor to the right, to the character just before the specified character *x*.
<Tx>	← Move the cursor left to the character just after the specified character *x*.

Positioning by Lines

<j>	Move the cursor down one line in the same column, if posible.

(Continued on next page)

SUMMARY OF POSITIONING IN THE WINDOW *(continued)*

<k>	Move the cursor up one line in the same column, if possible.
<–>	Move the cursor up one line.
<+>	Move the cursor down one line.
<CR>	Move the cursor down one line.

Word Positioning

<w>	Move the cursor forward to the first character in the next word.
<W>	Ignore all punctuation, and move the cursor forward to the next word delimited only by blanks.
<b>	Move the cursor backward one word to the first character of that word.
<B>	Move the cursor to the left one word, which is delimited only by blanks.
<e>	Move the cursor to the end of the current word.
<E>	Delimit the words by blanks only. The cursor is placed on the last character before the next blank space, or end of the line.

(Continued on next page)

SUMMARY OF POSITIONING IN THE WINDOW *(continued)*

Positioning by Sentences

<(>	Move the cursor to the beginning of the current sentence.
<)>	Move the cursor to the beginning of the next sentence.

Positioning by Paragraphs

<{>	Move the cursor to the beginning of the current paragraph.
<}>	Move the cursor to the beginning of the next paragraph.

Positioning in the Window

<H>	Move the cursor to the first line on the screen, or "home".
<M>	Move the cursor to the middle line on the screen.
<L>	Move the cursor to the last line on the screen.

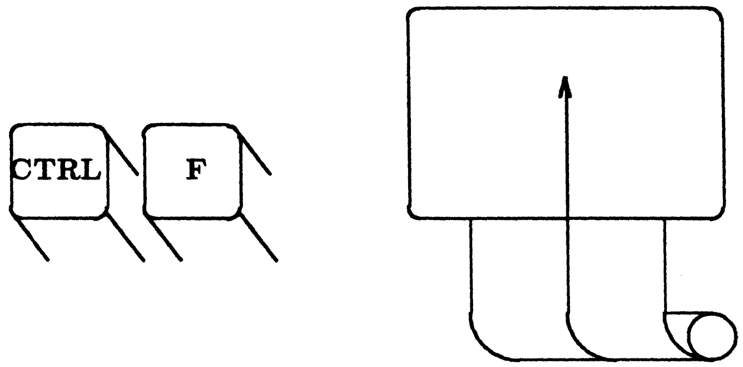
POSITIONING THE CURSOR IN THE FILE

How do you move the cursor to text that is not in the current editing window? You can type in the commands **<20j>** or **<20k>**. However, if you are editing a large file, you need to move quickly and accurately to another place in the file. This section covers those commands that help you move around within the file. You can:

- Scroll forward or backward in a file,
- Go to a specified line in the file, or
- Search for a pattern in the file.

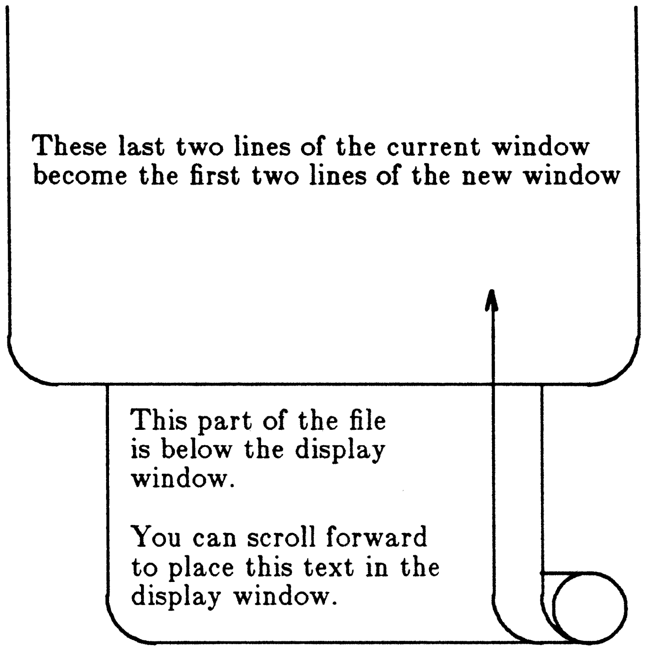
Scrolling the Text

Four basic commands scroll the text of the file. **<^f>** and **<^d>** scroll the screen forward. **<^b>** and **<^u>** scroll the screen backward.

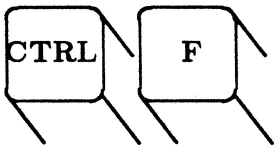


<^f> Scroll the text forward one full window, revealing the window of text below the current window.

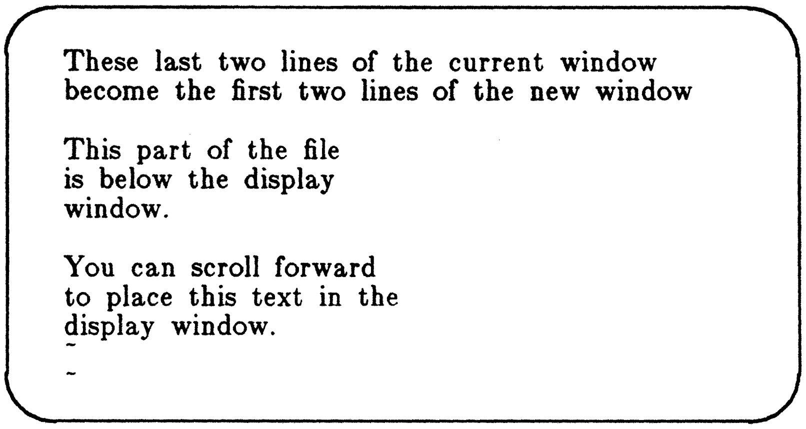
To scroll the file forward, **vi** clears the screen and redraws the window. The last two lines that were at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen will display the ~ to indicate the empty lines.


These last two lines of the current window
become the first two lines of the new window
This part of the file
is below the display
window.
You can scroll forward
to place this text in the
display window.

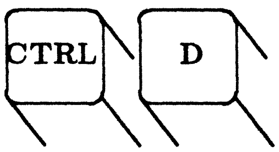
Type in:



vi clears the screen and redraws the new screen shown next.



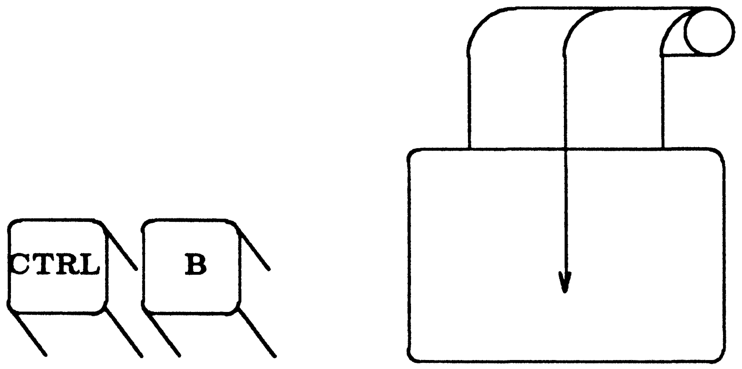
```
These last two lines of the current window
become the first two lines of the new window

This part of the file
is below the display
window.

You can scroll forward
to place this text in the
display window.
~
~
```



Scroll down a half screen to reveal lines below the window.

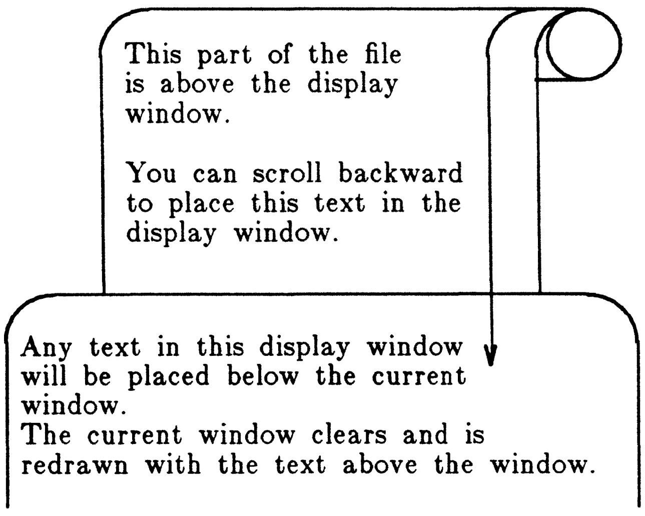
<^d> Scroll down a half screen to reveal text below the window.

When you use <^**d**>, it seems as if the text is being rolled up at the top and unrolling at the bottom to allow the lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell will sound indicating there are no more lines.

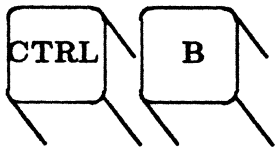


<^b> Scroll the screen back a full window to reveal the text above the current window.

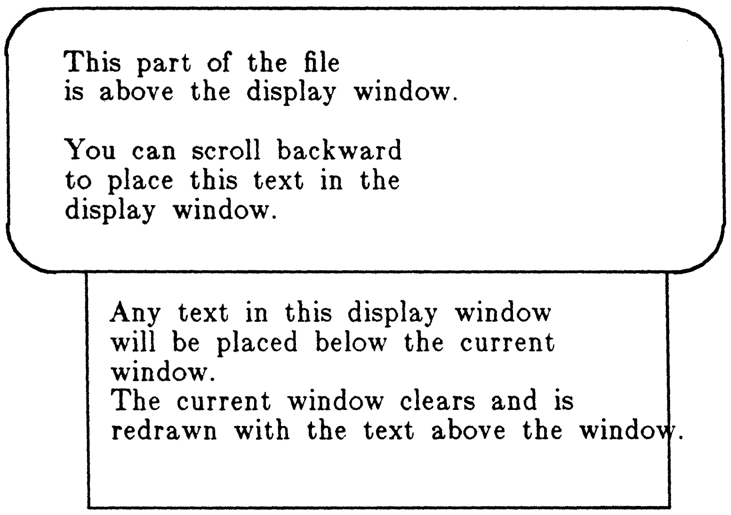
The <^**b**> command clears the screen and redraws the window with the text that is above the current screen. Unlike the <^**f**> command, <^**b**> does not leave any reference lines from the previous window. Also, it does not use the ~ to indicate space above the top of the file. If there are not enough lines above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.


This part of the file is above the display window.
You can scroll backward to place this text in the display window.
Any text in this display window will be placed below the current window.
The current window clears and is redrawn with the text above the window.

Type in:



vi clears the screen and redraws the new screen shown next.

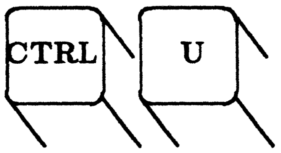


This part of the file
is above the display window.

You can scroll backward
to place this text in the
display window.

Any text in this display window
will be placed below the current
window.
The current window clears and is
redrawn with the text above the window.

Any text that was in the display window is placed below the current window.



Scroll up a half screen to reveal lines above the window.

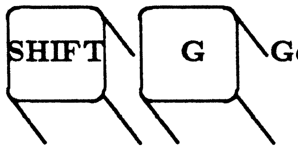
<^u>	Scroll up a half window of text to reveal the lines just above the window. At the same time, the lines at the bottom of the window will be erased.

When you use <^**u**>, it appears as though the text in the file is on a scroll that is being unwound at the top and wound up at the bottom of the screen.

When the cursor is near the top of the file, it will move to the first line of the file and then sound a bell, alerting you it cannot scroll any farther. Try the <^**u**> and <^**d**> commands now. Watch the file scroll through the window.

Go to a Specified Line

The <**G**> command will position the cursor on a specified line in the window, or it will clear the screen and redraw the window around that line. If you do not specify a line, <**G**> will go to the last line of the file.



<G>	Go to the last line of the file.
<nG>	Go to the "nth" line of the file.

Line Numbers

Each line of the file has a line number, that corresponds to the number of lines in the buffer. How can you find out the line numbers? There are two basic ways. One way is to use a line editor command, which you will learn about in the section on the line editor commands. The other way is to position the cursor on the line and type in a <^**g**> command. Try the <^**g**> command now.

The <^**g**> command will give you a status notice at the bottom of the screen. The notice tells you:

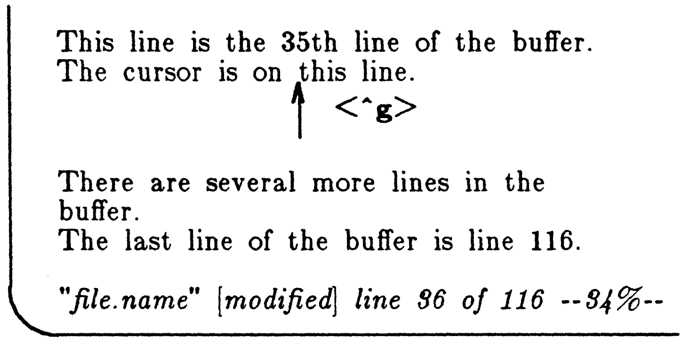
— Name of the file,

— If the line has been changed [modified],

— Line number,

— Number of the last line in the file, and

— Percent the current line is of the total lines in the buffer.



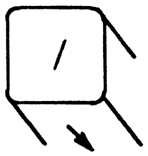
```
This line is the 35th line of the buffer.
The cursor is on this line.
                  ↑ <^g>

There are several more lines in the
buffer.
The last line of the buffer is line 116.
```

"file.name" [modified] line 36 of 116 --34%--

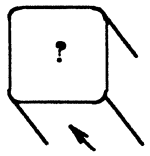
Search for a Pattern of Characters

The fastest way to reach a specific place in your text is to use one of the search commands. You can search forward or backward for the first occurrence of a specified pattern of characters or words in the buffer. The search pattern is ended by <CR>.

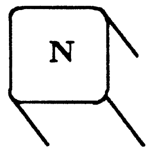
The search commands, / and ?, are not silent. They will print out on the bottom of the screen along with the search pattern. However, the command to repeat the search <n> is silent, it does not print out on the bottom of the screen.



Search forward in the buffer.



Search backward in the buffer.



Repeat the previous search.

/pattern<CR>

Search forward in the buffer for the next occurrence of the characters **pattern**. Position the cursor on the first character of the **pattern**.

/Hello world<CR>

Find the next occurrence in the buffer of the two words **Hello world**. Position the cursor under the **H**.

?pattern<CR>

Search backward in the buffer for the first occurrence of the **pattern**. Position the cursor under the first character of the **pattern**.

?data set design<CR>

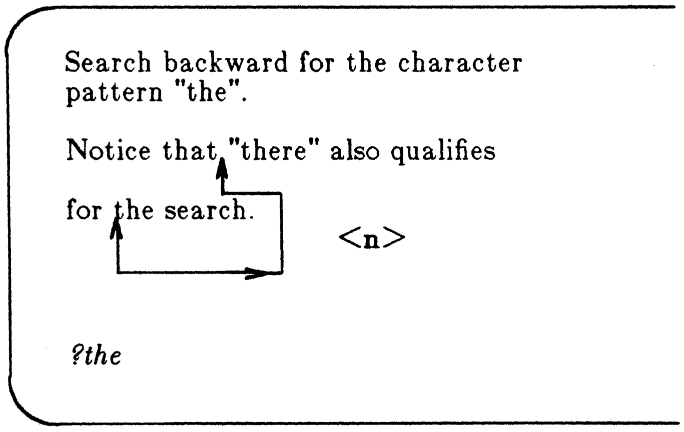
Search backward in the buffer until the first occurrence of **data set design**. Position the cursor under the "d" of **data**.

<**n**>	Repeat the last search command.
<N>	Repeat the search command in the opposite direction.

The search commands will not wrap around the end of the line in searching for two words. If you are searching for "Hello world", and "Hello" is at the end of one line, and "world" is at the beginning of another line, the search commands will not find that occurrence of "Hello world". However, the search commands will wrap around the end or the beginning of the buffer to continue the search. For example, if you are toward the end of the buffer, and the pattern you are searching for with the / command is at the top of the buffer, / will find that pattern.

The <**n**> command continues the last search, remembering the pattern and direction of the search.

The following example shows the results of first typing in **?the** and then typing in <**n**>.



```
Search backward for the character
pattern "the".

Notice that "there" also qualifies

for the search.
                    <n>


?the
```

Experiment for a minute. What happens if you try to type in a number before **?** or / or <**n**>? Experiment with commands in a file called *junk*. If you tried to type in a number before / or **?**, you found out it does not work. However, if you tried to type in <**7n**>, you found out that it searched for the seventh identical pattern.

SUMMARY OF POSITIONING IN THE FILE

Scrolling

<^f>	Scroll the screen forward a full window, revealing the window of text below the current window.
<^d>	Scroll the screen down a half window, revealing lines below the current window.
<^b>	Scroll the screen back a full window, revealing the window of text above the current window.
<^u>	Scroll the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

<G>	Go to the last line of the file.
<^g>	Give the line number and status.

Searching for a Pattern

/pattern	Search forward in the buffer for the next occurrence of the **pattern**. Position the cursor on the first character of the **pattern**.

(Continued on next page)

?pattern	Search backward in the buffer for the first occurrence of the **pattern**. Position the cursor under the first character of the **pattern**.
<n>	Repeat the last search command.
<N>	Repeat the search command in the opposite direction.

EXERCISE 2

2-1. Create a file called *exer2*. Type a number on each line, numbering the lines from 1 to 50. Your files should look similar to the following.

```
1
2
3
4
5
 .
 .
 .
45
46
47
48
49
50
```

2-2. Try using each of the scroll commands, notice how many lines scroll through the window. Try the following:

<^f>
<^b>
<^u>
<^d>

2-3. Go to the end of the file. Append the following line of text.

```
123456789 123456789
```

What number does the command **7h** place the cursor on? What number does the command **3l** place the cursor on?

2-4. Try the command $ and the command **0** (number zero)

2-5. Go to the first character on the line that is not a blank. Move to the first character in the next word. Move back to the first character of the word to the left. Move to the end of the word.

2-6. Go to the first line of the file. Try the commands that place the cursor on the middle of the window, on the last line of the window, and on the first line of the window.

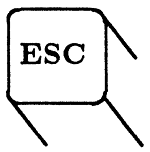
2-7. Search for the number 8. Find the next occurrence of number 8. Find 48.

CREATING TEXT

There are three basic commands for creating text:

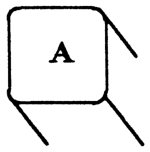
- Append command <**a**>,
- Insert command <**i**>, and
- Open command that creates text on a new line <**o**>.

After you finish creating text with any one of these commands, you can return to the command mode of **vi** with the <**ESC**> command.



The ESC key ends the text input mode.

Append Text



Append text.

<**a**>	Create text to the right of the cursor, or after the cursor.
<**A**>	Append text at the end of the current line.

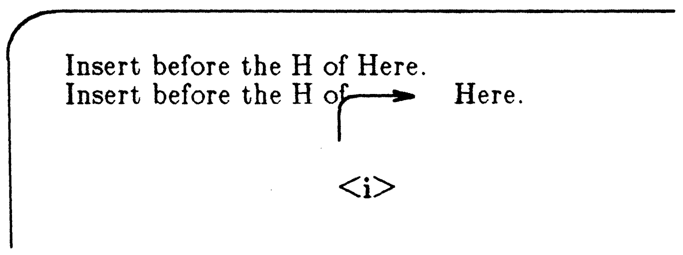
You have already experimented with the <**a**> command in the section on *Getting Started*. Make a new file named *junk2*. Append some text using the <**a**> command. Escape or return to the command mode of **vi** by pressing the ESC key. Then, compare the <**a**> command with the <**A**> command.

Insert Text

I **Insert text.**

<i>	Insert text to the left of the cursor, or before the cursor.
<I>	Create text at the beginning of the current line before the first character that is not a blank.

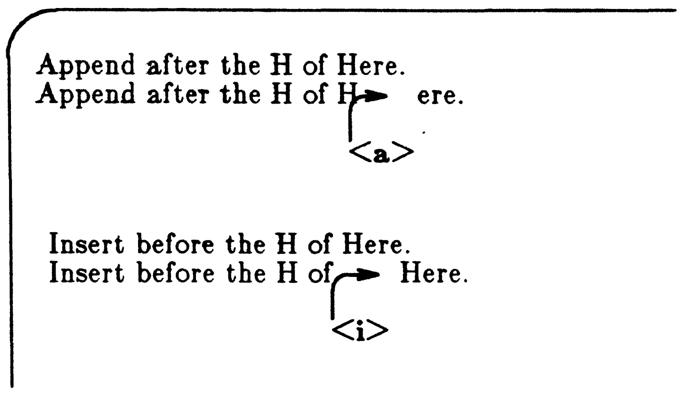
In the example below, the arrow shows where the new text will be created.



```
Insert before the H of Here.
Insert before the H of        Here.
```

Press **ESC**

To end the insert mode and return to the command mode of **vi**, press the "ESC" key. In the next example you can compare the append command with the insert command.


Append after the H of Here.
Append after the H of H → ere.
<a>

Insert before the H of Here.
Insert before the H of → Here.
<i>

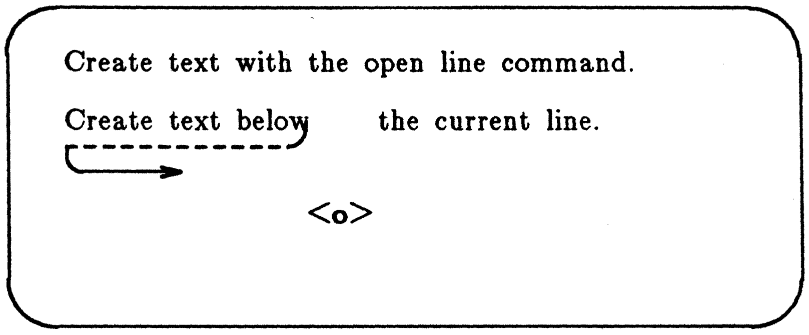
Remember to end the append mode and the insert mode with the <**ESC**> command.

O **Create a new line of text.**

<**o**> The open command <**o**> creates text at the beginning of a new line below the current line. The cursor can be on any character in the current line.

<**O**> To create text at the beginning of a new line above the current line, use the <**O**> command.

In the next screen the <**o**> command opens a new line below the current line and begins creating text at the beginning of the new line.



Create text with the open line command.

Create text below

the current line.

<o>

SUMMARY OF CREATE COMMANDS

<a>	Create text after the cursor.
<A>	Create text at the end of the current line.
<i>	Create text in front of the cursor.
<I>	Create text before the first character on the current line that is not a blank.
<o>	Create text at the beginning of a new line below the current line.
<O>	Create text at the beginning of a new line above the current line.
<ESC>	Return **vi** to the command mode from any of the above text input modes.

EXERCISE 3

3-1. Create a test file *exer3*.

3-2. Insert the following four lines of text.

```
Append text
Insert text
a computer's
job is boring.
```

3-3. Create a line of text

    financial statement and

above the last line.

3-4. Create a line of text

    Delete text

above the third line using an insert command.

3-5. Create a line of text

    byte of the budget

below the current line.

3-6. Using an append command create a line of text

    But, it is an exciting machine.

below the last line.

3-7. Move to the first line and append "some" before "text".

Now, practice each of the six commands for creating text until you are familiar with using them.

3-8. Leave **vi** and go on to the next section to find out how to delete any mistakes you made in creating text.

DELETING TEXT

You can delete text from the text input mode or the command mode of **vi**. In addition, you can undo the effect of your most recent command that changed the buffer.

Delete Commands in the Text Input Mode

To delete text in the text input mode, you will use **<BS>**.

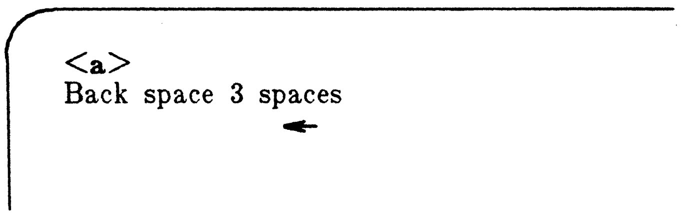
<BS>	Delete the current character, the character indicated by the cursor.

BACK SPACE **Delete a character in the create mode of vi.**

The BACK SPACE key <BS> backs up the cursor in the create mode and deletes each character that the cursor backs across. However, the deleted characters are not erased from the screen until you type over them, or use <ESC> and return to the command mode of **vi**.

In the next examples, the arrows show the movement of the cursor.

Press BACK SPACE **three times.**



```
<a>
Back space 3 spaces
              ←
```

Press **ESC**

```
<a>
Back space 3 spa
               ←
```

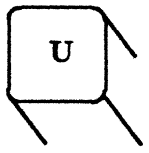
Notice that the characters do not erase from the screen until you press the ESC key.

There are two other commands that delete text in the text input mode. Although you may not use them often, you want to be aware that they are commands in the text input mode and need a special command to type them into your text, see the section on special commands.

<^**w**>	Delete the current word, or a specified portion of the word from the cursor to the end of the word.
<@>	Delete all of the portion of the line that is currently being created.

Undo the Last Command

Before you experiment with the commands that can delete a good portion of your text, you will want to try out the "undo" command, which will undo the last command.



Undo the last command.

<u>	Undo the last command.
<U>	Erase the last change on the current line.

If you deleted a line, **<u>** will bring it back on the screen. If you hit the wrong command, **<u>** will undo that command.

If you press the "u" key twice, it will undo the "undo". That is, if you delete a line, the first **<u>** will restore the line. If you press **<u>** again, it will delete the line again.

Delete Commands in the Command Mode

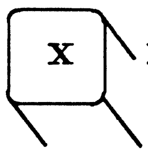
You know that you can precede a number before the command. Many of the commands in **vi**, such as the delete and change commands, allow an argument after the command. The argument can specify a text object such as a word, or a line, or a sentence, or a paragraph. The general form of a **vi** command is:

[number]command[argument]

The brackets around objects in the general form of the command line denote optional parts of the command. They are not part of the command line.

You will see many examples of this form for the delete and change commands.

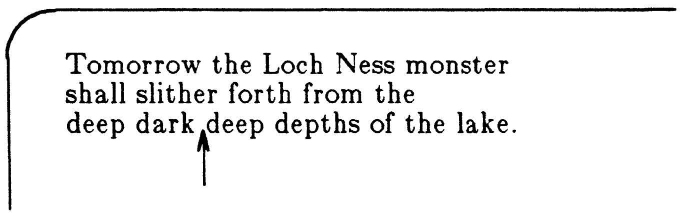
All of the delete commands in the command mode of **vi** immediately remove the deleted text from the screen and redraw that part of the screen.



Delete a character.

<x>	Delete one character.
<nx>	Delete "n" characters, where n is the number of characters you want to delete.

You used <x> in the *Getting Started* section of this chapter. Now try preceding <x> with the number of characters you want to delete.

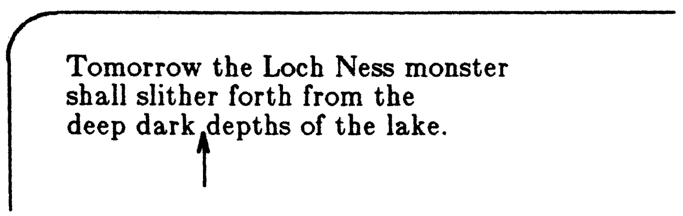


```
Tomorrow the Loch Ness monster
shall slither forth from the
deep dark deep depths of the lake.
```

Put the cursor on the first letter you want to delete, in this example the "d" of the second "deep".

Type in: **5x**

The screen will delete "deep", plus the extra space, and readjust the text on the screen so that it will now read:



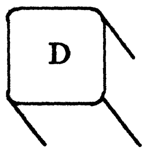
Tomorrow the Loch Ness monster
shall slither forth from the
deep dark depths of the lake.

You can also use the delete word command, which is discussed next.

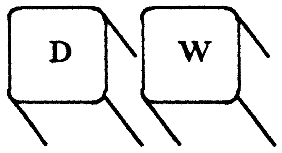
Delete Text Objects

The delete command follows the general form of a **vi** command.

[number]d[text object]

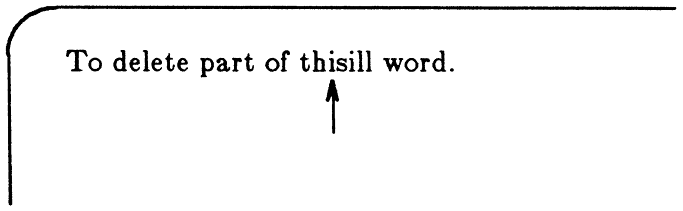


Delete a word, a line, a sentence, or a paragraph.



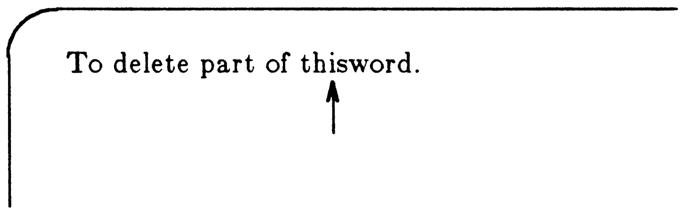
Delete a word.

You can delete all of a word or part of a word with <**dw**> by moving the cursor to the first character you want deleted. Pressing <**dw**> deletes that character and all characters up to and including the next space or punctuation character.



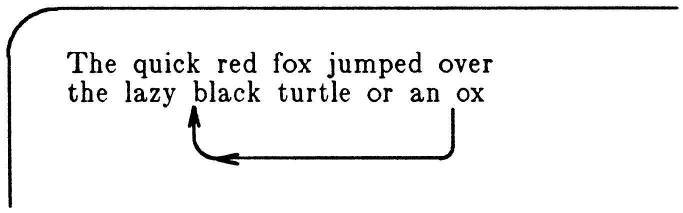
```
To delete part of thisill word.
```

Type in: **dw**



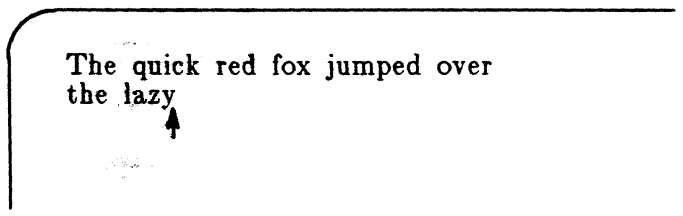
```
To delete part of thisword.
```

You can delete one word with <**dw**> or several words by prefixing the "dw" with a number. The cursor must be on the first character of the first word to be deleted. To delete five words, you would type in **5dw**. An example of how to do this follows.



```
The quick red fox jumped over
the lazy black turtle or an ox
```

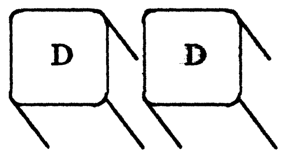
Type in: **5dw**



Try typing in the arguments for other text objects that you learned in the section on positioning the cursor.

Type in: **d(** or **d}**

Observe what happens to your file. Remember, you can restore the text that you just deleted with <**u**>.



<dd> Delete a line of text.

To delete a line, press the "d" key twice. You do not need to worry about deleting text if you press the "d" key once. Nothing will happen, unless you press the space bar. The <**d space bar**> acts like the <**x**> command and deletes one character. If you accidentally press "d" key in the command mode, press the ESC key. The ESC key will cancel the previous typed command.

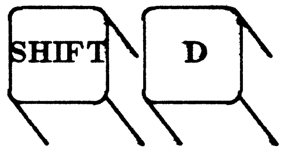
Try to delete ten lines.

Type in: **10dd**

The lines will be deleted from the screen. If some of the lines are below the current window, **vi** will display a notice on the bottom of the screen:

10 lines deleted

If there are not ten lines below the current line in the file, a bell will sound and no lines will be deleted.



Delete the line from the cursor to the end of the line.

If you are erasing the end of a line, use the <D> command. Put the cursor on the first character to be deleted, hold down the SHIFT key while you press the "d" key.

Type in: **D**

The <**D**> command will not allow you to specify more than the current line. You cannot type in "3D". However, you could type in <**3d$**>. Remember the general form of a **vi** command? The $ refers to the end of the line in **vi**.

SUMMARY OF DELETE COMMANDS

For the CREATE Mode:

<BS>	Delete the current character.
<^h>	Delete the current character.
<^W>	Delete the current word.
<@>	Delete the current line of new text, or delete all new text on the current line.

For the COMMAND Mode:

<u>	Undo the last command.
<U>	Erase the last change on the current line.
<x>	Delete the current character.
<ndx>	Delete "n" number of text objects "x".
<dw>	Delete the word at cursor through the next space or to the next punctuation mark.
<dd>	Delete the current line.
<D>	Delete the line at the cursor to the end of the line.
<d)>	Delete the current sentence.
<d}>	Delete the current paragraph.

EXERCISE 4

4-1. Create a file *exer4* containing the following four lines:

```
When in the course of human events there are many
repetitive, boring chores, then one ought to get a
robot to perform those chores.
```

4-2. Move the cursor to line 2 and append to the end of that line:

```
tedious and unsavory.
```

Delete "unsavory" while in the append mode.

Delete "boring" in the command mode.

What is another way you could have deleted "boring"?

4-3. Insert at the beginning of line 4:

congenial and computerized.

Delete the line.

How could you delete the line and leave it blank?

Delete all the lines with one command.

4-4. Leave the screen editor and remove the empty file from your directory.

CHANGING TEXT

Instead of deleting text using a delete command and then creating text with a text input command, the three basic commands, **<r>**, **<s>**, and **<c>** both erase the text and then create new text.

Replacing Text

$\boxed{R}$ **Replace one character that is typed over.**

<r>	Replace the current character, the character pointed to by the cursor. This is not a text input mode. It does not need to be ended by **<ESC>**.
<nr>	Replace "n" characters with the same letter. This command automatically terminates after "nth" character is replaced. It does not need the **<ESC>**.
<R>	Replace only those characters typed over until the **<ESC>** command is given. If the end of the line is reached, this command will then begin appending new text.

The <**r**> command will replace the current character with the next character that is typed in. For example, in the sentence below you want to change "acts" to "ants".

The circus has many acts.

Place the cursor under the "c" of "acts".

Type in: **rn**

The sentence becomes:

The circus has many ants.

To change "many" to "6666", place the cursor under the "m" of "many".

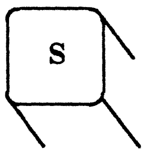
Type in: **4r6**

The <**r**> command changes the four letters of "many" to 6s.

The circus has 6666 ants.

Substituting Text

The substitute command replaces characters, but then allows you to continue to create text from that point until you press <**ESC**>.



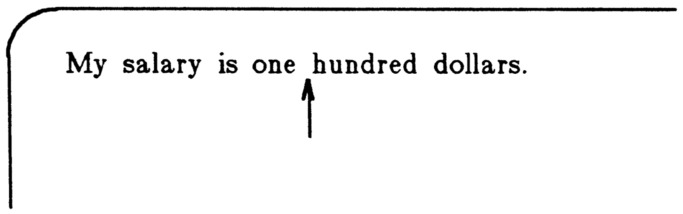
Substitute for a character of text.

<**s**>	Delete the character the cursor is on and append text. End the text input mode with the ESC key.
<**ns**>	Delete "n" characters and append text. End the text input mode with <**ESC**>.
<**S**>	Replace all the characters in the line.

The <**s**> command indicates the last character in the substitution with a $. The characters are not erased from the screen until you type over them, or leave the text input mode with the <**ESC**> command.

Notice that you cannot use an argument with either <**r**> or <**s**>. Did you try?

Suppose you want to substitute "million" for "hundred" in the following example.

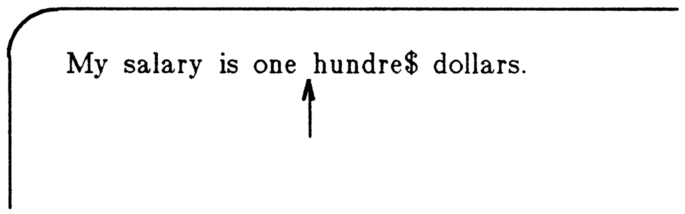


```
My salary is one hundred dollars.
```

Put the cursor under the h of hundred.

Then type in: **7s**

Notice where the $ is placed.



```
My salary is one hundre$ dollars.
```

Now type in: **million**

Press the ESC key, and you will owe the Internal Revenue Service $500,000.

Changing Text

The substitute command replaces characters. The change command replaces text objects, and then continues to append text from that point until you press **<ESC>**. To end the change command and return to the command mode in **vi**, you must press the ESC key.

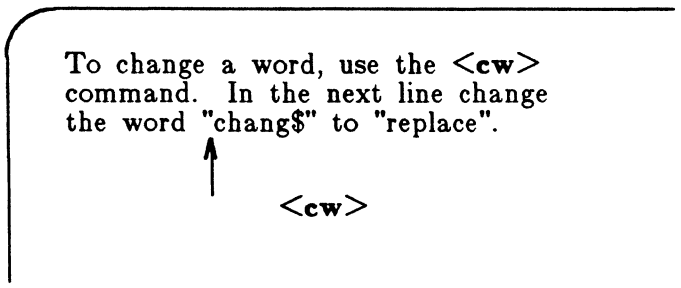
C

Change. Replace a text object with new text.

The change command can take an argument. You can replace a character, word, or an entire line with new text.

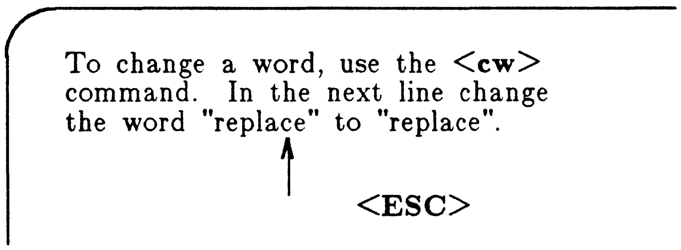
<cw>	Replace a word or the remaining characters in a word with new text. The **vi** editor prints a $ indicating the last character to be changed.
<ncw>	Replace "n" number of words with new text.
<cc>	Replace all the characters in the line.
<ncc>	Replace all the characters in the current line and up to "n" lines of text.
<ncx>	Replace "n" number of text objects "x", such as sentences) and paragraphs }.
<C>	Replace the remaining characters in the line, from the cursor to the end of the line.
<nC>	Replace the remaining characters from the cursor in the current line and replace all the lines under the current line up to "n" lines.

For the **<cw>** command and the <C>, a $ will indicate the last letter that will be replaced. The characters will remain on the screen until you have pressed the ESC key. When used to change one or more lines of text, the change command simply deletes the lines that are to be replaced, and then places you in the text input mode of **vi**.



```
To change a word, use the <cw>
command.  In the next line change
the word "chang$" to "replace".
```

In the example, notice that "replace" has more letters then "change". Once you have executed the change command you are in the text input mode of **vi** and you can add as much text as you want, until you press <ESC>.



```
To change a word, use the <cw>
command.  In the next line change
the word "replace" to "replace".
```

Try the other change commands. Watch the screen. When you use <**C**> the $ will appear at the end of the line. Try using other arguments.

Type in: **c{**

Since you know the undo command, do not hesitate to experiment with different arguments, or preceding the command with a number. You must press <**ESC**> before you can use <**u**> since <**c**> places you in a text input mode.

Compare <**S**> to <**cc**>. The results should be the same for both commands.

SUMMARY OF CHANGE COMMANDS

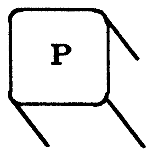
<r>	Replace only the current character.
<R>	Replace only those characters typed over with new characters until the <ESC> command is given.
<s>	Delete the character the cursor is on and append text. End the append mode with the ESC key.
<S>	Replace all the characters in the line.
<cw>	Replace a word or the remaining characters in a word with new text.
<cc>	Replace all the characters in the line.
<ncx>	Replace "n" number of text objects "x", such as sentences) and paragraphs }.
<C>	Replace the remaining characters in the line, from the cursor to the end of the line.

CUTTING AND PASTING TEXT ELECTRONICALLY

There is a set of commands that will cut and paste text in a file. Another set of commands will copy a portion of text and place it in another section of a file.

Moving Text

You can move text from one place to another in the **vi** buffer by deleting the lines and then placing them at the spot in the text that you want them. The last text or lines that were deleted go into a temporary buffer. If you move the cursor to that part of the file where you want the deleted lines to be placed and press the "p" key, the deleted lines will be added below the current line.



The put command <p> puts the last yank or delete in the proper place.

<p>	Place the contents of the temporary buffer after the cursor.
<np>	Place "n" number of copies of the temporary buffer after the cursor.

A partial sentence that was deleted by the **<D>** command can be placed in the middle of another line. Position the cursor in the space between two words, then press "p". The partial line is placed after the cursor.

Characters deleted by **<nx>** also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with **<p>**.

The **<p>** command should be used right after a delete command since the temporary buffer only stores the results of one command at a time. The **<p>** command also places a copy of text after the cursor that had been placed in the temporary buffer by the yank command. Yank **<y>** is discussed next in *Copying Text.*

Fixing Typos

A quick way to fix typos that consist of transposed letters is to combine the **<x>** and the **<p>** commands as **<xp>**. **<x>** deletes the letter. **<p>** places it after next character.

Notice the error in the next line.

A line of tetx

This error can be quickly changed by placing the cursor under the "t" in "tx" and then pressing first "x" and then "p" keys. The result is:

A line of text

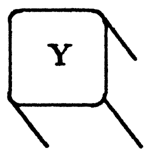
Try it. Make a typing error in your file. Then use **<xp>**.

Copying Text

You can "yank" (copy) a part of the text into a temporary buffer, then move the cursor to that part of the file where you want to place a copy of the text, and place it there. <p> places the text after the current line.

The "yank" command follows the general form of a **vi** command. It allows you to specify the number of text objects that you want copied.

[number]y[text object]



The "yank" command <y> saves a copy of the text object.

<yw>	Yank a copy of a word.
<yy>	Yank a copy of the current line into a temporary buffer to be placed below another line.
<nyy>	Yank "n" lines into a temporary buffer to be placed below the current line. "n" is the number of lines.
<y)>	Yank a copy of a sentence.
<y}>	Yank a copy of the paragraph.
<nyx>	Yank "n" number of text objects "x", such as sentences) and paragraphs }.

Try the following command lines and see what happened to your screen. Of course you can undo the last command.

Type in: **5yw**

Move the cursor to another spot.

Type in: **p**

Try yanking a paragraph <y}> and placing it after the current paragraph, then move to the end of the file **<G>** and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

If you have several sections of text that you wish moved or copied to a different part of the file, it would be tedious to move each portion one at a time. **vi** has named registers, which are electronic storage boxes where you can store the text until you want to place it into a specific spot in the file. These registers are named for each letter of the alphabet, a through z. You can either yank or delete text to one of these registers.

These commands are handy if you have an example that you want to use several times in the text. The example will stay in the specified register until you end the editing session or yank or delete another section of text to that register.

The general form of the command is:

[number"l]command[text object]

The l represents any letter, and is the name of the register. You can precede the command with a number to indicate how many text objects, such as words or lines, that you want to save in the register.

Place the cursor at the beginning of a line.

Type in: **3"ayy**

Now, type in more text. Then, go to the end of the file.

Type in: **"ap**

Did the lines you saved in register "a" appear at the end of the file?

SUMMARY OF CUT AND PASTE COMMANDS

<**p**>	Place the contents of the temporary buffer containing the last delete or yank command into the text after the cursor.
<**yy**>	Yank a line of text and place it into a temporary buffer.
<**ny*x***>	Yank a copy of "n" number of text objects "x" and place them in a temporary buffer.
<**"lyn**>	Place a copy of text object "n" in the register named by a letter "l".
<**"lp**>	Place the contents of register l after the cursor.

EXERCISE 5

5-1. Edit the file *exer2*. Notice that this is the same file you created in Exercise 2.

Go to line 8 and change that line to read "END OF FILE".

5-2. Yank the first eight lines of the file and place them in register "z". Put the contents of register "z" after the last line of the file.

5-3. Go to line 8 and change that line to read "8 is great".

5-4. Go to line 18 and make the same change as you did in 5-3.

5-5. Go to the last line of the file. Substitute "EXERCISE" for "FILE". Replace "OF" with "TO".

SPECIAL COMMANDS

There are some special commands that you will find useful.

<.>	Repeat the last command.
<J>	Join two lines together.
<\> or <^v>	Print out nonprinting character.
<^l>	Clear the screen and redraw it.
<~>	Change lowercase to uppercase and vice versa.

Repeating the Last Command

. **Repeat the last change command.**

You may have already accidentally pressed the "." key, thinking that you were adding a period at the end of your sentence. If you were in the command mode of **vi**, you were unpleasantly surprised by the last text change suddenly appearing on the screen.

The period repeats the last change command. This is a very handy command when it is used with the search command. For example, you forgot to capitalize the "S" in United States. However, you do not want to capitalize the "s" in "chemical states". One way you could correct this problem is search for "states". The first time you found "states" in United states, you would change the "s" to "S". The next occurrence you found, you would simply press the "." key and **vi** would remember to change the "s" to "S".

The <.> will repeat change, or create, or delete, or put commands. Experiment with the commands. Watch the screen to see how the text is affected.

Joining Two Lines

SHIFT J **Join the line below the current line with the current line.**

The <J> command joins lines. Place the cursor on the current line, hold down the SHIFT key and press the "j" key. The line below the current line is joined to the current line at the end of the current line.

Now is the time to join forces.

To join these two lines into one line, place the cursor under any character in the first line.

Type in: **J**

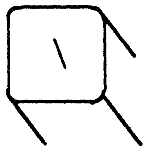
Those two lines become:

Now is the time to join forces.

Notice that **vi** automatically places a space between the last word on the first line and the first word on the second line.

Typing Nonprinting Characters

In the section of this tutorial on deleting in the text input mode, two commands were mentioned that are probably seldom used, but act as commands and will not print out in your text. How do you get characters that are commands in the text input mode to type out in your text? Precede them with a \ .

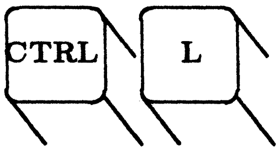


Type in nonprinting characters.

What happens when you want to type in the @ character? Try it. It erased the line you are working on. How do you type in the @ character?

Type in: \@

Clearing and Redrawing the Window



Clear and redraw the current screen.

One of the frustrating things that can happen to you in **vi** is that another user in your ICON/UXV system decides to send you a message using the **write** command. If you have not turned off your messages in the shell, the message will appear right at the spot where you are editing in the current window. After you have read the message, how do you restore the current window? If you are in the text input mode, you must end it with the <ESC> command to get you into the command mode of **vi**. Then, hold down the CTRL key and press the "l" key. **vi** will clear away the garbage, and redraw the window exactly as it was before the message arrived.

Changing Lowercase to Uppercase and Vice Versa

~ **Change uppercase to lowercase, or lowercase to uppercase.**

A quick way to change any lowercase letter to a capital letter or any capital letter to lowercase is the <~> command. To change a to A, or B to b press ~. This command does not allow you type in a number before the command and change several letters with one command.

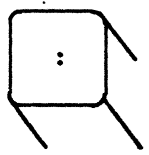
SUMMARY OF SPECIAL COMMANDS

<.>	Repeat the last command.
<J>	Join the line below the current line with the current line.
<\x>	Print the nonprinting character x that does not print out in the text input mode.
<^v>	Print characters that do not normally print out in the text input mode.
<^l>	Clear and redraw the current window.
<~>	Change lowercase to uppercase, or vice versa.

LINE EDITING COMMANDS

The screen editor **vi** also has some line editing capabilities. The line editor associated with **vi** is called **ex**. However, the **ex** commands are very similar to the **ed** commands discussed in *Chapter 5.* If you know the **ed** commands, you may want to experiment on a test file and see how many will work in **vi**.

There are many commands in the **ex** editor that can be called from **vi**. These commands are discussed at length in the *ICON/UXV Editing Guide.* (See *Appendix A.*) Only a few of the most useful commands are discussed here.



Call in the line editor commands.

To call in the line editor commands, type in a ":" from the command mode of **vi**. The cursor will drop down to the bottom of the screen and display the ":". As you try out the line editing commands notice that they print out at the bottom of the editing window.

A powerful and useful command of **ex** is the command that temporarily returns you to the shell. You can return to the shell, perform some shell commands (even edit and write another file in **vi**) and then return to the current window of **vi**.

:sh<CR>	Temporarily return to the shell, leaving the **vi** buffer with the cursor on the current line.
<^d>	After you have executed the shell commands, hold CTRL and press "d". You will return to the exact line and window you were editing before you left **vi**.

Even if you change directories while you are temporarily in the shell and then execute **<^d>**, you will return to the **vi** buffer in the directory where you were editing the file.

Write Text to a New File

What do you do if you want only part of the file in the editing buffer placed in a ICON/UXV file?

Many of the commands in **ex** will accept a line number or a range of line numbers typed in before the command **w**. Try to write the third line of the buffer to a file named *three.*

Type in: **:3w three<CR>**

Notice the system response.

"three" [New file] 1 line, 20 characters

The "**.**" is the special character that indicates the number of the current line.

Type in: **:.w junk<CR>**

A new file called *junk* will be created containing only the current line in the **vi** buffer.

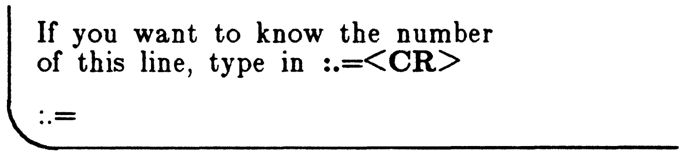
You can also specify the range of lines. To write lines 23 through 37 to a file, type in:
23,37w newfile<CR>

Finding the Line Number

If you want to specify a range of lines, you can find out the line number of that line by moving the cursor to that line.

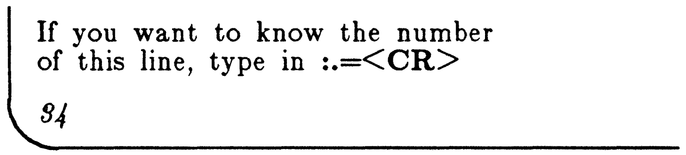
Type in: :.=<CR>

The editor will come back with the response that is the number of that line.



```
If you want to know the number
of this line, type in :.=<CR>

:.=
```

As soon as you press RETURN, the bottom line will clear and give you the number of the line in the buffer.



```
If you want to know the number
of this line, type in :.=<CR>

34
```

You can move the cursor to any line in the buffer by typing in a ":" and the line number.

:n<CR> Go to the "nth" line of the buffer.

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines from the current line to the end of the buffer is to use the line editor command to delete lines.

Type in: **:.,$d<CR>**

The "." is the current line, and the last line is $.

Adding a File to the Buffer

If you have a file with some data or text in it that you would like to add below a specific line in the editing buffer, you can do so with the **:r** command. To read in the file *data* place the cursor on the line above the desired insertion.

Type in: **:r data<CR>**

You may also specify the line number instead of moving the cursor. Insert the file *data* below line 56 of the buffer.

Type in: **:56r data<CR>**

Do not be afraid to experiment, <**u**> will undo the **ex** commands too.

Making Global Changes

One of the most powerful commands in **ex** is the global command. The global command is given here to help those users who are familiar with the line editor. Even if you are not familiar with a line editor, you may want to try the command on a test file.

If you had typed in several pages of text about the DNA molecule, calling its structure a "helix", you would have to change each occurrence of the word "helix" to "double helix". This could be a long involved process searching for each one and probably using the "." command of **vi** to repeat the change. If you are sure you want every "helix" changed, you can use the global command of **ex**. You need to understand a series of commands to do this. Let's take one at a time.

:g/characters<CR>

Search for these exact characters.

Type in: **:g/helix<CR>**

The line editor does a global search for the first instance of the characters "helix" on a line.

:s/text/new words/<CR>

This is the substitute command. Instead of writing over the word **text**, as the screen editor would have done, the line editor searches for the first instance of the characters **text** on the current line, and changes them to **new words**. You must tell **ex** what word you are looking for and it must appear between the first two delimiters, /. It will then replace only those exact characters with the exact characters, **new words**, between the last two delimiters.

:s/text/new words/g<CR>

By adding a "g" at the end of the last delimiter of this command line, **ex** will change every occurrence on the current line.

:g/helix/s//double helix/g<CR>

This command line searches for the word **helix**. Each time **helix** is found, the substitute command substitutes **double helix** for every instance of **helix** on that line. The delimiters after the **s** do not need to have **helix** typed in again. The command remembers the word from the delimiters after the global command **g**.

This is a very powerful command. If it is confusing to you, but you still want to add it to your **vi** command knowledge, read *Chapter 5* on the line editor **ed** for a more detailed explanation of the global and substitution commands.

SUMMARY OF LINE EDITOR COMMANDS

:	Indicates that the next commands are line editor commands.
:sh<CR>	Temporarily return to the shell to perform some shell commands.
<^d>	Escape the temporary shell and return to edit the current window of **vi**.
:n<CR>	Go to the "nth" line of the buffer.
:x,z**w data<CR>**	Write lines from the number "x" through the number "z" into a new file called *data*.
:$<CR>	Go to the last line of the buffer.
:.,$d<CR>	Delete all the lines in the buffer from the current line to the last line.
:r shell.file<CR>	Insert the contents of *shell.file* under the current line of the buffer.
:s/text/new words/<CR>	Replace the first instance of the characters **text** on the current line with **new words**.
:s/text/new words/g<CR>	Replace every occurrence of **text** on the current line with **new word**.
:g/text/s//new word/g<CR>	Change every occurrence of **text** to **new word**.

QUITTING VI

There are six basic command sequences to quit the **vi** editor.

<ZZ>	Write the contents of the **vi** buffer to the ICON/UXV file currently being edited and quit **vi**.
:wq<CR>	Write the contents of the **vi** buffer to the ICON/UXV file currently being edited and quit **vi**.

:w filename<CR> **:q<CR>**	Write the temporary buffer to a new file named *filename* and quit **vi**.
:w! filename<CR> **:q<CR>**	Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**.
:q!<CR>	Quit **vi** without writing to the shell file.
:q<CR>	Quit **vi** without writing the buffer to a ICON/UXV file. This command, without the write command **w**, can only be used in special cases, such as the **view** command discussed in the next section, or if the buffer has not been changed.

The commands that are preceded by a ":" are line editor commands.

The **<ZZ>** command and **:wq** command sequence both write the buffer to a ICON/UXV file, then quit **vi**, and return you to the shell command level. You have tried the **<ZZ>** command, now try to exit **vi** with **:wq**.

Type in: **:wq<CR>**

The system response is the same as it is for the **<ZZ>** command. It gives you the name of the file, and the number of lines and characters in the file.

vi remembers the file name of the file currently being edited, so you do not have to reiterate the file name when you want to write the buffer of the editor back into that file. What do you do if you want to give the file a different name?

If you want to write to a file called *junk*:

Type in: **:w junk<CR>**

After you write to a new file, you can leave **vi** by just typing in the **:q**.

Type in: **:q<CR>**

If you try to write to a file called *letter* that already exists in the shell, you will receive a warning:

"letter" File exists - use "w! letter" to overwrite

Type in: **:w! letter<CR>**

You will erase the current file called *letter* and overwrite it with the new file.

If you began editing a file called *memo*, made some changes to the file, and then decided you didn't want to make the changes, or you accidentally pressed a key that gave **vi** a command you could not undo, you can leave **vi** without writing to the file.

Type in: **:q!<CR>**

SUMMARY OF QUIT COMMANDS

<ZZ>	Write the file and quit **vi**.
:wq<CR>	Write the file and quit **vi**.
:w filename<CR> **:q<CR>**	Write the editing buffer to a new file named *filename* and quits **vi**.

(Continued on next page)

:w! filename<CR> **:q<CR>**	Overwrite an existing file called *filename* with the contents of the editing buffer and quits **vi**.
:q!<CR>	Quit **vi** without writing to the buffer.
:q<CR>	Quit **vi** without writing the buffer to a ICON/UXV file.

SPECIAL OPTIONS FOR vi

The **vi** command has some special options. It allows you to:

- Recover a file lost by an interrupt to the ICON/UXV system,
- Place several files in the editing buffer and edit each in sequence, and
- View the file with the **vi** cursor positioning commands.

Recovering a File Lost by an Interrupt

There are times when an interrupt or a disconnect will cause the system to exit the **vi** command without writing the temporary buffer to the ICON/UXV file. Or, you may become confused or have a problem with the **vi** editor that you cannot solve. If that happens, one solution is simply to hang up, or disconnect from the ICON/UXV system. In both of these cases, the ICON/UXV system will store a copy of the buffer for you. When you log back into the ICON/UXV system you will want to restore the file with the **−r** option for the **vi** command:

Type in: **vi −r filename<CR>**

The changes you made to the file *filename*, before the interrupt occurred, are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the file name and write to that file.

Editing Multiple Files

If you wish to edit more than one file in the same editing session, type in the **vi** command followed by each file name.

Type in: **vi file1 file2<CR>**

vi will respond by telling you how many files you are going to edit.

2 files to edit

After you have edited the first file, *file1*, you need to write the changes to the shell file.

Type in: **:w<CR>**

The system response to the **:w <CR>** command will be a message at the bottom of the screen giving the name of the file, and how many lines and characters are in that edited file. Then you must ask for the next file in the editing buffer with the **:n** command.

Type in: **:n<CR>**

The system response to the command **:n<CR>** is a notice at the bottom of the screen with the name of the next file to be edited and the character and line count of that file.

Pick two of the files in your current directory and enter **vi** to place the two files in the editing buffer at the same time. Notice the system responses to the commands at the bottom of the screen.

SUMMARY OF SPECIAL OPTIONS FOR vi

vi file1 file2 file3<CR>	Enter three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.
:w<CR> **:n<CR>**	Write the current file and call the next file in the buffer.
vi –r file1<CR>	Restore the changes made to the file *file1*.

EXERCISE 6

6-1. Try to restore a file lost by an interrupt.

Enter **vi**, create some text in a file called *exer6*.

Turn off your terminal without writing to a file or leaving **vi**.

Log back in to your terminal.

Try to get back into **vi** and edit the *exer6* file.

6-2. Place *exer1* and *exer2* in the **vi** buffer to be edited.

Write *exer1* and call in the next file in the buffer, *exer2*.

Write *exer2* to a file called *junk*.

Quit **vi**.

6-3. Try out the command:

vi exer*<CR>

What happens? To quit **vi**:

Type in: **ZZ ZZ**

6-4. Look at *exer4* in read only mode.

Scroll forward.

Scroll down.

Scroll backward.

Scroll up.

Quit and return to the shell.

CHANGING YOUR ENVIRONMENT

If you are going to edit with **vi** you will want to change your login environment so that you do not have to reconfigure your terminal each time you login. Your login environment is controlled by a file in your login directory called the *.profile*. The *shell tutorial in Chapter 7.*

You are about to edit your *.profile* that sets up your environment each time you login. If you are concerned that you might cause a problem with your *.profile* in the editing process, you may want to keep a backup copy of your original *.profile* for safekeeping.

From your login directory, type in:

cp .profile safe.profile<CR>

Now that you have a copy of your *.profile* in a safe place, *safe.profile*, you can edit your *.profile* just like any other file in **vi**.

Type in: **vi .profile<CR>**

Go to the last line of the file, ignoring all the lines currently in the file.

Type in: **G**

You are going to add two lines to the bottom of the file, the same terminal configuration you typed in at the beginning of your login session so that you could enter **vi**.

Type in: **<o>**

Now you are ready to append text to the end of the file.

Type in:
```
TERM=code<CR>
export TERM<CR>
```

Remember "code" is the special code characters for your type of terminal.

Write and quit **vi**. Now, the next time that you log into the ICON/UXV system **TERM** is automatically set and you can immediately begin editing with **vi**.

Setting the Automatic Carriage Return

If you want an automatic carriage return, create a new file *.exrc*. The *.exrc* file controls the editing environment for **vi**. There are several options you can set in this file. If you want to know more about *.exrc*, read the *Editing Guide*. (See *Appendix A.*)

Type in: **vi .exrc<CR>**

Add one line to this file.

Type in: **wm=n<CR>**

"n" is the number of characters from the right side of the screen where the carriage return will occur. If you want a carriage return at 20 characters from the right edge of the screen,

Type in: **wm=20<CR>**

Write and quit that file. The next time you login this file will give you an automatic carriage return.

You can check on these settings, the terminal setting and the wrapmargin (automatic carriage return) when you are in **vi**.

Type in: **:set<CR>**

vi will tell you the terminal type and the wrapmargin. You can also use the **:set** command to create or change the wrapmargin. Try experimenting with it.

Now you know the basics of **vi**! Experiment with the commands, find the ones that work best for you.

ANSWERS TO EXERCISES

There is often more than one way to perform a task in **vi**. If the way you tried worked, then your answer is correct. Below are suggestions for performing the task given in the exercise.

Exercise 1

1-1. Look up your terminal code with the following command. Type in:

```
grep "your type of terminal" /etc/termcap<CR>
```

The first two letters of of the system response are your terminal code. Type in:

```
TERM=code<CR>
export TERM<CR>
```

1-2. Type in:

```
vi exer1<CR>
<a>
This is an exercise!<CR>
Up, down<CR>
left, right,<CR>
build your terminal's<CR>
muscles bit by bit.<ESC>
```

1-3. Use the **<k>** and the **<h>** commands.

1-4. Use **<x>**.

1-5. Use the **<j>** and **<l>** commands.

1-6. Type in:

<a> <CR>
and byte by byte<ESC>

Use **<j>** and **<l>** to move to the last line and character of the file. Use **<a>** to add text. **<CR>** will create the new line. **<ESC>** will end the create mode.

1-7. Type in:

ZZ

1-8. Type in:

vi exer1<CR>

System response: *"exer1" 6 lines, 100 characters*

Exercise 2

2-1. Type in:

```
vi exer2<CR>
<a>1<CR>
2<CR>
3<CR>
 .
 .
 .
48<CR>
49<CR>
50<ESC>
```

2-2. Type in:

```
<^f>
<^b>
<^u>
<^d>
```

Notice the line numbers as the screen changes.

2-3. Type in:

```
<G>
<o>
123456789 123456789<ESC>
```

2-4. **$** = end of line
0 = first character in the line

2-5. Type in:

```
<^>
<w>
<b>
<e>
```

2-6. Type in:

```
<1G>
<M>
<L>
<H>
```

2-7. Type in:

```
/8
<n>
/48
```

Exercise 3

3-1. Type in:

```
vi exer3<CR>
```

3-2. Type in:

```
<a> Append text <CR>
Insert text<CR>
a computer's <CR>
job is boring.<ESC>
```

3-3. Type in:

```
<O>
financial statement and<ESC>
```

3-4. Type in:

```
<3G>
<i>Delete text<CR><ESC>
```

The text in your file now reads:

```
Append text
Insert text
Delete text
a computer's
financial statement and
job is boring.
```

3-5. The current line is "a computer's". To create a line of text below that line use the <o> command.

3-6. The current line is "byte of the budget".
<G> will put you on the bottom line.
<A> will begin appending at the end of the line.
<CR> will create the new line.
Then, type in the text "But, it is an exciting machine."
<ESC> ends the append mode.

3-7. Type in:

```
<1G>
/text
<i>some<space bar><ESC>
```

3-9. <ZZ> will write the buffer to *exer3* and put you in the command mode of the shell.

Exercise 4

4-1. Type in:

```
vi exer4<CR>
<a> When in the course of human events<CR>
there are many repetitive, boring<CR>
chores, then one ought to get a<CR>
```

robot to perform those chores.**<ESC>**

4-2. Type in:

<2G>
<A> tedious and unsavory**<CR>**
<8BS>
<ESC>

Press **<h>** until you get to the "b" of "boring" then press **<dw>**. Or, you could have used **<6x>**.

4-3. You are at the second line. Type in:

<2j>
<I> congenial and computerized**<ESC>**
<dd>

To delete the line and leave it blank, type in:

<0> (zero to place you at the beginning of the line)
<D>

<H>
<3dd>

4-4. Write and quit **vi**.

<ZZ>

Remove the file.

rm exer4<CR>

Exercise 5

5-1. Type in:

vi exer2<CR>
<8G>
<cc> END OF FILE <ESC>

5-2. Type in:

<1G>
<8"zyy>
<G>
<"zp>

5-3. Type in:

<8G>
<cc> 8 is great<ESC>

5-4. Type in:

```
<18G>
<.>
```

5-5. Type in:

```
</FI>
<cw> EXERCISE<ESC>

<?OF>
<R>TO<ESC>
```

Exercise 6

6-1. Type in:

```
vi exer6<CR>
<a> (append several lines of text)
<ESC>
```

Turn off the terminal.

Turn on the terminal.
Log into the ICON/UXV system. Type in:

```
vi −r exer6<CR>
:wq<CR>
```

6-2. Type in:

```
vi exer1 exer2<CR>
:w<CR>
:n<CR>

:w junk<CR>
ZZ
```

6-3. Type in:

```
vi exer*<CR>
```

(Response)
8 files to edit (**vi** calls in all files with
names that begin with *exer*.)

```
ZZ
ZZ
```

6-4. Type in:

```
view exer4<CR>
<^f>
<^d>
<^b>
<^u>
```

Chapter 7

SHELL TUTORIAL

Chapter 7

SHELL TUTORIAL

MAKING LIFE EASIER IN THE SHELL

You have used the shell to interact with the ICON/UXV system by typing in commands that give you information, such as **who**, or commands that perform a task, such as **sort**. This chapter introduces some methods and commands that will help expedite the day-to-day tasks that you perform in the shell.

The first part of the tutorial, *Shell Command Language*, introduces some basic shortcuts and commands to help you perform tasks in the ICON/UXV system quickly and easily. The second part of the tutorial, *Shell Programming*, shows you how to put these tasks into a file and call on the shell to execute the commands in the file while you go get a cup of coffee. The following basics are covered:

- How to use some special characters in the shell,
- How to redirect input and output,
- How to execute and terminate processes,
- How to create and execute a simple shell program,
- How to use variables in a shell program,
- How to use shell programming constructs for looping, conditional execution, and unconditional execution,
- How to locate problems and debug a shell program, and
- How to modify your login environment by editing the file called *.profile*.

If, after you have read this tutorial, you want to learn more advanced concepts in shell programming, read *Part 3, Shell Commands and Programming* of this manual. (See *Appendix A*.)

HOW TO READ THIS TUTORIAL

Log into your ICON/UXV system and try the examples as you read the text. Experiment with the concepts and perhaps combine them into a shell program. Often, there is more then one correct way to write a shell program. You may discover a different method. If your shell program works, if it performs the task, then it is a correct method.

Here is a quick review of the text conventions mentioned in *Chapter 2* that are used throughout this book.

bold command(Type in the command line exactly as shown.)

italic response(The system's response to a command.)

< >	(Commands that are typed in, but not displayed on your terminal, are enclosed in < >.)
^g	(A control character, hold down the control key CTRL while your press "g".)

A display screen like the one above is used to illustrate the commands and the text of the shell programs. You may not be working on a terminal with a screen. This will not affect the shell tasks that you perform or shell programs that you create. The lines that you type in and the system responses should be the same.

SHELL COMMAND LANGUAGE

Special Characters in the Shell

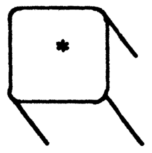
The shell language has special characters that give you some shortcuts for performing tasks in the shell. These special characters are listed below and are discussed in this section of the tutorial.

* ? [ ]	These are metacharacters. A metacharacter is a character that has a special meaning in shell command language. These metacharacters give you shortcuts for file names.
&	This character places commands in the background mode. While the shell is performing the commands in the background, your terminal is free for you to work on other tasks.
;	This character allows you to type in several commands on one line. Each command must be followed by a ; . When you type in the **<CR>**, each command will execute sequentially from the beginning of the line to the end of the line.
\	This character allows you to turn off the meaning of special characters such as *, ?, [ ], & and ; .
" ... " ´ ... ´	Both double and single quotes turn off the delimiting meaning of the space, and the special meaning of special characters. However, double quotes will allow the characters $ and \ to retain their special meaning. (The $ and \ are discussed later in this chapter and are important for shell programs.

Metacharacters

The meaning of the metacharacters is similar to saying "etc. etc. etc.", "all of the above", or "one of these". Using metacharacters for all or part of a file name is called file name generation. It is a quick and easy way to refer to file names.

Metacharacter That Matches All Characters



This metacharacter matches "all", any string of characters, including no characters at all.

The * alone refers to all the file names in the current directory, the directory you are in now. To see the effect of the *, try the next command.

Type in: **echo *<CR>**

The **echo** command displays its arguments on your terminal. The system response to **echo *** should have been a listing of all file names in the current directory. However, unlike **ls**, the file names were displayed in horizontal lines instead of a vertical listing.

Since you may not have used the **echo** command before, here is a brief recap of the command.

Command Recap

echo - write any arguments to the output

command	*options*	*arguments*
echo	none	**any character**

Description: **echo** writes arguments, which are separated by blanks and ended with <CR>, to the output.

Remarks: In shell programming, **echo** will be used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All of these uses will be discussed later in this chapter.

Problem:
Be very careful with * because it is a powerful character. If you type in **rm** * you will erase all the files in your current directory.

The * metacharacter is also used to expand file names in the current directory. If you have written several reports and have named them:

```
report
report1
report1a
report1b.01
report25
report316
```

then

```
report*
```

refers to all six reports in the current directory. If you want to find out how many reports you have written, you could use the **ls** command to list all the reports that begin with the letters **report**.

```
$ ls report*<CR>
report
report1
report1a
report1b.01
report25
report316
$
```

The * refers to any characters after the letters **report**, including no letters at all. Notice that * calls the files in numerical and alphabetical order. A quick and easy way to print out all of those reports in order is:

Type in: **pr report*<CR>**

Choose a character that your file names have in common, such as an **a**, and list all those files in the current directory.

Type in: **ls *a*<CR>**

The * can be placed anywhere in the file name.

Type in: **ls F*E<CR>**

This command line would list all of the following files in order:

```
F123E
FATE
FE
Fig3.4E
```

Metacharacter That Matches One Character

?

This metacharacter matches any single character.

The **?** metacharacter replaces any one character of a file name. If you have created text for several chapters of a book, but you only want to list the chapters you have written through **chapter9**, you would use the **?** .

```
$ ls chapter?<CR>
chapter1
chapter2
chapter5
chapter9
$
```

Although **?** matches any one character, you can use it more than once in a file name. To list the rest of the chapters up through **chapter99**, type in:

ls chapter??<CR>

Of course, if you want to list all the chapters in the current directory you would use **chapter***.

Problem:
Sometimes when you **mv** or **cp** a file you accidentally press a character that does not print out on your terminal as part of the file name when you do an **ls**. If you try to **cat** that file, you get an error message. The * and **?** are very useful in calling up the file and moving it to the correct name. Try the following example.

1. Make a very short file called *trial*.

2. Type in: **mv trial trial<^g>1<CR>**

   Remember to type in <^**g**> you hold down the CTRL key and press the "g" key.

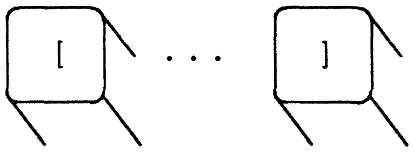
3. Type in: **ls trial1<CR>**

```
$ ls triall<CR>
triall not found
$
```

4. Type in: **ls trial?1<CR>**

```
$ ls trial?1<CR>
triall
$ mv trial?1 triall<CR>
$ ls triall<CR>
triall
$
```

Metacharacters That Match One of a Specific Range of Characters



The shell matches one of the specified characters or range of characters within the brackets.

Characters enclosed in [ ] act as a specialized form of the ? . The shell will match only one of the characters enclosed in the brackets in the position specified in the file name. If you use [**crf**] as part of a file name, the shell will look for **c**, or **r**, or **f**.

```
$ ls [crf]at<CR>
cat
fat
rat
$
```

The shell will also look for a range of characters within the brackets. For **chapter[0-5]** the shell looks for the files named *chapter0* through *chapter5*. This is an easy way to print out only certain chapters at one time.

Type in: **pr chapter[2-4]<CR>**

This command will print out the contents of *chapter2*, *chapter3*, and *chapter4* in that order.

The shell will also look for a range of letters. For **[A-Z]**, the shell will look for uppercase letters, or for **[a-z]**, the shell will look for lowercase letters.

Try out each of these metacharacters on the files in your current directory.

Commands in the Background Mode

&

This character, placed at the end of a command line, runs a task in background mode.

Some shell commands take considerable time to execute. It is convenient to let these commands run in background mode to free your terminal so that you can continue to type in other shell tasks. The general format for a command to run in background mode is:

command &<CR>

The **grep** command can perform long searches that may take a lot of time. If you place the **grep** command in a background mode, you can continue doing some other task at your terminal while the search is being done by the shell. In the example below, the background mode is used while all the files in the directory are being searched for the characters **word**. The **&** is the last character after the command.

```
$ grep word * &<CR>
21940
$
```

21940 is the process number. This number is essential if you want to stop the execution of a background command. This will be discussed in *Executing and Terminating Processes.*

In the next section of this tutorial you will see how to redirect the system response of the **grep** command into a file so that it does not display on your terminal and interrupt your current work. Then, you can look at the file when you have finished your task.

Sequential Execution

The shell performs sequential execution of commands typed on one line and separated by a ; .

If you want to type in several commands on one line, you must separate each command with a ; . The general format to place **command1**, **command2**, and **command3** on one command line is the following:

command1; command2; command3<CR>

Sequential execution is very useful if you need to execute several shell commands while you are in the line editor **ed**. (See the section on *Other Useful Commands and Information* in *Chapter 5.*) Try out the ; . Type in several commands separated by a ; . Notice that, after you press <CR>, the system responds to each command in the order that they appear on the command line.

Type in: **cd; pwd; ls; ed trial<CR>**

The shell will execute these commands sequentially:

1. **cd** Change to login directory.
2. **pwd** Print the path of the current directory.
3. **ls** List the files in the current directory.
4. **ed trial** Enter the line editor **ed** and begin editing the file *trial.*

Did you notice the rapid fire response to each of the commands? You may not want these responses to display on your terminal. The section on *Redirecting Output* will show you how to solve this problem.

Turning Off Special Character Meaning

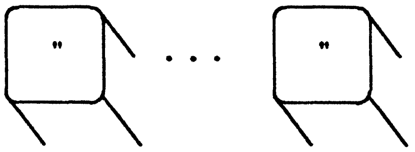
\ **The backslash turns off the special meaning of a metacharacter.**

How do you search for one of the special characters in a file? Type in a backslash just before you type in the metacharacter. The backslash turns off the special meaning of the next character that you type in. Create a file called *trial* that has one line containing the sentence "The all * game". Search for the * character in the file *trial*.

```
$ grep \* trial<CR>
The all * game
$
```

Turning Off Special Characters by Quoting

' . . . ' **All special characters enclosed in single quotes lose their special meaning.**



All special characters except $, \, and lose their special meaning when they are in double quotes.

The special characters in the shell lose their special meaning when they are enclosed by quotes. The single quote turns off the special meaning of any character. The double quote will turn off the special meaning of any character except $ and . The $ and are very important characters in shell programming.

A delimiter separates arguments, telling the shell where one argument ends and a new one starts. The space has a special meaning to the shell because it is used as a delimiter between arguments of a command.

The **banner** command uses spaces to delimit arguments. If you have not used the **banner** command, try it out. The system response is rather surprising.

Type in: **banner happy birthday to you<CR>**

Was each word displayed in large poster sized letters?

Now put quotes around **to you**.

Type in: **banner happy birthday "to you"<CR>**

Notice that **to** and **you** appear on the same poster display line. The space between the **to** and the **you** has lost its special meaning as a delimiter.

Since you may not have used the **banner** command before, the following is a quick recap of that command. You may find that you do not have access to the **banner** command. Not all systems have all the commands referenced in this chapter. If you cannot access a command, check with your system administrator.

Command Recap

banner - make posters

command	*options*	*arguments*
banner	none	**characters**

Description: Displays arguments, up to ten characters on a poster-sized line, in large letters.

Remarks: Later in this chapter you will learn how to redirect the **banner** command into a file to be used as a poster.

If you use single quotes in the argument for the **grep** command, the space loses the meaning of a delimiter. You can search for two words. The line, **The all * game** is in your file *trial.* Look for the two words **The all** in the file *trial.*

```
$ grep 'The all' trial<CR>
The all * game
$
```

Try turning off the special character meaning of the * using single quotes.

```
grep '*' trial<CR>
The all * game
$
```

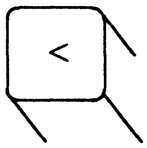
If you want to know more about quoting, read the *ICON/UXV User Reference Manual* pages on the **sh** command.

Redirecting Input and Output

The redirection of input and output are important tools for performing many shell tasks and programs.

Redirecting Input

You can redirect the text of a file to be the input for a command.



This character redirects the contents of a file into a command.

The general format to redirect the contents of a file into a command is shown below.

command < filename<CR>

If you write a report to your boss, you probably do not want to type in the **mail** command and then type in your text. You want to be able to put your report in an editor and correct errors. You want to run the file through the **spell** command to make sure there are no misspelled words. You can **mail** a file containing your report to another login using the redirection symbol. In the example below, a file called *report* is checked for misspelled words and then redirected to be the input to the **mail** command and mailed to login **boss**.

```
$ spell report<CR>
$
$ mail boss < report<CR>
$
```

Since the only response to the **spell** command is the prompt, there are no misspelled words in *report*. The **spell** command is a useful tool that gives you a list of words that are not in a dictionary spelling list. The following is a brief recap of **spell**.

Command Recap

spell - find spelling errors

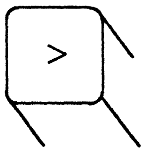
command	*options*	*arguments*
spell	**available***	**filename**

Description:	**spell** collects words from the specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.
Options:	**spell** has several options, including one for checking the British spelling.
Remarks:	The misspelled words can be redirected into a file. See the redirection symbol > discussed next.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Redirecting Output

You can redirect the output of a command to be the contents of a file. When you redirect output into a file, you can either create a new file, append the output to the bottom of a file, or you can erase the contents of an old file and replace it with the redirection output.



This character redirects the output of a command into a file.

The single redirection symbol > will create a new file, or it will erase an old file and replace the contents with new output. The general format to redirect output is shown below.

command > filename<CR>

If you want the **spell** command list of misspelled words placed in a file instead of displayed on your terminal, redirect **spell** into a file. In the example, **spell** searches the file *memo* for misspelled words and places those words in the file *misspell*.

```
$ spell memo > misspell<CR>
$
```

The **sort** command can be redirected into a file. Suppose a file called *list* contains a list of names. In the next example, the output of the **sort** command lists the names alphabetically and redirects the list to a new file *names*.

```
$ sort list > names<CR>
$
```

Problem:
Be careful to choose a new name for the file that will contain the alphabetized list. The shell first cleans out the contents of the file that is going to accept the redirected output, then it sorts the file and places the output in the clean file. If you type in

sort list > list<CR>

the shell will erase *list* and then sort nothing into *list*.

Problem:
If you redirect a command into a file that exists, the shell will erase the existing file and put the output of the command into that file. No warning is given that you are erasing an existing file. If you want to assure yourself that there is not an existing file, first execute the **ls** command with the file name as an argument.

If the file exists, **ls** will list the file. If the file does not exist, **ls** will tell you the file was not found in the current directory.

```
$ ls filename<CR>
filename
$ ls junk<CR>
junk not found
$
```

The double redirection symbol >> appends the output of a command after the last line of a file.

The general format to append output to a file is:

command >> filename<CR>

In the next example, the contents of *trial2* are added after the last line of *trial1* by redirecting the **cat** command output of *trial2* into *trial1.*

The first command, **cat trial1**, displays the contents of *trial1.* Then, **cat trial2** displays the contents of *trial2.* The third command line, **cat trial2 >> trial1**, adds the contents of *trial2* to the bottom of file *trial1*, and **cat trial1** displays the new contents of *trial1.*

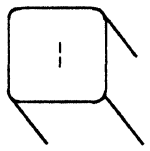
```
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
$
$ cat trial2<CR>
Add this to file trial1
This is the last line of file trial2
$
$ cat trial2 >> trial1<CR>
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
Add this to file trial1
This is the last line of file trial2
$
```

In the section on *Special Characters*, one of the examples showed how to execute the **grep** command in background mode with &. Now, you can redirect the output of that command into a file called *wordfile*, and then look at the file when you have finished your current task. The & is the last character of the command line.

```
$ grep word * > wordfile &<CR>
$
```

Pipes

The ¦ character is called a pipe. It redirects the output of one command to be the input of another command.



This character directs the output from one command to be the input of the next command.

If two or more commands are connected by a pipe, ¦ , the output of the first command is "piped" into the next command as the input for that command.

The general format for the pipe line is:

command1 ¦ command2 ¦ command3<CR>

The output of **command1** is used as the input of **command2**. The output of **command2** is then used as the input for **command3**.

You have already tried the banner display on your terminal. The pipe can be used to send a banner birthday greeting to someone by electronic mail.

If the person using login **david** has a birthday, pipe the banner display of happy birthday into the **mail** command.

Type in: **banner happy birthday ¦ mail david<CR>**

Login **david** will get a banner display in his electronic mail.

The **date** command gives you the date and the time. Since you may not have used the **date** command before, a brief recap of **date** follows.

Command Recap

date - display the date and time

command	*options*	*arguments*
date	**+%m%d%y*** **+%H%M%S**	available*

Description: **date** displays the current date and time on your terminal.

Options: +% followed by m for month, d for day, y for year, H for hour, M for month, and S for second will echo these back to your terminal. You can add an explanation to these such as:

date '+%H:M is the time'

Remarks: If you are working on a small computer system in which you are acting as both user and system administrator, you may be able to set the date and time using optional arguments to the **date** command. Check your reference manual for details. When working in a multiuser environment, the arguments are available only to the system administrator.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Try out the **date** command on your terminal.

```
$ date<CR>
Mon Nov 25 17:57:21 CST 1985
$
```

Notice that the time is given from the 12th character through the 19th character. If you want to know just the time and not the date, you can pipe the output of the **date** command into the **cut** command. The **cut** command looks for characters only in a specified part of each line of a file. If you use the **−c** option, **cut** will choose only those characters in the specified character positions. Character positions are counted from the left. To display only the time on your terminal pipe the output of the **date** command into the **cut** command asking for characters 12 through 19.

```
$ date | cut −c12-19<CR>
18:08:23
$
```

Several pipes can be used in one command line. The output of the example can be piped into the **pr** command.

Type in: **date | cut −c12-19 | pr<CR>**

Try each of these examples. Check the system response.

Later in this chapter, you will write a shell program that will give you the time.

Since you may not have used the **cut** command until now, a brief recap of that command follows next.

Command Recap

cut - cut out selected fields of each line of a file

command	*options*	*arguments*
cut	−clist −flist [−d]	**file1 file2**

Description: **cut** will cut out columns from a table or fields from each line of a file.

Options: **−c** lists the number of character positions from the left. A range of numbers such as characters 1-9 can be specified by **−c1-9**

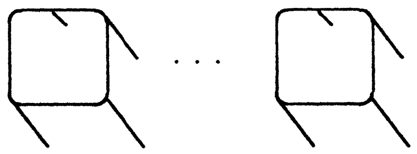
−f lists the number of fields from the left separated by a delimiter described by **−d**.

−d gives the field delimiter for **−f**. The default is a tab. If the delimiter is a colon, this would be specified by **−d :** .

Remarks: If you find the **cut** command useful, you may also want to use the **paste** command and the **split** command.

Command Output Substitution

The output of any command line or shell program that is enclosed in back quotes, \` , can be substituted anywhere on a shell command line. In the section on *Shell Programming*, you will substitute the output of a command line as the value for a variable.



Substitute the output of the command line in back quotes.

The output of the time command can be substituted for the argument in a banner printout.

```
Type in:   banner `date | cut -c12-19` <CR>
```

Did you get a banner display of the time?

Executing and Terminating Processes

Running Commands at a Later Time

When you type in a command line at your terminal, the ICON/UXV system tries to execute that command immediately. It is possible to tell the system to execute those commands at another time with the **batch** or the **at** command. End the commands with **<^d>** to let the shell know you have finished listing the commands to be executed.

The **batch** command is useful if you are running a process or shell program that uses a longer then normal amount of system time. The **batch** command submits a "batch" job, which consists of the commands to be executed, to the system. The job is put in a queue, and then the job is run when the load on the system falls to an acceptable level. This frees the system to rapidly respond to other input by yourself or others on the system.

The general format for **batch** is:

```
batch<CR>
first command<CR>
.
.
.
last command<CR>
<^d>
```

If there is only one command line, it may follow the **batch** command.

```
batch command line<CR>
<^d>
```

The next example uses the **batch** command to execute the **grep** command at a convenient time. When the system can execute that command and still respond quickly to other users, it will execute the **grep** command to search all the files for the word **dollar**, and redirect the output into the file *dol–file*. Using the **batch** command is a courtesy to other users sharing your ICON/UXV system.

```
$ batch grep dollar * > dol-file<CR>
<^d>
job 155223141.b at Mon Dec 7 11:14:54 1983
$
```

A brief recap of the **batch** command follows.

Command Recap

batch - execute commands at a later time

command	*options*	*arguments*
batch	none	**command lines**

Description:	**batch** submits a "batch job", which is placed into a queue and executed when the load on the system falls to an acceptable level.
Remarks:	The list of commands must end with a **<^d>** to tell the system the last command has been typed in for this batch job.

The **at** command gives the system a specific time that the commands are to be executed. The general format for the **at** command is:

```
at time<CR>
first command<CR>
.
.
.
last command<CR>
<^d>
```

The **time** must first give the time of day and then the date, if the date is not today.

If you are afraid you will forget login *david*'s birthday, you can use the **at** command to make sure the banner birthday greeting will arrive on his birthday.

```
$ at 8:15am Feb 27<CR>
banner happy birthday | mail david<CR>
<^d>
job 453400603.a at Mon Feb 27 08:15:00 1984
$
```

Both the **batch** and **at** commands give you a job number. If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs with the **−r** option of the **at** command and the job number. The general format is:

at −r jobnumber<CR>

Try erasing the previous **at** job for the happy birthday banner.

Type in: **at −r 453400603.a<CR>**

If you have forgotten the job number, the **at −l** command will give you the current jobs in the **batch** or **at** queue.

```
$ at -l<CR>
: login 168302040.a a Tue  Nov 29 13:00:00 1983
: login 453400603.a a Mon Feb 27 08:15:00 1984
$
```

login will be your login name.

Try the following request. Using the **at** command, mail yourself a file at noon. The file, called *memo*, says that it is lunch time. You must redirect the file into **mail**. (You cannot type in the text directly unless you use the here document discussed in the *Shell Programming* section.) Now try the **at** command with the **−l** option.

```
$ at 12:00pm<CR>
mail mylogin < memo<CR>
<^d>
job 263181754.a at Jun 30 12:00:00 1985
$
$ at -l<CR>
: mylogin 263181754.a at Jun 30 12:00:00 1985
$
```

A brief recap of the **at** command follows next.

Command Recap

at - execute a list of commands at a specified time.

command	*options*	*arguments*
at		**time date**
	−r	**jobnumber**
	−l	

Description: Executes commands at the time specified. The order of the arguments is the time which can be 1 to 4 digits and "am" or "pm". The date need not be added if it is today. The date is specified by a month name followed by the number for the day.

Options: The **−r** option with the job number removes previously scheduled jobs.

The −l option without arguments gives the status of **at** and **batch** jobs and job numbers.

Remarks: Some times and dates are: at 08:15am Feb 27 *and* at 5:14pm Sept 24.

Obtaining the Status of Running Processes

The **ps** command will give you the status of the processes you are running.

Running a process or command in background with **&** was discussed in the section on special characters. The **ps** command will tell you the status of those processes. In the next example, the **grep** command was run in the background, and then the **ps** command was typed in. The system response, the output from the **ps** command, gives the PID, which is the process identification number, and TTY, which is the current number identification assigned to the terminal you are logged in on. It also gives the cumulative execution TIME for each process, and the COMMAND that is being executed. The PID is an important number if you decide to stop the execution of that command.

```
$grep word * &<CR>
28223
$
$ ps<CR>
PID   TTYTIME COMMAND
28124 100:00    sh
28223 100:04    grep
28224 100:04    ps
$
```

The example not only gives you the PID for the **grep** command, but also for the other processes that are running, the **ps** command itself, and the **sh** command that is always running as long as you are logged in. **sh** is the shell program that interprets the shell commands. It is discussed in *Chapter 1* and *Chapter 4*.

Command Recap

ps - report process status

command	*options*	*arguments*
ps	several*	none

Description: Displays information about active processes.

Options: This command has several options. If you do not use any options you will get the status of the active shell processes that you are running.

Remarks: Gives you the PID, the Process ID. This is needed if you are going to **kill** the process, that is, stop the process from executing.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Terminating Active Processes

The **kill** command is used to stop active shell processes. The general format for the **kill** command is:

kill PID<CR>

What do you do if you decide you do not need to execute the command that you are running in the background?. If you press the BREAK key or the DEL key, you will find it does not stop the background process as it does the interactive commands. The **kill** command terminates a background process. If you want to terminate the **grep** command used in the previous example:

```
$ kill 28223<CR>
28223 Terminated
$
```

A recap of the **kill** command follows.

Command Recap

kill - terminate a process

command	*options*	*arguments*
kill	available*	**job number or PID**
Description:	**kill** will terminate the process given by the PID.	

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Using the No Hang Up Command

Another way to kill all processes is to hang up on the system, to log off. What if you want the background process to continue to run after you have logged off? The **nohup** command will allow background commands to continue to run even if you log off.

nohup command &<CR>

If you place the **nohup** command at the beginning of the command that you will be running as a background process, the background process will continue to run to completion after you have logged off.

Type in: **nohup grep word * > word.list &<CR>**

The **nohup** command can be stopped by the **kill** command. The recap of the **nohup** command is the following:

Command Recap

nohup - runs a command, ignoring hanging up or quitting the system

command	*options*	*arguments*
nohup	none	**command line**

Description:	Executes a command line, even if you hang up or quit the system.

Now that you have mastered these shortcuts in the shell commands, use them in your shell programs.

COMMAND LANGUAGE EXERCISES

1-1. What happens if you use the * at the beginning of a file name? Try to list some of the files in a directory using the * with a last letter of one of your file names. What happens?

1-2. Try out the following two commands.

Type in: **cat [0-9]*<CR>**
echo *<CR>

1-3. Can you use **?** at the beginning or in the middle of a file name generation? Try it.

1-4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lowercase letter between a and m? (Hint use a range of numbers or letters in [ ]).

1-5. Can you place a command in background mode on the line that is executing several other commands sequentially. Try it. What happens? (Hint use ; and **&**.) Can the command in background mode be placed at any position on the command line? Try it. Experiment with each new character that you learn, so that you can learn the full power of the character.

1-6. Using the command line

cd; pwd; ls; ed trial<CR>

redirect the output of **pwd** and **ls** into a file. Remember, if you want to redirect both commands to the same file, you have to use >> for the second redirection or you will wipe out the information from the **pwd** command.

1-7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part that you need to change in the "time" command line?

```
banner `date ¦ cut -c12-19`
```

SHELL PROGRAMMING

Getting Started

Let a shell program perform your tasks for you. A shell program is a ICON/UXV file that contains the commands that you would use to perform your task.

- How do you create a simple shell program?
- What makes the program run?
- Is there a special directory for your shell programs?

In this section of the tutorial you will learn the answers to these questions. The examples for creating shell programs usually show two display screens. The first screen displays the contents of the file containing the commands used in your program. It shows the command line

cat file<CR>

and the system response to that command, which is the contents of the file.

```
$ cat file<CR>
First command
     .
     .
     .
Last command
$
```

The **$** indicates the shell prompt. The second screen shows the results of executing your shell program.

```
$ file<CR>
Results
$
```

The names of the file containing the shell program will be printed in **bold** in the text, since it is a command and not an ordinary text file.

Before you begin to create shell programs, you should be familiar with one of the editors. The editors are discussed in the tutorials in *Chapter 5* and *Chapter 6.*

Creating a Simple Shell Program

How do you think you would create a simple shell program that would:

- Tell you the directory you were in,
- List the contents of that directory, and then
- Display on your terminal: "This is the end of the shell program".

Think about it now before you read any further.

To create the shell program, you will need the following three shell commands:

pwd	The command that prints the path name of the current directory,
ls	The command that lists the contents of the current directory, and
echo	The command that displays on your terminal the characters following **echo**.

To create your shell program, using **pwd**, **ls**, and **echo**, enter an editor and type in the following three commands.

Type in:

```
pwd<CR>
ls<CR>
echo This is the end of the shell program.<CR>
```

Write the contents of the editor buffer to a file called **dl** (for directory list) and quit the editor. You have just created a shell program.

```
$ cat dl<CR>
pwd
ls
echo This is the end of the shell program.
$
```

Executing a Shell Program

How do you tell the shell that your file is a shell program that needs to be executed? The simplest way to execute a program is to use the **sh** command.

Type in: **sh dl<CR>**

What happened?

Did you notice the path name of the current directory printed out first, then the list of the contents of the current directory, and last of all the comment *This is the end of the shell program.* ?

The **sh** command is a good way to test out your shell program to make sure that it works.

If **dl** is a useful command, you will want to change the file permissions so that you need only type in **dl** to execute the command. The command that changes the permissions on a file, **chmod**, is discussed in *Chapter 3*. The example below reminds you how to type in the **chmod** command to make a file executable, and then do an **ls −l** so you can see the change in the permissions.

```
$ chmod u+x dl<CR>
$  ls −l<CR>
total 4
 -rw-------   1      login  login    3661  Nov  210:28 mbox
drwxrwxrwx    2      login  login    1056  Nov 1118:20 rje
 -rwx------   1      login  login      48  Nov 1510:50 dl
$
```

Now you have an executable program **dl** in your current directory.

Type in: **dl<CR>**

Did the **dl** command execute?

Creating a bin Directory for Executable Files

If your shell program is useful, you will want to keep it in a special directory called *bin*, which is under your login directory.

If you want your **dl** command accessible from all your directories, make a *bin* directory from your login directory and move the **dl** file to your *bin*. Below is a reminder of those commands. In this example, **dl** is in the login directory.

Type in:
```
mkdir bin<CR>
mv dl bin/dl<CR>
```

Move to the *bin* directory and type in the **ls −l** command. Does **dl** still have execute permission?

Now move to another directory other than the login directory.

Type in:
```
dl<CR>
```

What happened?

A command recap of your new program **dl** follows.

Shell Program Recap

dl - display the directory path and directory contents

command	*arguments*
dl	none

Description:	Displays the output of the shell command **pwd** and then lists the contents of the directory.

The *bin* is the best place to keep your executable shell programs. It is possible to give the *bin* directory another name, but you need to change the shell variable **PATH** to do so. The shell variables are discussed briefly in this chapter. For more advanced information read the document *Shell Commands and Programming* in Part 3 of this manual.

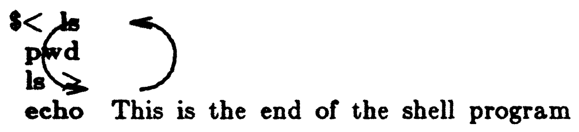
Problem:
You can give your shell program file any appropriate file name. However, you should not name your program with the same name as a system command. The system will execute your command and not the command of the system.

If you had named your **dl** program **mv**, each time you tried to move a file, the system would not move your file. It would have executed your program to display the directory name and list the contents.

Problem:
Another problem would occur if you had named the **dl** file **ls**, and then tried to execute the file **ls**. You would create an infinite loop. After some time, the system would give you an error message:

Too many processes, cannot fork

What happened? You typed in your new command **ls**. The shell read the command **pwd** and executed that command. Then the shell read the command **ls** in your file and tried to execute your **ls** command. This formed an infinite loop:



ICON/UXV system designers wisely set a limit on how many times this infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, **/bin/ls**.

The following **ls** shell program would work.

```
$ cat ls<CR>
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system command with **/bin/ls**.

Variables

If you enjoyed sending the **banner** birthday greeting, you could make a shell program that would pipe the banner printout into the electronic **mail**. A good shell program would let you send to a different login each time you executed the program. The login would then be a variable. There are two ways you can specify a variable for a shell program:

- Positional parameters and
- Variables that you define.

Positional Parameters

A positional parameter is a variable that is found in a specified position in the command line of your shell program. Positional parameters are typed in after the command. They are strings of characters delimited by spaces, except for the last parameter, which is ended with <CR>. If **pp1** is the first positional parameter, **pp2** is the second positional parameter, and ... **pp9** is the ninth positional parameter, then the command line of your shell program called **shell.prog** will look like this.

```
shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9<CR>
```

The shell program will take the first positional parameter (**pp1**) and substitute it in the shell program text for the characters **$1**. The second positional parameter (**pp2**) will be substituted for the characters **$2**. The ninth positional parameter (**pp9**), of course, will be substituted for the characters **$9**.

If you want to see how the positional parameters are substituted into a program, try typing the following lines into a file called **pp** (positional parameters).

```
Type in:  echo  The first positional parameter is: $1<CR>
          echo  The second positional parameter is: $2<CR>
          echo  The third positional parameter is: $3<CR>
          echo  The fourth positional parameter is: $4<CR>
```

First the **echo** command tells which parameter will be displayed and then displays the parameter. The next example shows the contents of the file **pp**.

```
$ cat pp<CR>
echo The first positional parameter is: $1
echo The second positional parameter is: $2
echo The third positional parameter is: $3
echo The fourth positional parameter is: $4
$
```

The following example shows the results of giving the four positional parameters one, two, three, and four to the shell program **pp**. Remember to change the mode of **pp** to be executable.

```
$ chmod u+x pp<CR>
$
$ pp one two three four<CR>
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

Now, return to creating your shell program for the banner birthday greeting. Call the file **bbday**. What command line would go into that file? Before you go on reading, try it.

Did you get the following?

```
$ cat bbday<CR>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is *slowmo*, then the command line would be:

```
$ bbday slowmo<CR>
you have mail
$
```

The following is a brief recap of the shell program command **bbday**.

Shell Program Recap

bbday - mail a banner birthday greeting

command	*arguments*
bbday	**login**

Description: **bbday** mails "happy birthday" in poster-sized letters to the specified login.

The **who** command will tell you every login that is currently using the system. How would you make a simple shell program called **whoson** that will tell you if a particular login is currently working on the system? You could try the following:

```
$ who | grep boss<CR>
boss     tty51Nov 29 17:01
$
```

This command pipes the output of the **who** command into the **grep** command. The **grep** command is searching for the characters "boss". Since login *boss* is currently logged into the system, the shell will respond with:

```
boss   tty51      Nov 29 17:01
```

If the only response is a prompt sign, then login *boss* is not currently on the system because the **grep** command found nothing. Create a **whoson** shell program.

Below are the ingredients for your shell program **whoson**.

who	The shell command that lists everyone on the system,
grep	The search command, and
$1	The first positional parameter for your shell program.

The **grep** command searches the output of the **who** command for the parameter designated in the program by **$1**. If it finds the login, it will display the line of information. If it does not find the login in the output from **who**, it will display your prompt.

Enter an editor and type the following command line into a file called **whoson**.

Type in: **who | grep $1<CR>**

Write the file, quit the editor, and change the mode of the file **whoson** to have execute permission.

Now try using your login as the positional parameter for the new program **whoson**. What was the system's response?

If your login name is *slowmo*, your new shell command line would look like:

```
$ whoson slowmo<CR>
slowmo      tty26         Jan 24 13:35
$
```

The first positional parameter is **slowmo**. The shell substitutes **slowmo** for the **$1** in your program.

```
who | grep slowmo<CR>
```

The following is a brief recap of the **whoson** command.

Shell Program Recap

whoson - display login information if user is logged in

command	*arguments*
whoson	**login**

Description:	If a user is on the system, displays the user's login, the **TTY** number, the time and date the user logged in.

The shell command line will allow 128 positional parameters. However, your shell program text is restricted to **$1** through **$9**, unless you use the **$*** described below, or the **shift** command, which is described in *Part 3, Shell Commands and Programming,* of this manual.

Parameters with Special Meaning

$# This variable in your shell program will record and display the number of positional parameters you typed in for your shell program.

Let's look at an example that will show you what happens when you use **$#**. Put the following command lines in a shell program called **get.num**.

```
$ cat get.num<CR>
echo The number of parameters is: $#
$
```

The program counts all the positional parameters and displays that number. Give **get.num** four parameters. They can be any string of characters.

```
$ get.num test out this program<CR>
The number of parameters is: 4
$
```

Shell Program Recap

get.num - count and display the number of arguments

command	*arguments*
get.num	(any string)

Description: **get.num** counts the number of arguments given to the command and then displays that number.

Remarks: This command demonstrates the special parameter **$#**.

$*** This variable in your shell program will substitute all positional parameters starting with the first positional parameter. The parameter **$* does not restrict you to nine parameters.

You can make a simple shell program to demonstrate **$***. Make a shell program called **show.param** that will **echo** all of the parameters. Type in the **echo** command line shown in the following screen.

```
$ cat show.param<CR>
echo The parameters for this command are: $*
$
```

Make **show.param** executable and try it out.

```
$ show.param hello how are you<CR>
The parameters for this command are: hello how are you
$
```

Now try **show.param** using more than nine positional parameters.

```
$ show.param one two 3 4 5 six 7 8 9 10 11<CR>
The parameters for this command are: one two 3 4 5 six
7 8 9 10 11
$
```

The **$*** is very handy if file generation names are used as the parameters.

Try a file name generation parameter in your **show.param** command. If you have several chapters of a manual in your directory called chap1, chap2 through chap7, you will get a printout listing of all of those chapters.

```
$ show.param chap?<CR>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

A quick recap of **show.param** follows.

Shell Program Recap

show.param - display all of the parameters

command	*arguments*
show.param	(any positional parameters)

Description:	**show.param** displays all of the parameters.
Remarks:	If the parameters are file name generations, it will display each of those file names.

You may want to practice with positional parameters so that they are familiar to you before you continue on to the next section in which you will name the variables within the program, rather than use them as arguments in a command line.

Variable Names

The shell allows you to name the variables within a shell program. Naming the variables in a shell program makes it easier for another person to use. Instead of using positional parameters, you will tell the user what to type in for the variable, or you will give the variable a value that is the output of a command.

What does a named variable look like? In the example below, **var1** is the name of the variable and **myname** is the value or character string assigned to that variable. There are no spaces on either side of the = sign.

```
var1=myname<CR>
```

Within the shell program, a **$** in front of the variable name alerts the shell that a substitution is needed in the shell program. **$var1** tells the shell to substitute the value **myname**, which was given to **var1**, for the characters **$var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in the case of shell program file names, it is a risky business to use a shell command as a variable name. Also, the shell has reserved some variable names to be used by the shell. The following names are used by the shell and should not be used as the name of one of your variables. A brief explanation of each variable is given.

CDPATH
This variable defines the search path for the **cd** command.

HOME
This is the default variable for the **cd** command (Home Directory).

IFS
This variable defines the internal field separators, normally the space, the tab, and the carriage return.

MAIL
This variable is set to the name of the file that contains your electronic mail.

PATH
This variable determines the path that is followed to find commands.

PS1
PS2
These variables define the primary and secondary prompt strings. The defaults are **$** and >. Do you have a prompt sign **$** ?

TERM
This variable tells the shell what kind of terminal you are working on. It is important to set this variable if you are editing with **vi**.

Many of these named variables are explained in the last section of this chapter on your login environment.

Assign Values to Variables

If you edit with **vi**, you know that you must set the variable TERM to equal the code for your type of terminal before you use the **vi** editor. For example:

```
TERM=T3<CR>
```

This is the simplest way to assign a value to a variable.

There are several other ways to assign values to variables. One way is to use the **read** command to assign input to the variable. Another way is to assign the value from the output of a command using back quotes ` ... `. A third way would be to assign a positional parameter to the variable.

Assign Values by the Read Command

You can set up your program so that you can type in the command and then be prompted by the program to type in the value for the variable. The **read** command assigns the input to the specified variable. The general format for the **read** command is:

```
read var<CR>
```

The values assigned by **read** to **var** will be substituted for **$var** in the program. If the **echo** command is executed just before the **read** command, the program can display the directions "type in ...". The **read** command will wait until you type in the value, and then assign the string of characters that you type in as the value for the variable.

If you had a list that contained the names and telephone numbers of people you called often, you could make a simple shell program that would automatically give you someone's number. Stop for a minute. How would you make up the program using the following ingredients?

echo The command that echoes the instructions.

read The command that assigns the input value to the variable **name**.

grep The command that searches for the person's name and number.

First, you would use the **echo** command to inform the user to type in the name of the person to be called.

```
echo Type in the last name<CR>
```

The **read** command will then assign the person's name to the variable **name**.

```
read name<CR>
```

Notice that you do not use the = to assign the variable, the **read** command automatically assigns the typed in characters to **name**.

The **grep** command will then search your phone list for the name. If your phone list were called *list*, the command line would be:

```
grep $name list<CR>
```

In the next example, the shell program is called **num.please**. Remember, the system response to the **cat** command is the contents of the shell program file.

```
$ cat num.please<CR>
echo Type in the last name
read name
grep $name list
$
```

Make a list of last names and phone numbers and try **num.please**. Or, try the next example, which is a program that creates a list. You can use several variables in one program. If you have a phone list, you may want a quick and easy way to add names and numbers to the list. The program:

- Asks for the name of the person,

- Assigns the name to the variable **name**,
- Asks for the person's number,
- Assigns the number to the variable **num**, and
- Echos the **name** and **num** into the file *list*. You must use >> to redirect the output of the **echo** command to the bottom of your list. If you use >, your list will contain only the last phone number.

The program is called **mknum**.

```
$ cat mknum<CR>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$
$ chmod u+x mknum<CR>
$
```

Now try out the new programs for your phone list. In the next example, **mknum** creates the new listing for Mr. Niceguy. Then, **num.please** gives you Mr. Niceguy's phone number.

```
$ mknum<CR>
Type in the name
Mr. Niceguy<CR>
Type in the number
668-0007<CR>
$
$ num.please<CR>
Type in last name
Niceguy<CR>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Here is a brief recap of **mknum** and **num.please**.

Shell Program Recap

mknum - place name and number on a phone list

command	*arguments*
mknum	(interactive)

Description:	Asks you for the name and number of a person and adds the name and number to your phone list.
Remarks:	This is an interactive command.

Shell Program Recap

num.please - display a person's name and number

command	*arguments*
num.please	(interactive)

Description:	Asks you for a person's last name, and then displays the name and telephone number.
Remarks:	This is an interactive command.

Substitute Command Output for the Value of a Variable

Another way to assign a value to a variable is to substitute the output of a command for the value. This will be very useful in the next section when you try loops and conditional constructs.

The general format to assign output as the value for a variable is:

```
var=`command`<CR>
```

The variable **var** has the value of the output from **command**.

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was:

```
date | cut -c12-19<CR>
```

You can place that command in a simple shell program called **t** that will give you the time.

```
$ cat t<CR>
time=`date | cut -c12-19`
echo The time is: $time
$
```

Remember there are no spaces on either side of the equal sign.

Change the mode on the file and you now have a program that gives you the time.

```
$ chmod u+x t<CR>
$ t<CR>
The time is: 10:36
$
```

The recap for the **t** shell program follows.

Shell Program Recap

t - display the correct time

command	*arguments*
t	none

Description: **t** gives you the correct time in hours and minutes.

Assign Values with Positional Parameters

A positional parameter can be assigned to a named parameter. For example:

var1=$1<CR>

The example below is a simple program **simp.p** that demonstrates how you can assign a positional parameter to a variable. The command lines in the file would be the following:

```
$ cat simp.p<CR>
var1=$1
echo $var1
$
```

Or, you can assign the output of a command that uses a positional parameter.

person=`who | grep $1` <CR>

If you wanted to keep track of the results of your **whoson** program, you could create the program **log.time**. The output of your **whoson** shell program is assigned to the variable **person**. Then, that value **$person** is added to the file *login.file* with the **echo** command. The last part of the program displays the value of **$person**, which is the same as the response to the **whoson** command.

```
$ cat log.time<CR>
person=`who`| grep $1
echo $person >> login.file
echo $person
$
```

The system response to **log.time** would appear as in the following screen.

```
$ log.time maryann<CR>
maryann       tty61           Apr 11 10:26
$
```

The following is a quick recap of the **log.time** program.

Shell Program Recap

log.time - log and display a specified login that is currently logged in

command	*arguments*
log.time	**login**

Description: If the specified login is currently on the system, **log.time** places the line of information from the **who** command into the file *login.file* and then displays that line of information on your terminal.

As you do more programming, you may discover other ways to assign variables that will help you in shell programs.

Shell Programming Constructs

The shell programming language has several constructs that give you more flexibility in your programs.

- The "here document" allows you to redirect lines of input into a command.
- The looping constructs **for** or **while** cause a program to reiterate commands in a loop.
- The conditional control commands, **if** or **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command gives the unconditional end of a loop.

Comments

Before you begin writing shell programs with loops, you may want to know how to put comments about your program into the file, which the system will ignore. To place comments in a program, begin the comment with # and end it with <CR>. The general format for a comment line is:

```
#comment<CR>
```

The shell will ignore all characters after the #. These lines

```
# This program sends a generic birthday greeting<CR>
# This program needs a login as the positional parameter<CR>
```

will be ignored by the system when your program is being executed. They only serve as a reminder to you, the programmer.

The Here Document

The here document allows you to redirect lines of input of a shell program into a command. The here document consists of the redirection symbol << and the delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters. The ! is often used as a delimiter. The general format for the here document is:

```
command <<!<CR>
...input lines...<CR>
!<CR>
```

The here document could be used in a shell program, to redirect lines of input into the **mail** command. The program shown below sends a generic birthday greeting with the **mail** command. The program is called **gbday**.

```
$ cat gbday<CR>
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

The person's login is the first positional parameter **$1**.

The redirected input is:

Best wishes to you on your birthday.

To send the greeting:

```
$ gbday mary<CR>
$
```

To receive the greeting, login *mary* would execute the **mail** command.

```
$ mail<CR>
From mylogin Mon May 14 14:31 CDT 1984
Best wishes to you on your birthday
$
```

The following is a recap of **gbday**.

Shell Program Recap

gbday - send a generic birthday greeting

command	*arguments*
gbday	**login**

Description: **gbday** sends a generic birthday greeting to the login given as an argument.

Using ed in a Shell Program

The line editor **ed** can be used within a shell program if it is combined with the here document commands.

Suppose you want to make a shell program that will enter the editor, **ed**, make a global substitution to a file, write the file, and then quit the editor. The **ed** command to make a global substitution is:

g/text to be changed/s//new text/g<CR>

Before you read any further, jot down what you think the command sequence will be. Put your command sequence into a file called **ch.text**. If you want to suppress the character count of **ed** so that it will not appear on your terminal, use the – option:

ed – filename<CR>

Try to execute the file. Did it work?

If you used the **read** command to enter the variables, your program **ch.text** may look similar to what appears in the following screen.

```
$ cat ch.text<CR>
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read oldtext
echo Type in the exact new text to replace the above.
read newtext
ed – $file1 <<!
g/$oldtext/s//$newtext/g
w
q
!
$
```

This program uses three variables. Each of them is entered into the program with the **read** command.

$file	The name of the file to be edited.
$oldtext	The exact text to be changed.
$newtext	The new text.

Once the variables are entered into the program, the here document redirects the global, write, and quit commands into the **ed** command. Try out the new **ch.text** command.

```
$ ch.text<CR>
Type in the filename.
memo<CR>
Type in the exact text to be changed.
Dear John:<CR>
Type in the exact new text to replace the above.
To whom it may concern:<CR>
$ cat memo<CR>
To whom it may concern:
$
```

Did you try to use positional parameters? Did you have any problems entering the text changes as variables, or did you quote the character strings for each parameter?

The recap of the **ch.text** command is:

Shell Program Recap

ch.text - change text in a file

command	*arguments*
ch.text	(interactive)

Description: Replaces text in a file with new text.

Remarks: This shell program is interactive. It will prompt you to type in the arguments.

If you want to become more familiar with the line editor **ed**, see *Chapter 5, Line Editor Tutorial (ed).*

The stream editor **sed** can also be used in shell programming. More information on that editor can be found in the *ICON/UXV Editing Guide.* (See *Appendix A.*)

Looping

Until now, the commands in your shell program have been executed once and only once and in sequence. Looping constructs give you repetitive execution of a command or group of commands. The **for** or **while** commands will cause a program to loop and execute a sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands for each member of a list. The **for** command loop also requires the keywords **in, do,** and **done**. The **for, do,** and **done** keywords must be the first word on a line. The general format of the **for** loop is:

```
for variable<CR>
      in this list of values<CR>
do the following commands<CR>
     command 1<CR>
     command 2<CR>
           .
           .
           .
     last command<CR>
done<CR>
```

The variable can be any name you choose. If it is **var**, then the values given after the keyword **in** will be sequentially substituted for **$var** in the command list. If **in** is omitted, the values for **var** will be the positional parameters. The command list between the keywords **do** and **done** will be executed for each value.

When the commands have been executed for the last value, the program will execute the next line below **done**. If there is no line, the program will end.

It is easier to read a shell program if the looping constructs stand out. Since the shell ignores spaces at the beginning of the lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can quickly check to make sure each **do** has a corresponding **done** statement to end the loop.

The easiest way to understand a shell programming construct is to try an example. Try to create a program that will move files to another directory.

The ingredients for the program are:

echo	You want to echo directions to type in the path name to reach the new directory.
read	You want to type in the path name, and assign it to the variable **path**.
for variable	You must name the variable. Call it **file** for your shell program. It will appear as **$file** in the command sequence.
in list of values	The list of values will be the file names. If the **in** clause is omitted, the list of values is taken to be **$*** , that is, the parameters entered on the command line.
do command sequence	The command sequence for this program is: **mv $file $path/$file<CR>**
done	

Your shell program text for the program called **mv.file** will look like:

```
$ cat mv.file<CR>
echo Please type in the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done
$
```

Notice that you did not type in any values for the variable **file**. The values are already in your program. If you want to change the files each time you invoke the program, use positional parameters or variables that you name. You do not need the **in** keyword to list the values when you use positional parameters. If you choose positional parameters, your shell program will look like:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done
$
```

It is likely that you will want to move several files using the special file name generation characters.

If this is a useful command, remember to move it into your *bin*.

Following is a recap of the **mv.file** shell program.

Shell Program Recap

mv.file - move files to another directory

command	*arguments*
mv.file	**file names (*)** (interactive)

Description:	Moves files to a directory.
Remarks:	This program requires the file names to be given as positional parameters. The path to the new directory is asked for interactively by the program.

The **while** Loop

The **while** loop will continue executing the sequence of commands in the **do...done** list as long as the final command in the **while** command list returns a status of true, that is can be executed. The **while, do,** and **done** keywords must be the first characters on the line. The general format of the **while** loop is the following:

```
while<CR>
      command 1<CR>
            .
            .
            .
     last command<CR>
do<CR>
     command 1<CR>
            .
            .
            .
     last command<CR>
done<CR>
```

A simple program using the **while** loop enters a list of names into a file. The command lines for that program called **enter.name** are:

```
$ cat enter.name<CR>
while
   read x
do
   echo $x>>xfile
done
$
```

This shell program needs some instructions. You have to know to delimit or separate the names by a <CR>, and you have to use a <^d> to end the program. Also, it would be nice if your program displayed the list of names in the *xfile* at the end of the program. If you added those ingredients to the program, the commands lines for the program become:

```
$ cat enter.name<CR>
echo 'Please type in each person's name and then a <CR>'
echo 'Please end the list of names with a <^d>'
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Notice that after the loop is completed, the program executes the commands below the **done**.

In the **echo** command line, you used characters that are special to the shell, so you must use the **'...'** to turn off that special meaning. Put the above command lines in an executable file and try out the shell program.

```
$ enter.name<CR>
Please type in each person's name and then a <CR>
Please end the list of names with a <^d>
Mary Lou<CR>
Janice<CR>
<^d>
xfile contains the following names:
Mary Lou
Janice
$
```

Conditional Constructs if...then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**. The general format for the **if** construct is as follows:

```
if<CR>
     command1<CR>
        .
        .
        .
     last command<CR>
   then<CR>
        command1<CR>
           .
           .
           .
        last command<CR>
fi<CR>
```

The next shell program demonstrates the **if...then** construct. The program will search for a word in a file. If the **grep** command is successful then the program will **echo** that the word is found in the file. In this example the variables are read into the shell program. Type in the shell program shown below and try it out. Call the program **search**.

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

Notice that the **read** command is assigning values to two variables. The first characters that you type in, up to a space, are assigned to **word**. All of the rest of the characters including spaces will be assigned to **file**.

Pick a word that you know is in one of your files and try out this shell program. Did you see that even though the program works, there is an irritating problem? Your program displayed more than the line of text called for by the program. The extra lines of text displayed on your terminal were the output of the **grep** command.

The Shell Garbage Can /dev/null

The shell has a file that acts like a garbage can. You can deposit any unwanted output in the file called */dev/null*, by redirecting the command output to */dev/null*.

Try out */dev/null* by throwing out the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the response into the file */dev/null*.

```
who > /dev/null<CR>
```

The response displayed on your terminal was your prompt. The response to the **who** command was placed in */dev/null* and became null, or nothing. If you want to dispose of the **grep** command response in your **search** program, change the **if** command line.

```
if grep $word $file > /dev/null<CR>
```

Now execute your **search** program. The program should only respond with the text of the **echo** command line.

The **if...then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. The general format of the **if...then...else** construct follows.

```
if<CR>
     command1<CR>
         .
         .
         .
     last command<CR>
  then<CR>
       command1<CR>
           .
           .
           .
       last command<CR>
  else<CR>
       command1<CR>
           .
           .
           .
       last command<CR>
fi<CR>
```

You can now improve your **search** command. The shell program **search** can look for a word in a file. If the word is found, the program will tell you the word is found. If it is not found (**else**) the program will tell you the word was NOT found. The text of your **search** file will look like the following:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

Following is a quick recap of the enhanced shell program called **search**.

Shell Program Recap

search - tell if a word is in a file

command	*arguments*
search	interactive

Description:	Tells the user whether or not a word is in a file.
Remarks:	The arguments, the word and the file, are asked for interactively.

The test Command for Loops

test is a very useful command for conditional constructs. The **test** command checks to see if certain conditions are true. If the condition is true, then the loop will continue. If the condition is false, then the loop will end and the next command is executed. Some of the useful options for the **test** command are:

test –r filename<CR>
True if the file exists and is readable

test –w filename<CR>
True if the file exists and has write permission

test –x filename<CR>
True if the file exists and is executable

test –s filename<CR>
True if the file exists and has at least one character

If you have not changed the values of the **PATH** variable that were initially given to you by the system, then the executable files in your *bin* directory can be executed from any one of your directories. You may want to create a shell program that will move all the executable files in the current directory to your *bin* directory. The **test –x** command can be used to select the executable files from the list of files in the current directory. Review the **mv.file** program example of the **for** construct.

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

Include an **if test −x** statement in the **do...done** loop to move only those files that are executable.

If you name the shell program **mv.ex**, then the shell program will be as follows:

```
$ cat mv.ex<CR>
echo type in the directory path
read path
for file
  do
    if test -x $file
       then
          mv $file $path/$file
    fi
  done
$
```

The directory path will be the path from the current directory to the *bin* directory. However, if you use the value for the shell variable **HOME**, you will not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME/bin** gives the path to your *bin*.

```
$ cat mv.ex<CR>
for file
  do
    if test -x $file
       then
          mv $file $HOME/bin/$file
    fi
  done
$
```

To execute the command, use all the files in the current directory, * , as the positional parameters. The following screen executes the command from the current directory and then moves to the *bin* directory and lists the files in that directory. All the executable files should be there.

```
$ mv.ex *<CR>
$ cd; cd bin; ls<CR>
```

Shell Program Recap

mv.ex - move all executable files in the current directory to the *bin* directory

command	*arguments*
mv.ex	**all file names (*)**

Description:	Moves all the files with execute permission that are in the current directory to the *bin* directory
Remarks:	All executable files in the *bin* directory (or the directory indicated by the **PATH** variable) can be executed from any of your directories.

The Conditional Construct case..esac

The **case...esac** is a multiple choice construction that allows you to choose one of several patterns and then execute a list of commands for that pattern. The keyword **in** must begin the pattern statements with their command sequence. You must place a) after the last character of each pattern. The command sequence for each pattern is ended with **;;**. The **case** construction must be ended with **esac** (letters of case reversed). The general format for the **case** construction is:

```
case characters<CR>
in<CR>
    pattern1)<CR>
    command line 1<CR>
    .
    .
    .
    last command line<CR>
    ;;<CR>
    pattern2)<CR>
    command line 1<CR>
    .
    .
    .
    last command line<CR>
    ;;<CR>
    pattern3)<CR>
    command line 1<CR>
    .
    .
    .
    last command line<CR>
    ;;<CR>
    *)<CR>
    command 1<CR>
    .
    .
    .
    last command<CR>
    ;;<CR>
 esac<CR>
```

The **case** construction will try to match characters with the first pattern. If there is a match, the program will execute the command lines after the first pattern and up to the ;;.

If the first pattern is not matched, then the program will proceed to the second pattern. After a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**. The * used as a pattern at the end of the list of patterns allows you to give instructions if none of the patterns are matched. The * means any pattern, so it must be placed at the end of the pattern list if the other patterns are to be checked first.

If you have used the **vi** editor, you know you must assign a value to the **TERM** variable so that the shell knows what kind of terminal is going to display the editing window of **vi**. A good example of the **case** construction would be a program that will set up the shell variable **TERM** for you according to what type of terminal you are logged in on. If you log in on different types of terminals, the program **set.term** will be very handy for you.

set.term will ask you to type in the terminal type, then it will set **TERM** equal to the terminal code. You may want to glance back at the beginning of the **vi** tutorial for the explanation of those commands. The command lines are:

```
TERM=terminal code<CR>
export TERM<CR>
```

In this example of **set.term**, the person uses either a TELETYPE 4420, TELETYPE 5410, or a TELETYPE 5420.

The **set.term** program will first check if the value of **term** is 4420. If it is, then it will assign the value T4 to **TERM**, and exit the program. If it is not 4420, it will check for 5410 and then for 5420. It will execute the commands under the first pattern that it finds, and then go to the next command after the **esac** command.

At the end of the patterns for the TELETYPE terminals, the pattern * , meaning everything else, will warn you that you do not have a pattern for that terminal, and it will also allow you to leave the **case** construct.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
   in
   4420)
   TERM=T4
   ;;
   5410)
   TERM=T5
   ;;
   5420)
   TERM=T7
   ;;
   *)
   echo not a correct terminal type
   ;;
esac
export TERM
echo end of program
$
```

What would have happened if you had placed the * pattern first? The **set.term** program would never assign a value to TERM since it would always fit the first pattern *, which means everything.

When you read the section on modifying your login environment, you may want to put the **set.term** command in your *bin*, and add the command line

```
set.term<CR>
```

to your *.profile*.

Following is a quick recap of the **set.term** shell program.

Shell Program Recap

set.term - assign a value to TERM

command	*arguments*
set.term	interactive

Description:	Assigns a value to the shell variable TERM and then exports that value to other shell procedures.
Remarks:	This command asks for a specific terminal code to be used as a pattern for the **case** construction.

Unconditional Control Statement break

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done, fi**, or **esac** statement. If there are no commands after that statement, the program ends.

In the example for the program **set.term**, the **break** command could have been used instead of the **echo** command.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
in
4420)
TERM=T4
;;
5410)
TERM=T5
;;
5420)
TERM=T7
;;
*)
break
;;
esac
export TERM
echo end of program
$
```

As you do more shell programming, you may want to use two other unconditional commands, the **continue** command and the **exit** command. The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

Normally, a shell program terminates when the end of the file is reached. If you want the program to end at some other point, you can use the **exit** command. Both of these commands are explained in detail in the *Part 3*
of this manual.

Debugging Programs

Debugging is computer slang for finding and correcting errors in a program. There will be times when you will execute a shell program and nothing will happen. There is a "bug" in your program.

Your program may consist of several steps or several groups of commands. How do you discover which step is the culprit? There are two options to the **sh** command that will help you debug a program.

sh –v<CR>	Prints the shell input lines as they are read by the system.
sh –x<CR>	Prints commands and their arguments as they are executed.

To try out these two options, create a shell program that has an error in it. For example, type in the following list of commands in a file called **bug**.

```
$ cat bug<CR>
today= date
person=$1
mail $2
$person
When you log off come into my office please.
$today.
MLH
$
```

The mail message sent to Tom ($1) at login *tommy* ($2) should read as shown in the following screen.

```
$ mail<CR>
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1984
MLH
$
?
.
```

If you try to execute **bug**, you will have to press the BREAK key or the DEL key to end the program.

To debug this program, try **sh –v**, which will print the lines of the file as they are read by the system.

```
$ sh -v bug tom tommy<CR>
today= date
person=$1
mail $2
```

Notice that the output stops on the **mail** command. There is a problem with **mail**. The here document must be used to redirect input into **mail**.

Before you fix the **bug** program, try **sh -x**, which prints the commands and their arguments as they are read by the system.

```
$ sh -x bug tom tommy<CR>
+date
today=Thu  Apr 10  11:07:28  CST  1984
person=tom
+ mail tommy
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected **bug** program is as follows:

```
$ cat bug<CR>
today=`date`
person=$1
mail $2 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

The **tee** command is a helpful command to debug pipe lines. It places a copy of the output of a command into a file that you name, as well as piping it to another command. The general form of the **tee** command is:

command1 ¦ tee save.file ¦ command2<CR

save.file is the name of the file that will save the output of **command1** for you to study.

If you wanted to check on the output of the **grep** command in the following command line

who ¦ grep $1 ¦ cut -c1-9<CR>

you can use **tee** to copy the output of **grep** into a file to check after the program is done executing.

who ¦ grep $1 ¦ tee check ¦ cut -c1-9<CR>

The file *check* contains a copy of the output from the **grep** command.

```
$ who ¦ grep mlhmo ¦ tee check ¦ cut -c1-9<CR>
$ mlhmo
$ cat check<CR>
mlhmo    tty61    Apr 10    11:30
$
```

If you do a lot of shell programming, you will want to refer to *Part 3* of this manual and learn about command return codes and redirecting standard error.

Modifying Your Login Environment

What is a .profile?

When you log in, the shell first looks at a file in your login directory called the *.profile* (pronounced "dot profile"). The *.profile* is a shell program that issues commands to control your shell environment.

Since the *.profile* is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, and on other systems the system administrator will do this for you.

If you can edit the file yourself, you may want to be cautious the first few times and make a copy of your *.profile* in another file called *safe.profile*.

```
$ cp .profile safe.profile<CR>
$
```

You can add commands to your *.profile* just as you can add commands to any other shell program. You can also set some terminal options with the **stty** command, and you can set some shell variables.

Adding Commands to .profile

How do you add commands to your *.profile*? Try this pleasant example. The ICON/UXV system will allow you to start out your day with a message from your computer. Edit your *.profile* and add the following **echo** command to the last line of the file.

Type in: **echo** Good Morning! I am ready to work for you.

Write and quit the editor.

Whenever you make changes to your *.profile* and you want to initiate them in the current work session, you may type in a . and space before *.profile*. The shell will reinitialize your environment, that is, it will read and execute the commands in your *.profile*.

Now, experience communicating with your computer.

Type in: **. .profile<CR>**

The system should respond with:

```
Good Morning! I am ready to work for you
$
```

Setting Terminal Options

The **stty** command can make your shell environment more convenient for you. You can set the following options for **stty**.

stty −tabs

This option preserves tabs when you are printing. It expands the tab setting to eight spaces, which is the default. The number of spaces for each tab can be changed. Read the *ICON/UXV User Reference Manual* on **stty** for more details.

stty erase <^h>

This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character #. Usually this key is the BACK SPACE key.

stty echoe

If you have a terminal with a screen, this option erases characters from the screen as you erase them with the BACK SPACE key.

If you want to use these options for the **stty** command, you create those command lines in your *.profile* just as you would create them in one of your the shell programs. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding those four command lines to your *.profile*.

```
$ tail −4 .profile<CR>
echo Good Morning! I am ready to work for you
stty −tabs
stty erase <^h>
stty echoe
$
```

If you have not used the **tail** command before, the following is a brief recap of **tail**.

Command Recap

tail - display the last portion of a file

command	*options*	*arguments*
tail	-*n*	**file name**

Description:	Displays the last lines of a file.
Options:	Using the option you can specify number of lines *n*. The default (no options) is ten lines. There are other options, besides specifying -*n*. You can specify blocks (b) or characters (c) instead of lines.

Using Shell Variables

Several of the variables reserved by the shell are used in your *.profile*.

Let's take a quick look at four of these variables.

HOME

This variable gives the path for your login directory. Go to your login directory and type in **pwd<CR>**. What was the system response? Now type in **echo $HOME<CR>**. Was the system response the same as the response to **pwd**? **$HOME** is the default option for **cd**. If you do not specify a directory for **cd**, it will move you to **$HOME**.

PATH

This variable gives the system the search path for finding and executing commands.

If you want to see the current values for your **PATH** variable type in: **echo $PATH**.

```
$ echo $PATH<CR>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The **:** is a delimiter. Notice that for this **PATH** the system looks in */mylogin/bin*, for the command first, then into */bin*, then into */usr/bin*, and so on.

If you are working on a project with several other people, you may want to set up a group *bin*, a directory of special shell programs used only by your group. The directory would be found from the root directory. The path would be */group/bin*. How do you add this to your

PATH variable? Edit your *.profile*, and add *:/group/bin* to the end of your **PATH**.

```
PATH=:/mylogin/bin:/bin:/usr/lib:/group/bin<CR>
```

TERM

This variable tells the shell what kind of terminal you are working on. If you have done any editing with **vi** you know that you have to specify:

```
TERM=code<CR>
export TERM<CR>
```

Not only do you have to tell the shell what kind of terminal you are working on but you must **export** the variable. If you read *Part 8, Shell Commands and Programming* of this manual, you will learn why variables need to be exported.

If you do not want to specify the **TERM** variable each time you log in you can add those two command lines to your *.profile* and they will automatically be recognized each time you log into the ICON/UXV system. Or, if you log in on more than one type of terminal, you will want to add your **set.term** command to your *.profile.*

PS1

One of the delightful things about your *.profile* is that you can change your prompt. This one is fun to experiment with. Try the following example. If you wish to use several words, remember to quote the phrase. Also, if you use quotes you can add a carriage return to your prompt.

Type in: **PS1="Your wish is my command<CR>"**

Now your prompt sign looks like:

```
$ . .profile<CR>
Your wish is my command
```

The mundane **$** is gone forever, or until you delete the PS1 variable from your *.profile.*

Conclusion

This tutorial has given you the basics for creating some shell programs. If you have logged in and tried the examples and exercises as you read the tutorial, you can probably perform many of your day-to-day tasks with your new shell programs. Shell programming can be much more complex and perform more complicated tasks than shown in this brief tutorial. If you want to read further on shell commands and programming, read the *ICON/UXV User Reference Manual* on the **sh** command, and read *Part 3, System Shell Commands and Programming,* in this manual.

SHELL PROGRAMMING EXERCISES

2-1. Make the command line

```
       `               `
banner   date ¦ cut –c12-19 <CR>
```

into a shell program called **time.**

2-2. Make a shell program that will give only the date in a banner display. Be careful what you name the program!

2-3. Make a shell program that will send a note to several people on your system.

2-4. Redirect the date command without the time into a file.

2-5. Echo the phrase "Dear colleague" in the same file as the date command without erasing the date.

2-6. Using the above exercises, make a shell program that will send a memo with:

- Current date and the "Dear colleague" at the top of the memo,
- Body of a file that is the memo, and
- Closing statement

to the same people on your system as in Exercise 2-3.

2-7. How would you **read** variables into the **mv.file** program.

2-8. Use the **for** loop to move a list of files in the current directory to another directory.

How would you move all files to another directory?

Ingredients:
```
*
$*
mv $file newdirectory
```

2-9. How would you change the program **search**, to search through several files?

Hint:
```
for file
 in $*
```

2-10. Set the **stty** options for your environment.

2-11. Give yourself a new prompt that includes a carriage return. (Hint "   <CR>")

2-12. Check to see what $HOME, $TERM, and $PATH are set to in your environment.

ANSWERS TO EXERCISES

Command Language Exercises

1-1. The * at the beginning of a file name will refer to all files that end in that file name, including that file name.

```
$ls *t<CR>
cat
123t
new.t
t
$
```

1-2. **cat [0-9]*** would display the files:

```
1memo
100data
9
05name
```

echo * will list all the files in the current directory.

1-3. You can place **?** any place in a file name.

1-4. **ls [0-9]*** will list only those files that start with a number.

ls [a-m]* will list only those files that begin with letters "a" through "m".

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID for the job.

No, the **&** must be placed at the end of the command line.

1-6. The command line would be:

```
cd; pwd > junk; ls >> junk; ed trial<CR>
```

1-7. Change the -c option of the command line to read:

```
          \              \
banner    date | cut -c1-10 <CR>
```

Shell Programming Exercises

2-1.

```
$cat time<CR>           \
banner   date | cut -c12-19
$
$chmod u+x time<CR>
$ time<CR>
(banner display of the time 10:26)
$
```

2-2.

```
$cat mydate<CR>        \
banner   date | cut -c1-10
$
```

2-3.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail janice marylou bryan < $note
$
```

Or, if you wanted to use parameters for the logins.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail $* < $note
$
```

2-4. `date | cut -c1-10 > file1<CR>`

2-5. `echo Dear colleague >> file1<CR>`

2-6.

```
$cat send.memo<CR>
date | cut -c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1
$
```

2-7.

```
$cat mv.file<CR>
echo type in the directory path
read path
echo "type in file names, end with <^d>"
while
read file
    do
    for file
         in $file
        do
             mv $file $path/$file
        done
    done
echo all done
$
```

2-8.

```
$cat mv.file<CR>
echo Please type in directory path
read path
for file
     in $*
do
     mv $file $path/$file
done
$
```

The command line would then be:

```
$ mv.file *<CR>
$
```

2-9. See hint.

```
$ cat search<CR>
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done
```

2-10. Type the following lines into your *.profile*.

```
stty -tabs<CR>
stty erase <^h><CR>
stty echoe<CR>
```

2-11. Type the following command line into your *.profile*

```
PS1="Hello<CR>" <CR>
```

2-12.

```
$ echo $HOME<CR>
```

```
$ echo $TERM<CR>
```

```
$ echo $PATH<CR>
```

Chapter 8

COMMUNICATION TUTORIAL

PAGE

Chapter 8

COMMUNICATION TUTORIAL

INTRODUCTION

Sooner or later, you will want to use the ICON/UXV system to get in touch with other ICON/UXV system users. You may want to send a message to someone; the message may be one that must be read immediately. Perhaps you might need to send another user information from a file in your login.

Whatever the case, this chapter teaches you how to use the communication tools available to you on the ICON/UXV system. The chapter begins with a brief overview of just who you might want to communicate with on the ICON/UXV system. You learn how to send basic messages to users on your system and other ICON/UXV systems, and also how to deal with messages you receive. You also learn about commands that enable you to send files to other users.

The following list is a review of the text conventions mentioned in *Chapter 2* that are used in this chapter.

bold(Commands typed in exactly as shown.)

italic(ICON/UXV system prompts and responses.)

roman(Input other than commands.)

<>(Commands that are typed in, but are not reflected on the screen, are enclosed in angle brackets.)

COMMUNICATING ON THE ICON/UXV SYSTEM

You can use the ICON/UXV system to communicate with just about anyone else who uses the ICON/UXV system. This means that your terminal does more than serve as a work station--it becomes your personal message-handling center as well, with the electronic equivalent of transmission, routing, and storage facilities.

Who would you want to communicate with over the ICON/UXV system? Here are some examples to consider:

- The person in your office who needs to know about a department meeting tomorrow,
- Other users on your ICON/UXV system who should see a posted message concerning their use of the system after office hours,
- The supervisor who wants a copy of your last two reports by 2:30 this afternoon,
- The supervisor who wants to review the memo you are presently working on as soon as you have finished it,
- A person working with you on the ICON/UXV system to modify several files you both have in common; you need to be in touch from time to time, but the phones are being used as links from your terminals to the computer and you would rather not shout down the halls, and
- A coordinator who wants your daily operations records (all in very large files), but does not want to have to wade through them all at once when he receives them on the terminal.

As you can see, you can keep in touch with any number of people for any number of reasons through the ICON/UXV system. The remainder of this chapter shows you how to use the various communication tools provided by the ICON/UXV system to reach these people.

HOW CAN YOU COMMUNICATE?

The ICON/UXV system offers several commands for user-to-user communication. This chapter explains the most important commands to know and suggests how to select the one to use in a given situation. The basic choice is between sending (or receiving) a message and sending (or receiving) a file.

To expand on one of the previous examples, suppose you are working at your terminal and you remember that you are giving a presentation at an officewide meeting tomorrow. You want to remind someone in your office about the presentation, but you do not want to take the time for a phone call or a walk to the other person's office. What can you do?

If the other person has a login on your ICON/UXV system, you can use the **mail** command to send a brief message. When the recipient of your message finishes whatever task he or she is using the ICON/UXV system for, a notice is posted that there is mail waiting to be read. The recipient can then read your message and send a reply back to your login.

To take another example, what if you need to send other people copies of things you already have on file--memos, reports, saved messages, documents, and the like? You can send such files using the **mail** command; however, this may not be the best way to send long files. For sending files over a page in length, you should use the **uuto** command. This command sends the file to a public directory on the recipient's system instead of sending it straight to the recipient's login. The recipient can then deal with it at his or her own leisure.

These are the important communication tools available to you. (Two other tools, the **uucp** and **mailx** commands, are discussed briefly at the end of the chapter.) Now that you have a general idea of how to communicate in the ICON/UXV system, let's move on to the specifics.

SENDING AND RECEIVING MESSAGES *(mail)*

The **mail** command works in two ways--it lets you send messages to other ICON/UXV system users, and it lets you read messages sent to you. This section deals first with sending messages, both to users on your ICON/UXV system and to users on other ICON/UXV systems that can communicate with yours.

Sending Mail

It is easy to send mail to another user. The basic command line format for sending mail is

```
mail login<CR>
```

where *login* is the recipient's login name on the ICON/UXV system. This login name can be either of the following:

- A login name if the recipient is on your system, or
- A system name and login name if the recipient is on a system that can communicate with yours.

For the moment, assume that the recipient is on your system (known as the local system); we will deal with sending mail to users on other systems (known as remote systems) a little later.

Basics of Sending Mail

Since the recipient is on your system, you type the **mail** command as follows at the system prompt ($):

```
mail login<CR>
text
.
```

where *login* is the recipient's login name. Then you type in the text of the letter, as many lines as you need. When your message is complete, you send the message on its way by typing a dot (.) at the beginning of a new line.

The resulting message looks like this:

```
$ mail login<CR>
After you enter the command line,<CR>
type in as many lines of text as you need<CR>
to get the message across.<CR>
When you're done,<CR>
type in a control-d or a dot<CR>
on a line by itself, as shown on the next line.<CR>
.<CR>
$
```

The system prompt returns to notify you that your message has been queued (placed in line) and will be sent.

Sending Mail to One Person

Let's look at a sample situation. You have to notify another person in your office of a meeting later this afternoon, but he is not in and you have to leave your office. He has a login on your ICON/UXV system with the login name *tommy*, so you can leave a message for him to read the next time he logs into the system:

```
$ mail tommy<CR>
Tom,<CR>
There's a meeting of the review committee<CR>
at 3:00 this afternoon. D.F. wants your<CR>
comments and an idea of how long you think<CR>
the project will take to complete.<CR>
B.K.<CR>
.
$
```

When Tom logs in at his terminal (or while he is already logged in), he receives a message that tells him he has mail waiting:

```
you have mail
```

To see how *tommy* can read his mail, see the section titled *Receiving Mail.*

You can practice using the **mail** command by sending mail to yourself. This may sound strange at first, but it is the easiest way to practice sending messages. Simply type in the **mail** command and your own login name, then write a short message to yourself. When you type in the dot, the mail will be sent to your login and you will receive the notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. Suppose your login name is *rover*; you are ready to log off of the system for the day and you want to leave a reminder to call someone first thing the next morning. You might enter the following:

```
$ mail rover<CR>
Remember to call Accounting and find out<CR>
why they haven't returned my 1984 figures!<CR>
.
$
```

When you log in the next day, you will get a notice of messages awaiting you. Reading your mail then brings up the reminder message (and any other messages you may have received).

Sending Mail to Several People Simultaneously

If you need to send the same message to more than one person, simply place their login names after the **mail** command on the command line, with a space between each one, in the following format:

mail login1 login2 ... <CR>

where *login1*, *login2*, and ... are the different login names. You can mail messages to as many logins as you wish.

For example, if you send a notice about the department softball game to team members with login names *tommy, switch, wombat,* and *dave,* it might look like this:

```
$ mail tommy switch wombat dave<CR>
Diamond cutters,<CR>
The game is on for tonight at diamond three.<CR>
Don't forget your gloves!<CR>
Your Manager<CR>
.<CR>
$
```

To provide you with a quick summary of what you can expect when using the **mail** command to send messages, a recap of how to use it follows.

Command Recap

mail - sends a message to another user's login

command	*options*	*arguments*
mail	none	**login**

Description:	**mail** followed by one or more login names, sends the message typed on the lines following the command line to the specified login(s).
Remarks:	Typing a dot at the beginning of a new line sends the message.

Sending Mail to Remote Systems *(uname, uuname)*

We have assumed to this point that you are sending messages to recipients on your (local) ICON/UXV system. You may have occasion, however, to send messages to recipients on other (remote) ICON/UXV systems. For example, your office may have three separate systems, each in a different part of the building. Or perhaps you may have offices in several different locations, each with its own system.

How do you send mail to someone on a remote system? The ICON/UXV system you are on must be able to communicate with a remote ICON/UXV system before mail can be sent between the two. So, if you plan to send a mail message to someone on a remote system, you need to do a little legwork to find out the following information:

- Recipient's login name,
- Name of the remote system, and
- If your system and the remote system can communicate.

Two commands are available to help you answer these questions--the **uname** and **uuname** commands.

You can get the login name and the remote system name from the recipient. If it happens that the recipient does not remember the system name, have him or her log into the system and type the following at the system prompt:

```
uname -n<CR>
```

The **uname -n** command responds with the name of the system you are logged into. For example, if you are logged into a system named *sys10* and you type in **uname -n**, your screen should look like this:

```
$ uname -n<CR>
sys10
$
```

Once you know the remote system name, the **uuname** command helps you find out if your system can communicate with the remote system. At the prompt, type:

uuname<CR>

This generates a list containing the names of remote systems with which your system can communicate. If the recipient's system is in that list, then you can send messages there by **mail**.

The **uuname** command may respond with a large list of names if your system can communicate with many other systems. To avoid having that long list scroll quickly up your screen, use the pipe and **grep** command in conjunction with **uuname**. At the prompt, type:

uuname ¦ grep system<CR>

where *system* is the recipient's system name. This generates the same list, then searches for and prints only the specified system name if it is found in the list.

For example, if you want to find out whether a system called *sys10* can communicate with your system, type:

```
$ uuname ¦ grep sys10<CR>
```

If this is the case, the system name is printed in response:

```
$ uuname ¦ grep sys10<CR>
sys10
$
```

If you get only the system prompt back, then the two systems cannot communicate:

```
$ uuname ¦ grep sys10<CR>
$
```

Once you determine that you can send messages to a login on a remote system, your **mail** command line is slightly different than it is for sending mail to someone on your local system. The command line format for remote systems is:

mail system!login<CR>

where *system* is the remote system name and *login* is the recipient's login name. The two parts of the address are separated by an exclamation point (!).

Now that you have all the parts, let's put them together into an example. Assume that you have a message for someone on a different system in another part of your office. You know from the recipient her login name, *sarah*, and her system name, *sys10*. To find out if her system can communicate with yours, use the **uuname** command:

```
$ uuname ¦ grep sys10<CR>
sys10
$
```

The system response tells you that your system is indeed networked to system *sys10*. Now all you have to do is send the message, using the expanded address format given previously:

```
$ mail sys10!sarah<CR>
Sarah,<CR>
The final counts for the writing seminar<CR>
are as follows:<CR>
<CR>
Our department - 18<CR>
Your department - 20<CR>
<CR>
Tom<CR>
.<CR>
$
```

Following is a quick summary of the two commands introduced in this section and what you can expect them to do.

Command Recap

uname - displays the system name

command	*options*	*arguments*
uname	**-n** and others*	none

Description: **uname -n** displays the name of the system on which your login resides.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Command Recap

uuname - displays a list of networked systems

command	*options*	*arguments*
uuname	none	none

Description: **uuname** displays a list of remote systems that can communicate with your system.

Receiving Mail

Once you learn to send messages, you may be anxious to read what others are sending your way. As stated earlier, the **mail** command also allows you to read messages sent by other ICON/UXV users.

After logging in, you may receive the following message at your terminal:

```
you have mail
```

This tells you that one or more messages are being held for you in a ICON/UXV directory named *usr/mail*, usually referred to as the the *mailbox*. Entering the **mail** command by itself allows you to read these messages.

To read your mail, type the **mail** command by itself at the system prompt:

mail<CR>

This displays the waiting messages at your terminal, one message at a time, with the most recently received message displayed first. In other words, as you read your messages, you go from the "newest" message to the "oldest" message.

A typical **mail** message looks like this:

```
$ mail
From tommy Mon May 21 15:33 CST 1984
B.K.
Looks like the meeting has been canceled.
Do you still want the technical review material?
Tom

?
```

The first line, called the *header*, displays information about a particular message—the login name of the sender, the date sent, and the time sent. The following lines (except for the last line) are the body of the message.

Notice the question mark (?) on the last line of the message. After displaying each message, the **mail** command displays a ? and a space, and waits for a response from you before going on to the next message. There are several responses; we will look at the most common responses and what they do.

After reading a message, you may want to delete it. To do so, type a **d** after the question mark.

```
? d<CR>
```

This response deletes the message from the *mailbox* and displays the next message waiting in the *mailbox* (if there is one). If there are no other messages, the system prompt returns to indicate that you've finished reading your messages.

If you would rather display the next message without deleting the message being displayed, type a carriage return after the question mark.

```
? <CR>
```

The current message goes back into the *mailbox* and the next message is displayed. If there are no more messages in the *mailbox*, the system prompt returns.

You may want to save the message for later reference. To do so, type an **s** after the question mark:

```
? s<CR>
```

This response saves the mail message by default in a file called *mbox* in your login directory. If you would rather save the message in another file, follow the **s** response with a file name or with a path name ending in a file name.

For example, to save the message in a file called *mailsave* in your current directory, enter the following response after the question mark:

```
? s mailsave<CR>
```

If you use the **ls** command to list the contents of this directory, you will find the file *mailsave*.

You can also save the message in a file under another of your directories. If you have a mail message about a particular project or piece of work that you keep in a certain directory, you may want to save that message in the same directory. Let's say you have such a directory, named *project1*, under your login directory. If a mail message comes in that you want to place in directory *project1*, under a file named *memo*, enter the following response after the question mark:

```
? s project1/memo<CR>
```

If you use the **cd** command to change directories from your login directory to *project1* and then use the **ls** command, you will find that the file *memo* is now listed. (You can use other, more complete path names as well; refer to *Chapter 3* for instruction on using path names.)

If you want to quit reading messages, enter the following response after the question mark:

```
? q<CR>
```

Any messages that you have left unread are put back in the *mailbox* until the next time you use the **mail** command.

If a long message is being displayed at your terminal, you can interrupt it by pressing the BREAK key. This stops the message display, prints the ?, and waits for your response.

Other responses are available; these are listed in the *ICON/UXV User Reference Manual.* The following command recap summarizes what you can expect when using the **mail** command to read messages.

Command Recap

mail - reads messages sent to your login

command	*options*	*arguments*
mail	available*	none

Description:	**mail** entered by itself displays any messages waiting in the system file *usr/mail* (the mailbox).
Remarks:	The question mark (?) at the end of a message indicates that a response is expected. A full list of responses is given in the *ICON/UXV User Reference Manual.*

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

SENDING AND RECEIVING FILES

In several examples cited so far in this chapter, the need to send files from your ICON/UXV system login to another UNIX system user has come up. Memos, reports, stories, baseball scores--there are numerous items that you can keep in your files. What do you do to send copies of those files to other UNIX system users?

Sending Small Files *(mail)*

The **mail** command uses the redirection symbol < to take its input from a specified file instead of from the keyboard. (For more detailed information on the use of redirection symbols, see *Chapter 7.*) The general format is as follows:

```
mail login < filename<CR>
```

where *login* is the recipient's login name and *filename* is the name of the file containing the information to be sent.

For example, assume you keep a standard meeting notice in a file named *meetnote.* If you want to send the letter to the owner of login *sarah* using the **mail** command, type the following at the prompt:

```
$ mail sarah < meetnote<CR>
$
```

The system prompt returns to let you know that the contents of *meetnote* have been sent. When *sarah* types in the **mail** command to read her messages, she will receive the standard meeting notice.

Likewise, if you want to send the same file to several users on your system, type in the **mail** command followed by the login names of the users, and then follow these with the < file redirection operator and the file name. It might look like this:

```
$ mail sarah tommy dingo wombat < meetnote<CR>
$
```

The system prompt tells you that the messages have been sent.

If the recipient for your file is on a remote system that can communicate with yours, simply redirect the file with the < operator:

```
mail system!login < filename<CR>
```

For example:

```
$ mail sys10!wombat < meetnote<CR>
$
```

Again, the system prompt notifies you that the message has been queued for sending.

Sending Large Files *(uuto)*

When you need to send large files, you should use the **uuto** command. This command can be used to send files to both local and remote systems. When the files arrive at their destination, the recipient receives a mail message announcing its arrival.

The basic format for the **uuto** command is

uuto filename system!login<CR>

where *filename* is the name of the file to be sent, *system* is the recipient's system, and *login* is the recipient's login name. The *filename* may be the name of a file or a path name ending in a specific file.

If you send a file to someone on your local system, you may omit the system name and use the following format:

uuto filename login<CR>

Have You Got Permission?

Before you actually send a file with the **uuto** command, you need to find out whether or not the file is transferable. To do that, you need to check the file's permissions. If they are not correct, you must use the **chmod** command to change them. (Permissions and the **chmod** command are covered in detail in *Chapter 3.*)

There are two permission criteria that must be met before a file can be transferred using **uuto**:

- The file to be transferred must have read permission (**r**) for *others*, and
- The directory that contains the file must have read (**r**) and execute (**x**) permission for *others.*

This may sound confusing, but an example should clarify the matter.

Assume that you have a file named *chicken*, under a directory named *soup*, that you want to send to another user with the **uuto** command. First you check the permissions on *soup*, which is under your login directory:

```
$ ls -l<CR>
total 35
-rwxr-xr-x    1       reader  group15598Mar 313:00  memos
drwxr--r--    2       reader  group1 477Mar 109:08  lists
drwxr-xr-x    2       reader  group1  45Feb 910:43  soup
$
```

Checking the line that contains the information for directory *soup* shows that it has read (**r**) and execute (**x**) permissions in all three groups; no changes have to be made. Now you use the **cd** command to change from your login directory to ***soup*** and then check the permissions on the file ***chicken***:

```
$ ls -l chicken<CR>
total 4
-rw------- 1 reader group1 3101 Mar 1 18:22 chicken
$
```

The output informs you that the file *chicken* has read permission for you, but not for the rest of the system. To add those read permissions, you use the **chmod** command:

```
$ chmod go+r chicken<CR>
```

This adds read permissions to the rest of the system--*group* (**g**) and *others* (**o**)--without changing the previous permissions. Now, checking again with the **ls –l** command reveals the following:

```
$ ls -l chicken<CR>
-rw-r--r-- 1 reader group1 3101 Mar 1 18:22 chicken
$
```

This confirms that the file is now transferable using the **uuto** command. After you send copies of the file, you can reverse the procedure and replace the previous permissions.

Sending a File *(uuto –m, uustat)*

Now that you know how to determine if a file is transferable, let's take an example and see how the whole thing works.

The process of sending a file by **uuto** is referred to as a *job*. When you enter a **uuto** command, your job is not sent immediately. First it is stored in a queue (a waiting line of jobs) and assigned a job number. When the job's number comes up, it is transmitted to the remote system and placed in a public directory there. The recipient is notified by **mail** message and must use the **uupick** command to retrieve the file (this command is discussed later in the chapter).

For the following discussions, assume this information:

wombat	Your login name.
sys10	Your system name.
marie	Recipient's login name.
sys20	Recipient's system name.
money	File to be sent.

Also assume that the two systems can communicate with each other.

To send the file *money* to login *marie* on system *sys20*, enter the following:

```
$ uuto money sys20!marie<CR>
$
```

The system prompt returns, notifying you that the file has been sent to the job queue. The job is now out of your hands; all you can do is wait for confirmation that the job reached its destination.

How do you know when the job has been sent? The easiest method is to alter the **uuto** command line by adding a **−m** option, like so:

```
$ uuto −m money sys20!marie<CR>
$
```

This option sends a **mail** message back to you when the job has reached the recipient's system. The message may look something like this:

```
$ mail<CR>
From uucp Tue Apr 3 09:45 EST 1984
file /sys10/wombat/money, system sys10
copy succeeded

?
```

If you would rather check from time to time while you are working on the system, you can use the **uustat** command. This command keeps track of all the **uuto** jobs you submit and gives you their status. For example,

```
$ uustat<CR>
1145 wombat sys20 10/05-09:31 10/05-09:33 JOB IS QUEUED
$
```

The elements of this sample status message are as follows:

- *1145* is the job number associated with sending file *money* to *marie* on *sys20.*
- *wombat* is your login name.
- *sys20* is the recipient's system.
- *10/05-09:31* is the date and time the job was queued.

- *10/05-09:33* is the date and time of this particular **uustat** message.
- The final part is the status of the job—in this case indicating that the job has been queued, but has not yet been sent.

If you are interested in just one **uuto** job, you can use the **–j** option and the job number when requesting job status:

uustat –jjobnumber<CR>

In the example, let's say you enter the **uustat** command with the **–j** option (for job 1145) until you receive the following response:

```
$ uustat -j1145<CR>
1145 wombat sys20 10/05-09:31 10/05-09:37 COPYFINISHED,JOB DELETED
$
```

This status message indicates that the job was sent and has been deleted from the job queue--in other words, it has reached the public directory of the recipient's system. There are other status messages and options for the **uustat** command which are described in the *ICON/UXV User Reference Manual.*

That is all there is to sending files. You can practice simply by sending another UNIX system user a file. You should practice with a test file until you have the procedures down pat.

The following command recaps give a summary of the **uuto** and **uustat** commands for your convenience.

Command Recap

uuto - sends files to another login

command	*options*	*arguments*
uuto	**—m** and others*	**file system!login**

Description: **uuto** sends the specified file to the public directory of the specified system. The owner of the login is notified by **mail** that a file has arrived.

Remarks: Files to be sent must have read permission for *others*; the directory above the file must have read and execute permissions for *others*.

The **—m** option notifies you by mail when the file arrives at its destination.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Command Recap

uustat - checks job status of a **uuto** job

command	*options*	*arguments*
uustat	**-j** and others*	none

Description: **uustat** checks on the status of all **uuto** jobs sent from your login and displays the results.

Remarks: The **-j** option, followed by a specific job number, displays the status of only the specified job.

* See the *ICON/UXV User Reference Manual* for all available options and an explanation of their capabilities.

Receiving Files *(uupick)*

When a file sent by **uuto** shows up in the public directory on your ICON/UXV system, you receive a **mail** message telling you that the file has arrived and where you can find it. To continue our previous example, let's see what the owner of login *marie* receives when she types in the **mail** command, not long after you (login *wombat*) have sent her the file *money*:

```
$ mail
From uucp Mon May 14 09:22 EST 1984
/usr/spool/uucppublic/receive/marie/sys10//money from sys10!wombat arrived
$
```

The message contains the following pieces of information:

- The first line tells you when the file arrived at its destination.
- The second line up to the two slashes (//) gives you the path name to the part of the public directory where the file has been stored.
- The second line after the two slashes tells you the name of the file and who sent it.

Once you have disposed of the **mail** message, you can use the **uupick** command to store the file where you want it. Type

uupick<CR>

at the system prompt. The command searches the public directory for any files sent to you. If it finds any, it prompts you with a **?** to do something with the file (much like the **mail** command).

Continuing with our previous example, if the owner of login *marie* enters the **uupick** command, she receives the following response:

```
$ uupick<CR>
from system sys10: file money
?
```

After the question mark (?), the command goes to the next line and waits for your response. There are several available responses; we will look at the most common responses and what they do.

The first thing you should do is move the file from the public directory and place it in your login directory so you can see what it is. To do so, type an **m** after the question mark.

```
?
m<CR>
$
```

This response moves the file into your current directory. If you wish to put it in some other directory instead, follow the **m** response with the directory name:

```
?

m directory<CR>
```

If there are other files waiting to be moved, the next one is displayed, followed by the question mark. If not, the prompt returns.

If you would rather display the next message without doing anything to the current file, press the carriage return key after the question mark.

```
?
<CR>
```

The current file remains in the public directory until you next use the **uupick** command. If there are no more messages, the system prompt returns.

If you already know that you do not want to save the file, you can delete it by typing in a **d** after the question mark:

```
?
d<CR>
```

This response deletes the current file from the public directory and displays the next message (if there is one). If there are no additional messages about waiting files, the prompt returns.

Finally, if you want to stop the **uupick** command, type a **q** after the question mark:

```
?
q<CR>
```

Any unmoved or undeleted files will wait in the public directory until the next time you use the **uupick** command.

Other available responses are listed in the *ICON/UXV User Reference Manual.* The following command recap summarizes what you can expect from the **uupick** command.

Command Recap

uupick - searches for files sent by **uuto**

command	*options*	*options*
uupick	none	none

Description:	**uupick** searches the public directory of your system for files sent by **uuto**. If any are found, the command displays information about the file and awaits a response.
Remarks:	The question mark (?) at the end of the message indicates that a response is expected. The full list of responses is given in the *ICON/UXV User Reference Manual.*

ADVANCED MESSAGE AND FILE HANDLING *(uucp, mailx)*

Once you master the **mail** and **uuto/uupick** commands, you may decide that you want commands that are more flexible or efficient. If so, you should try the **mailx** and **uucp** commands.

The **uucp** command enables you to send a copy of a file directly to another user's login directory, instead of to the public directory on that user's system. In some cases, you can even copy directly from files in another login and place the copy in your login directory. The **uucp** command also enables you to rename a file when it reaches its destination.

There are a number of considerations to deal with when using **uucp**, such as file permissions and system security procedures. The **uucp** system is more complex and requires more experience to use than **uuto** and **uupick**.

If you want an electronic mail facility with more features, there is the **mailx** command. This command is an interactive message-handling system that gives you, among other things, the following:

- The ability to use either the **ed** or **vi** text editor for use on incoming *and* outgoing messages,

- A list of waiting messages from which the user can decide which messages to deal with and in what order,
- Several options for saving files, and
- Commands for replying to specific messages and sending copies to other users (both of incoming and outgoing messages).

As you might gather, these two commands are complex and are not recommended for the beginning user. Because of this, we do not cover the uses of **uucp** or **mailx** in this guide. However, these commands are mentioned here because they may be available in your ICON/UXV package and are useful commands to know about.

Once you are thoroughly familiar with the standard tools for user communication, you may want to experiment with the **uucp** and **mailx** commands. Refer to the *ICON/UXV User Reference Manual* for more information on using these commands.

Chapter 9

USING SHELL COMMANDS

PAGE

Chapter 9

USING SHELL COMMANDS

INTRODUCTION

This chapter provides information to enhance uses of the **shell**. Most information should be useful to both the programmer and nonprogrammer alike. Some information may be of more use to the more advanced user. It is assumed that the user has been introduced to the ICON/UXV system and understands such basics as how to log in, set the terminal baud rate, etc.

EXECUTING SIMPLE SHELL COMMANDS

A simple **shell** command consists of the command name possibly followed by some arguments such as

```
cmd arg1 arg2 arg3 ...
```

where **cmd** is the command name consisting of a sequence of letters, digits, or underscores beginning with a letter or underscore. For example, the **shell** command

```
ls
```

prints a list of files in the current directory.

INPUT/OUTPUT REDIRECTION

Most commands produce output to a terminal. Output can be redirected to a file in two different ways. First, standard output may be redirected to a file by the notation ">", thus

```
ls -l > tempfile
```

causes the **shell** to redirect the output of the command **ls** to be put in *tempfile*. If there is no file *tempfile*, one is created by the **shell**. Any previous contents of *tempfile* are destroyed.

Standard output may be appended to the end of a file by the notation ">>" thus

```
ls -l >> tempfile
```

causes the **shell** to append the output of the command **ls** to the end of the contents of *tempfile*. If *tempfile* does not already exist, it is created.

Although input is normally from a terminal, it can also be redirected by the "<" notation. Thus

```
wc < tempfile
```

would send the contents of *tempfile* to the **wc** command which would give a character, word, and line count of *tempfile*. Another modification of input is possible with the "<<" notation. The form

```
cmd <<word
```

would send standard input to the specified command until a line the same as *word* is input. As an example

```
sort <<finished
```

would send all the standard input to **sort** until **finished** is input. Then the input would be sorted and output to the terminal. If the notation "<<–" is used, then all leading tabs would be stripped. As an example, the following is entered at the terminal (note that the primary system prompt $ and the secondary system prompt > provided by the system are shown in this example)

```
$sort <<end
>no one does anything about it
>everyone talks about the weather but
>end
```

and the following would be returned

```
everyone talks about the weather but no one does anything about it
```

PIPELINES AND FILTERS

The standard output of one command may be connected to the standard input of another by using the pipe (¦) operator between commands as in

```
ls –l ¦ wc
```

A sequence of one or more commands connected in this way constitutes a pipeline, and the overall effect is the same as

```
ls —l > file; wc < file
```

except no file is used. Instead the two processes are connected together by a pipe [see **pipe**(2)] and are run in parallel. Each command is run as a separate process.

Pipes allow one to execute several commands sequentially from left to right with the standard output from each command becoming the standard input of the next command. This prevents creating temporary files and is faster than not using pipes. Pipes are unidirectional. 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**(1), selects from its input those lines that contain some specified string. For example,

```
ls ¦ grep old
```

prints those lines that contain the string "old". Another filter is the **sort**(1) command that gives alphabetical listings.

PERMISSION MODES

All ICON/UXV files have three independent attributes (often called "permission"), read, write, and execute (rwx). These three permissions are assigned to three different levels of users. The first level is the owner level. Normally, the creator of the file is the owner. This ownership can be changed with the **chown**(1) command. The second level is the group level. The third level is the others level. The permission for each level must be set to allow reading, writing, or executing a file.

The **ls** command will display among other things the permissions for a file when used as follows

```
ls —l filename
```

The general format of the permissions is

```
-rwxrwxrwx
```

where the first character will be a dash if it is an ordinary file. The second, third and fourth characters (the first **rwx**) indicate the permission modes for the owner. The fifth, sixth, and seventh characters (the second **rwx**) indicate the permission modes of the group. And the eighth, ninth, and tenth characters (the last **rwx**) indicate the permission modes of others. A dash in any permission mode position indicates that the mode is not allowed.

For example, the input

```
ls –l wg
```

displays the permissions of *wg* as follows

```
-rwxr-x---   1 abc    ICON/UXV        66 May  4 09:25 wg
```

In this case, the owner has read (r), write (w), and execute (x) permission, the group has read and execute permission, and all others are denied (-) permission to *wg*.

The **chmod**(1) command is used by the owner to change the permission modes of a file. To change the permissions of *wg* so that everyone could execute the procedure, enter the following command

```
chmod 751 wg
```

which would result in a permission mode of **rwxr-x--x**. The **7** assigns the owner read, write, and execute permission [4 (read) + 2 (write) + 1 (execute) = 7]. The **5** assigns the group read and execute permission [4 (read) + 1 (execute) = 5]. The **1** assigns others execute permission.

The **chmod** command could also be entered as

```
chmod +x wg
```

which would add execute permission for owner, group, and all others.

FILE NAME GENERATION

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**(1) all file names in the current directory that end in **.c**. The character "*" is a pattern that will match 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 character enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
ls -l [a-z]*
```

matches all names in the current directory beginning with letters *a* through *z*. The input

```
ls -l /usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. This mechanism is useful both to save typing and to select names according to some pattern.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

prints all file names in the current directory not beginning with ".". The input

```
echo .*
```

prints all those file names that begin with ".". This avoids inadvertently matching the names "." and ".." that mean "the current directory" and "the parent directory," respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

QUOTING

Characters that have a special meaning to the **shell**, such as

```
< > * ? | & $ ; \ " ‘ ’ [ ]
```

are called metacharacters.

The **shell** can be inhibited from interpreting and acting upon the special meaning assigned metacharacters by preceding them with a backslash (\). Any character preceded by a \ loses its special meaning. For example

```
echo *
```

prints all the file names in the current directory. To echo an asterisk , enter

```
echo \*
```

The backslash turns off any special meaning of a metacharacter.

To allow long strings to be continued over more than one line, the sequence **\newline** (or RETURN) is ignored. The \ 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 may be quoted by enclosing the string between single quotes. All characters enclosed between a pair of single quote marks are quoted except for a single quote. For example,

```
echo xx'****'xx
```

will print

```
xx****xx
```

The quoted string may not contain a single quote but may contain new lines that are preserved. This quoting mechanism is the simplest and is recommended for casual use.

EXECUTING COMMANDS IN THE BACKGROUND

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. Executing commands in the background enables the terminal to be used for other tasks. Adding an ampersand (&) at the end of a command line before the RETURN starts the execution of a command and immediately returns to the **shell** command level. 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. This means the system will respond with a process number followed by the primary **shell** prompt.

Determining Completion of Background Commands

When a command is executed in the background, a prompt is not received when the command completes execution. The only way to see that the command is either in process or complete is to request process status. The status of all active processes assigned to a user can be reported as follows

```
ps —u ulist
```

where "ulist" is the login name. If the process number and associated command name are output by the **ps** command, then the command is running in the background. If the process number and associated command name are not output by the **ps** command, then the command has finished executing.

Terminating Background Commands

Once a command starts in the background, it will run until it is finished or is stopped. The BREAK, RUBOUT, DELETE, or other keys will not stop a command running in the background. Instead, the process must be "killed" with the **kill**(1) command as follows

```
kill PID
```

where "PID" is the process identification number. The **shell** variable $! contains the "PID" of the last process run in the background and can be obtained as follows

```
echo $!
```

All nonessential background processes can be stopped by executing the following command

```
kill 0
```

Some processes can ignore the software termination signal. To stop these processes, enter the following

```
kill –9 PID
```

A process running in the background is automatically killed when the user logs out. The **nohup**(1) command can be used to continue the process after logging off or hanging up. For example,

```
nohup nroff text &
```

would continue the formatting of the file *text* using the **nroff**(1) formatter even if one logged off or the telephone line to the computer went down. The system responds with the lines

```
28096
$ Sending output to nohup.out
```

The **28096** is the process id number. A file *nohup.out* is created by the **nohup** command, and all output of the process is directed to this file. To redirect the output to a particular file, use the redirect command as follows

```
nohup nroff text & > formatted
```

to direct the output to the file *formatted.*

SHELL VARIABLES

A variable is a name representing a string value. (Loosely defined, a string is a combination of one or more alphanumeric characters or symbols.) Variables that are normally set on a command line are called parameters. There are two types of parameters in the **shell**— positional and keyword.

Positional Parameters

When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters.* The **shell** assigns the positional parameters as follows

```
${0} ${1} ${2} ${3} ... ${9}
```

Since the general form of a simple command is

```
cmd arg1 arg2 arg3 ...
```

then the values of the positional parameters are

```
cmd  arg1 arg2 arg3 ... arg9
${0} ${1} ${2} ${3} ... ${9}
```

For instance, if the following command is entered

```
cmd temp1 temp2 temp3
```

then the positional parameter **${1}** would have the value **temp1**. Notice that the command procedure name is always assigned to **${0}**.

The positional parameters are used often in **shell** programs. If a **shell** program, **wg**, contained

```
who | grep $1
```

then the call to run the program

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

The variable $* is a special **shell** parameter used to substitute for all positional parameters except $0. Certain other similar variables are used by the **shell**. The following are set by the **shell**:

$?	The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.
$#	The number of positional parameters in decimal.

$$	The process number of this **shell** in decimal. Since process numbers are different from all other existing processes, this string is frequently used to generate temporary file names. For example,
	`ps a >/tmp/ps$$`
	`...`
	`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**.

Keyword Parameters

The **shell** uses certain variables known as keyword parameters for specific purposes. The following variables are discussed in this portion of the document:

HOME
PATH
CDPATH
MAIL
MAILCHECK
MAILPATH
PS1
PS2
IFS
SHACCT
SHELL.

HOME

The variable *HOME* is used by the **shell** as the default value for the **cd**(1) command. Entering

```
cd
```

is equivalent to entering

```
cd $HOME
```

where the value of *HOME* is substituted by the **shell**. If *$HOME=/d3/abc/def*, then each of the above two entries would be equivalent to

```
cd /d3/abc/def
```

Normally, *HOME* is initialized by **login**(1) to the login directory. The value of *HOME* can be changed to */d3/abc/ghi* by entering the following

```
HOME=/d3/abc/ghi
```

No spaces are permitted. The change of the variable will have no effect unless the value is **exported** [see **export** in Chapter 3 under "Special Commands" and in **sh**(1)]. All variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its environment. To change the environment to a new variable setting, the following must be entered

```
export variable-name
```

For instance, if **HOME** has been modified, then the command

```
export HOME
```

will cause the environment to be modified accordingly. The variable **HOME** need be exported only once. At login the next time, the original variable settings will be reestablished. A change to the *.profile* would modify the environment for each new login.

PATH

The variable *PATH* is used by the **shell** to specify the directories to be searched to find commands. Each directory entry in the *PATH* variable is separated by a colon (:). Several directories can be specified in the *PATH* variable but each directory before the command is found consumes processor time. Obviously, the directories that contain the most often used commands should be specified first to reduce searching time. The following is the default *PATH* value

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

Since no value precedes the first :, then the current directory is the first directory searched. Then directory */bin* is searched followed by */usr/bin*. To change the *PATH* variable, simply enter *PATH=* followed by the directories to be searched. Each directory should be separated by a colon. As when changing all variables, no spaces are allowed before or after the =.

CDPATH

The variable *CDPATH* specifies where the **shell** is to look when it is searching for the argument of the **cd** command if that argument is not null and does not begin with ../, ./, or /. For example, if the **CDPATH** variable were

```
CDPATH=:/d3/abc/def:/d3/abc
```

then the command

```
cd ghi
```

would cause the current directory, */d3/abc/def* directory, and */d3/abc* directory to be searched for the subdirectory *ghi*. If found in the */d3/abc/def* directory, the full pathname of the subdirectory would be printed and the current working directory would be changed to */d3/abc/def/ghi*.

MAIL, MAILCHECK, MAILPATH

When the *MAILPATH* variable is set, the **shell** informs the user of modifications to any of the files specified by the *MAILPATH* variable. The *MAIL* variable, if set, is ignored. When the *MAILPATH* variable is not set, the **shell** looks at the file specified by the *MAIL* variable and informs the user if there are any modifications.

If *MAILPATH=/d3/abc/def/mailfile,* then a change to *mailfile* would cause the message

```
You have mail
$
```

to be displayed when a check is made. Note that the prompt appears on the line after the message. To display a customized message, follow the file name with a % and the message. For example

```
MAILPATH=/d3/abc/def/mailfile%"Mailfile modified"
```

would cause the following message to be displayed after *mailfile* is modified

```
Mailfile modified
$
```

Several files can be checked by adding them to *MAILPATH.* For instance

```
MAILPATH=/usr/mail/def:/d3/abc/def/mailfile%"Mailfile
modified":/d3/abc/othermail%"Othermail modified"
```

would check for modifications to the three specified files. The standard mailfile is specified. Otherwise, the user would not be notified of the reception of standard mail except at login.

The *MAILCHECK* variable specifies how often (in seconds) the **shell** will check for mail. The default value is 600 seconds (10 minutes). If set to zero, the **shell** will check before each prompt. To set the *MAILCHECK* variable to zero, enter the following

```
MAILCHECK=0
```

The presence of mail in the standard mail file (*/usr/mail/*loginname) is announced at login regardless of the setting of *MAIL* or *MAILPATH* variables. Otherwise, to be notified of the arrival of mail, either the *MAIL* or *MAILPATH* variable must be set.

PS1

The variable *PS1* is used by the **shell** to specify the primary **shell** prompt. This is displayed at a terminal whenever the **shell** is awaiting a command input. The default primary prompt is **$**. To change the prompt to <>, for example, the following is entered

```
PS1="<>"
```

PS2

The variable *PS2* is used by the **shell** to specify the secondary **shell** prompt. This is displayed whenever the **shell** receives a newline in its input but more is expected. The default value of *PS2* is >. To change the prompt to **<more>** for example, the following is entered

```
PS2="<more>"
```

IFS

The variable *IFS* is used by the **shell** to specify the internal field separators. Normally, the *space*, *tab*, and *newline* characters are used. After parameter and command substitution, internal field separators are used to split the results of substitution into distinct arguments where such characters are found. Explicit null arguments (" " and ' ') are retained.

SHACCT

The variable *SHACCT* is used by the **shell** to specify a file for storing **shell** (as opposed to process) accounting records. Whenever a **shell** procedure is executed and the variable *SHACCT* is set, the **shell** will write an accounting record to the file specified by *SHACCT*. This file must be writable by the user. The accounting records can be analyzed by accounting routines such as **acctcom**(1) and **acctcms**(1M).

User Defined Variables

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

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by entering

```
user=fred box=m000 acct=mh000
```

to assign values to the variables *user*, *box*, and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with $; for example,

```
echo $user
```

will print *fred.*

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```

moves the *file* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution as in

```
echo ${user}
```

This 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 >$tmpa
```

directs the output of **ps**(1) to the file */tmp/psa,* whereas,

```
ps a >$tmpa
```

causes the value of the variable *tmpa* to be substituted.

SPECIAL COMMANDS

The following special commands are used in writing **shell** procedures. Many of the commands are only needed when programming. Others have nonprogramming uses.

```
:                  read
.                  readonly
break              return
continue           set
cd                 shift
echo               test
eval               times
exec               trap
exit               type
export             ulimit
hash               umask
newgrp             unset
pwd                wait
```

The ones that are useful to the casual (nonprogramming) user are described below.

cd

The **cd** command is used to change the current working directory as follows

```
cd [arg]
```

where *arg* specifies the new directory desired. For instance,

```
cd /d3/abc/ghi
```

moves the user from anywhere in the file system to the directory */d3/abc/ghi.* The full directory pathname must be specified to be used in this way. Execute permissions must be set in the desired directory.

If only the desired directory name is specified and the *CDPATH* variable is not set, then the current directory is searched for a subdirectory by that name. For instance, if the current directory */d3/abc* contains a subdirectory *subdir*, then the command

```
cd subdir
```

changes the current working directory to */d3/abc/subdir*. If the argument begins with **../**, the current working directory is changed relative to its parent directory. If the argument begins with **./**, the current directory value precedes additional arguments. For instance, if the current working directory is */d3/abc*, the following command

```
cd ./ghi
```

changes the current directory to */d3/abc/ghi.*

If the variable *CDPATH* is set, the **shell** searches each directory specified in *CDPATH* for the directory specified by the **cd** command. If the directory is present, the directory becomes the new working directory. (See "**CDPATH**" under "Keyword Parameters".)

exec

The command

```
exec [arg ...]
```

causes the command specified by *arg* to be executed in place of the **shell** without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the **shell** input/output to be modified.

hash

When a command is executed, it is entered into a special **hash** table. This table keeps track of what commands have been used, where they were located, and how much directory searching is involved in locating the command. Since this table is the first place that the **shell** looks, the amount of time used to search for a previously used command is reduced. Note that if a command is created and a command by the same name has been previously used, the **hash** table will contain only the location of the previously used command. The **hash** table is reinitialized upon each login session. The **hash** table can be cleared by entering

```
hash —r
```

To display the contents of the **hash** table, the following is entered

```
hash
```

The following is an example of a **hash** table:

```
hits        cost        command
3           5           /d3/abc/progbin/l
1           2           /bin/ed
1           7           /d3/abc/def/busy
1           2           /bin/date
2           2           /bin/who
1           2           /bin/ls
```

The **hits** column displays the number of times a command has been called. The **cost** column displays the number of nodes (i.e., **PATH**=node:node:node) searched to find the command. The **command** column displays the full pathname of the command.

An asterisk (*) displayed beside the **hits** information indicates that the command location may be reevaluated when the working directory is changed and the command is re-executed.

If a command *name* is entered with **hash**, the location of the command is determined and stored in the **hash** table without executing the command.

newgrp

By issuing the command **newgrp**(1), the user is assigned a new group identification. The command is of the form

```
newgrp [—] [group]
```

All access permissions are then evaluated with the new group. This allows access to files with different group ID permissions.

Entering **newgrp** with no argument changes the group identification back to the original group. When a — is entered, the environment is changed to the login environment.

pwd

The **pwd** command prints the full pathname of the current working directory. This command is especially useful when working directories are changed often.

set

The **set** command provides the capability of altering several aspects of the behavior of the **shell** by setting certain **shell** *flags.* Some of the more useful flags for the nonprogrammer and their meanings are:

—a	Mark variables that are modified or created for export.
—f	Disable file name generation.
—v	Print lines as they are read by the **shell**. The commands on each input line are executed after that input line is printed.
—x	Print commands and their arguments as they are executed. This causes a trace of only those commands that are actually executed.

To set the **x** flag for example, enter

```
set —x
```

To turn the **x** flag off for example, enter

```
set +x
```

These commands are especially useful for troubleshooting within **shell** procedures.

The **set** command entered with no arguments will display the values of variables in the environment.

type

The **type** command indicates how a specified command would be interpreted if used as a command name. The form of the command is

```
type [command-name]
```

For example, if the interpretation of the **cd** command is desired, enter

```
type cd
```

which returns

```
cd is a shell builtin
```

ulimit

The **ulimit** command has the form

```
ulimit [–f] [n]
```

When the option *–f* is used or if no option is specified, this command imposes a limit of *n* blocks on the size of files written by the **shell** and its child processes. Any size files may be read. If *n* is omitted, the current value of this limit is printed. The default value for *n* varies from one installation to another.

umask

The **umask** command has the form

```
umask [nnn]
```

The user file creation mask is set to *nnn*. This mask is used to determine the permission modes set on a file when it is created. For instance,

```
    umask 033
```

causes a newly created file to be assigned the permission set of 744. (See "PERMISSION MODES".)

unset

The **unset** command has the form

```
    unset [name ...]
```

For each variable *name*, the **shell** removes the corresponding variable or function. (This is not the same as making a variable null; removing a variable makes it nonexistent.) The variables *PATH*, *PS1*, *PS2*, *MAILCHECK*, and *IFS* cannot be *unset*.

RESTRICTED SHELL

A restricted **shell** is also available with the ICON/UXV operating system. This restricted version of **shell** is used to create an environment that controls and limits the capabilities. The actions of **rsh** are identical to that of **sh**, except that the following are disallowed:

- Changing directory
- Setting the value of *PATH* variable
- Specifying path or command names containing /
- Redirecting output (> and >>).

The system administrator often sets up a directory of commands that can be safely invoked by **rsh**. A restricted editor may also be provided.

Chapter 10

SHELL PROGRAMMING

PAGE

Chapter 10

SHELL PROGRAMMING

INTRODUCTION

This chapter describes **shell** as a programming language and builds upon the information provided in Chapter 2. It is expected that the reader has read Chapter 2 and has experience with ICON/UXV operating system commands.

INVOKING THE SHELL

The **shell** is an ordinary command and may be invoked in the same way as other commands:

sh *proc* [*arg*...]	A new instance of the **shell** is explicitly invoked to read *proc*.
sh **—v** *proc* [*arg* ...]	This is equivalent to putting **set —v** at the beginning of *proc*. Similarly for other set flags including **x**, **e**, **u**, and **n** flags.
proc [*arg* ...]	If *proc* is marked executable, and is not a compiled, executable program, the effect is similar to that of the **sh** *proc* [*args* ...] command. An advantage of this form is that *proc* may be found by the search procedure.

INPUT/OUTPUT

Unless redirected by a command inside the program, a **shell** program uses the input and output connections of the **shell** program. A redirection on a command changes redirection for that command only.

Single Line

The following could be used to print a line from a program

```
echo The date is:
date
```

and would result in

```
The date is:
Tue May 21 16:13:38 EDT 1984
```

Printing Error Messages

Normally, error messages are associated with file descriptor 2 and are sent to standard error. Error messages can be redirected to a file with the following command

```
sample 2>ERROR
```

If an error message is produced when running the program **sample**, the error output is redirected to the file *ERROR*.

Multiline Input (Here Documents)

One way to input several lines to programs is with what is referred to as "Here Documents". The general form is

```
cmd arg1 arg2 ... <<word
```

where everything entered at this command is accepted until *word* is entered on a line by itself. For example

```
sort <<finish
```

sends all the standard input to **sort** until **finish** is inputted. Then the input would be sorted and output to the terminal. For example

```
$ sort <<finish
> def
> abc
> finish
abc
def
```

Note that the primary system prompt ($) and the secondary system prompt (>) are shown. The final two lines are returned by the system.

The command

```
sort <<-word
```

removes all leading spaces or tabs.

SHELL VARIABLES

The **shell** has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*. *Parameters* are the variables normally set only on a command line. There are also *positional parameters* and *keyword parameters*. Other variables are simply names to which the user or the **shell** itself may assign string values.

Positional Parameters: When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters*. The argument in position zero on the command line (the name of the **shell** procedure itself) is called **$0**, the first argument is called **$1**, etc. The **shift** command may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the **set** command

```
set abc def ghi
```

which assigns "abc" to the first positional parameter (**$1**), "def" to the second (**$2**), and "ghi" to the third (**$3**). For this example, **set** also *unsets* **$4**, **$5**, etc. even if they were previously set. Positional parameter **$0** may not be assigned a value so that it always refers to the name of the **shell** procedure or to the name of the **shell** (in the login **shell**).

For instance,

```
set abc def ghi
echo $3 $2 $1
```

prints

```
ghi def abc
```

User-defined Variables: The **shell** also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement. Positional parameters can only be set as described in "Positional Parameters". A simple assignment is of the form

name=string

Thereafter, $*name* yields the value "string". A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment say appear in an assignment statement, but beware since *the*

shell *performs the assignments from right to left.* The following command line results in the variable *a* acquiring the value "abc"

```
a=$b b=abc
```

The following are examples of simple assignments. *Double* quotes around *the right-hand side* allow blanks, tabs, semicolons, and newlines to be included in "string", while also allowing *variable substitution* (also known as *parameter substitution*) to occur. In *parameter substitution*, references to positional parameters and other variable names that are prefaced by $ are replaced by the corresponding values, if any. *Single* quotes inhibit variable substitution. Some examples follow

```
MAIL=/usr/mail/gas
var="$1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable *var* has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of *, ?, [. . .]) does *not* apply in this context. Note that the value of **$asterisks** is the literal string "$stars", *not* the string "*****", because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in **$first** and **$second** having the same value

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces {} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore (digit only for positional parameters), then the braces are *required*

```
a='This is a string'
echo "${a}ent test"
```

returns the following message

```
This is a stringent test
```

Command Substitution: Any command line can be placed within grave accents (` . . .`) to capture the output of the command. This concept is known as *command substitution*. The command or commands enclosed between grave accents are first executed by the **shell** and then their output replaces the whole expression, grave accents and all. This feature is often combined with **shell** variables so that

```
today=`date`
```

assigns the string representing the current date to the variable *today* (e.g., **Tue Nov 27 16:01:09 EST 1984**). The command

```
users=`who | wc -l`
```

saves the number of logged-in users in the variable *users*. Any command that writes to the standard output can be enclosed in grave accents. Grave accents may be nested. The inside sets must be escaped with \. For example

```
logmsg=`echo Your login directory is \`pwd\``
```

Shell variables can also be given values indirectly by using the **shell** builtin command **read**. The **read** command takes a line from the standard input (usually the terminal) and assigns consecutive words on that line to any variables named

```
read first init last
```

will take an input line of the form

```
A. A. Smith
```

and has the same effect as if

```
first=A.   init=A.   last=Smith
```

had been typed.

The **read** command assigns any excess "words" to the last variable.

Predefined Speical Variables: Several variables have special meanings. The following are set *only* by the **shell**:

$# records the number of *positional* arguments passed to the **shell**, not counting the name of the **shell** procedure itself. The variable $# yields the number of the highest-numbered positional parameter that is set. Thus, **sh x a b c** sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments

```
if test $# -lt 2
then
   echo 'two or more args required'; exit
fi
```

$? is the exit status (also referred to as *return code, exit code,* or *value*) of the last command executed. Its value is a decimal string. Most ICON/UXV commands return **0** to indicate successful completion. The **shell** itself returns the current value of **$?** as *its* exit status.

$$ is the process number of the current process. Since process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names for temporary files. The ICON/UXV operating system provides no mechanism for the automatic creation and deletion of temporary files. A file exists until it is explicitly removed. Temporary files are generally undesirable. The ICON/UXV pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose

```
temp=$HOME/temp/$$
ls > $temp
commands, some of which use $temp, go here
rm $temp
```

$! is the process number of the last process run in the background. Again, this is a string of up to five digits.

$– is a string consisting of names of execution flags currently turned on in the **shell**. The $– variable has the value **xv** when tracing output.

CONDITIONAL SUBSTITUTION

Normally, the **shell** replaces occurrences of *$variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution depending upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string which may be assigned to a variable in any one of the following ways

```
A=
bcd=""
Ef_g=''
set '' ""
```

The first three of these examples assign the null string to each of the corresponding ***shell variables.*** The last example sets the first and second *positional parameters* to the null **string** and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null.* (Note that, in these expressions, *variable* refers to either a digit or a variable name.

${*variable:—string*} If *variable* is set and is non-null, then substitute the value $*variable* in place of this expression. Otherwise, replace the expression with *string.* Note that the value of *variable* is *not* changed by the evaluation of this expression.

${*variable:=string*} If *variable* is set and is non-null, then substitute the value $*variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value $*variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

${*variable:?string*} If *variable* is set and is non-null, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

> *variable*: *string*

and exit from the current **shell**. (If the **shell** is the login **shell**, it is not exited.) If *string* is omitted in this form, then the message

> *variable*: parameter null or not set

is printed instead.

${*variable:+string*} If *variable* is set and is non-null, then substitute *string* for this expression; otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon (:). In this case, the **shell** does *not* check whether *variable* is null or not. It only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If *PATH* has ever been set and is not null, then keep its current value. Otherwise, set it to the string *:/bin:/usr/bin.* Note that one needs an explicit assignment to set *PATH* in this form

```
PATH=${PATH:—':/bin:/usr/bin'}
```

2. If *HOME* is set and is not null, then change directory to it; otherwise, set it to */usr/gas* and change directory to it. Note that *HOME* is automatically assigned a

value in this case

```
        cd ${HOME:='/usr/gas'}
```

CONTROL COMMANDS

The **shell** provides several commands that are useful in creating **shell** procedures. A few definitions are needed before explaining the commands.

A *simple command* is defined as a sequence of nonblank arguments separated by blanks or tabs. The first argument usually specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed to the command. Input/output redirection arguments can appear in a simple command line and are passed to the **shell**, *not* to the command.

A *command* is a simple command or any of the **shell** commands described below. A *pipeline* is a sequence of one or more commands separated by ¦. (For historical reasons, ^ is a synonym for ¦ in this context.) The standard output of each command but the last in a pipeline is connected [by a *pipe(2)]* to the standard input of the next command. Each command in a pipeline is run separately. The **shell** waits for the last command to finish. If no exit status argument is specified, the exit status is that of the last command executed (an end-of-file will also cause the **shell** to exit).

A *command list* is a sequence of one or more pipelines separated by ;, **&**, **&&**, or ¦¦, and optionally terminated by ; or **&**. A semicolon (;) causes sequential execution of the previous pipeline (i.e., the **shell** waits for the pipeline to finish before reading the next pipeline), while **&** causes asynchronous execution of the preceding pipeline. Both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily or until its processes are **kill**ed.

More typical uses of **&** include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, typing

```
    nohup cc prog.c&
```

allows one to continue working while the C compiler runs in the background. A command line ending with **&** is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The **nohup** command is used for this purpose. Without **nohup**, if one hangs up while **cc** in the above example is still executing, **cc** will be killed and the output will disappear.

The **&&** and ¦¦ operators, which are of equal precedence (but lower than **&** and ¦), cause conditional execution of pipelines. In **cmd1** ¦¦ **cmd2**, **cmd1** is executed and its exit status examined. Only if **cmd1** fails (i.e., has a nonzero exit status) is **cmd2** executed. This is thus a more terse notation for

```
if   cmd1
        test $? != 0
then
     cmd2
fi
```

The **&&** operator yields the complementary test: in **cmd1 && cmd2**, the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints *two* separate documents

```
{ nroff —cm text1; nroff —cm text2; } | col
```

Programming Constructs

Several control flow commands are provided in the **shell** that are especially useful in programming. These are referred to as programming constructs and are described below.

A command often used with programming constructs is the **test**(1) command. An example of the use of the **test** command is

```
test —f file
```

This command returns zero exit status (true) if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) and "Test" under "SPECIAL COMMANDS" for more information].

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.

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 around the loop *command-list1* is executed. If a zero exit status is returned, then *command-list2* is executed; otherwise, the loop stops. For example,

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

The **shift** command is a **shell** command that renames the positional parameters **$2**, **$3**, **...** as **$1**, **$2**, **...** and loses **$1**.

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

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

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

A file **print** could be written to use **while** and **test** as follows

```
while test $# != 0
do
    echo "$1 being submitted"
    lp —dprtd42 —c —o12 —w —tuser1 $1
    shift
done
lpstat —oprtd42
```

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**. If a zero exit status is returned, the command-list following the **then** is executed. If a zero exit status is not returned, the command-list following the **else** is executed.

The **if** command may be used 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
```

A multiple test **if** command of the form

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

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

A file could be written to include the use of **if** and **test** as follows

```
if test $# = 0
then
    echo "enter a filename after $0"
else
    if [ ! –f $1 ]
    then
        echo "$1 does not exist"
        echo "Enter a filename that exists" ; exit
    else
    echo "$1 being submitted"
    lp –dprtd42 –c –o12 –w –tuser1 $*
    lpstat –oprtd42
    fi
fi
```

The [...] is shorthand for **test**. The **if [! –f $1]** means **if** the file $1 does not exist **then** do this.

The sequence

```
if command1
then
      command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

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".

The command

```
tel fred bert
```

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 words
do
    command-list
done
```

A **command-list** is a sequence of one or more simple commands separated or ended by a newline or a semicolon. A *name* is a **shell** variable that is set to *words*... in turn each time the **command-list** following **do** is executed. If *words* ... is omitted, then the loop is executed once for each positional parameter; that is, **in** $* is assumed. Execution ends when there are no more words in the list.

An 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* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

The **for** can also be used in a program. Assume a document is formatted and stored in chapters (files) that begin with the letters "ch" (ch1, ch2, ch3, and chtoc). A program can be written to send the document to the line printer. The program contains

```
for i in ch*
do
    lp –dprtd42 –c –o12 –w –tuser1 $i
done
    lpstat –oprtd42
```

This will send each chapter as a separate job. Notice that $i is used instead of $*.

Control Flow—case

A multiple way (choice) 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. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

$# is the string "1", and the standard input is appended (copied) onto the end of *file* using the **cat**(1) command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

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

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

Caution: No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

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

A program **print** can be used to send a document to different line printers. Assume there are two line printers named "prtd42" and "prtd43". Send a document to "prtd42" as follows

```
print 42 files
```

Send a document to "prtd43" as follows

```
print 43 files
```

The **print** program contains the following

```
case $1 in
42) shift;lp —dprtd42 —c —o12 —w —tuser1 $*;lpstat —oprtd42;;
43) shift;lp —dprtd43 —c —o12 —w —tuser1 $*;lpstat —oprtd43;;
 *) echo "line printer does not exist";;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

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

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

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

is equivalent to

```
case $i in
    —[xy])...
esac
```

The usual quoting conventions apply so that

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

will match the character **?**.

Functions

Functions may be defined and used in the current **shell**. The general form of a function is

> *name* () {*list*}

A space or a newline is required after the beginning brace ({). A semicolon or newline is required before the terminating brace (}). As an example, a function could be defined to see how many people are currently on the system as follows

```
busy () { who | wc —l;}
```

The **wc** —l command returns a count of the lines returned by the **who** command. Notice that a space is placed after the beginning brace ({) and a semicolon is placed before the terminating brace (}). This function is called by its name as follows

```
busy
```

which returns

```
12
```

if 12 people are logged into the system.

The same function could be defined using multiple lines as follows

```
busy ()
{
who | wc —l
}
```

Positional parameters can be used to pass information to a function. For instance,

```
present () { who ¦ grep $1 ; }
```

searches the output of the **who** command for the value of the positional parameter **$1** and prints all lines containing the value. For instance, the entry

```
present abc
```

returns

```
abc        tty09        May 7 09:31
```

if abc is logged in on tty09.

The current **shell** contains the function definition. A different **shell** would not be able to execute the function until it is defined in that **shell**. To display the functions defined in the current **shell**, enter

```
set
```

The value for all variable names will be displayed including the functions defined.

SPECIAL COMMANDS

There are several special commands that are *internal* to the **shell** (some of which have already been mentioned). These commands should be used in preference to other ICON/UXV commands whenever possible because they are faster and more efficient. The **shell** does not fork to execute these commands, so no additional processes are spawned.

Many of these special commands were described in Chapter 2. These commands include:

```
cd
exec
hash
newgrp
pwd
set
type
ulimit
umask
unset.
```

Descriptions of the remaining special commands follow. These commands include:

```
:
.
break
continue
echo
eval
exit
export
read
readonly
return
shift
test
times
trap
wait.
```

: (Colon)

The : command is the null command. This command can be used to return a zero (true) exit status.

. (Period)

The . command has the form

```
. file
```

This command reads and executes commands from **file** and returns. The search path specified by *PATH* is used to find the directory containing *file.* If the file **command1** contained the following

```
echo Today is:
date
```

then the command

```
. command1
```

returns

```
Today is:
Thu Sep 22 14:40:04 EDT 1984
```

Any currently defined variable can be used in the **shell** procedure called.

break

This command has the form

```
break [n]
```

This command is used to exit from the enclosing **for**, **until**, or **while** loop. If *n* is specified, then exit *n* levels. An example of **break** is as follows

```
# This procedure is interactive; the 'break'
# command is used to allow
# the user to control data entry.
while true
do
        echo "Please enter data"
        read response
        case "$response" in
            "done")     break       # no more data
                        ;;
            *)
                        process the data here
                        ;;
        esac
done
```

continue

This command has the form

```
continue [n]
```

This command causes the resumption of an enclosing **for**, **until**, or **while** loop. If *n* is specified, then it resumes at the *n*-th enclosing loop.

echo

The form of the echo command is

```
echo [arg ...]
```

The **echo** command writes its arguments separated by blanks and terminated by a newline on the standard output. For instance, the input

```
echo Message to be printed.
```

returns

```
Message to be printed.
```

The following escapes can be used with **echo**:

```
\b  backspace
\c  print line without new-line
\f  new-line
\r  carriage return
\t  tab
\  backslash
\n  the 8-bit character whose ASCII code is the 1-,
    2-, or 3-digit octal number, which must start
    with a zero.
\v vertical tab
```

For example

```
echo "The current date is \c"
date
```

would return

```
The current date is Tue May 16 08:00:30 EDT 1984
```

eval

Sometimes, one builds command lines inside a **shell** procedure. In this case, one might want to have the **shell** rescan the command line after all the initial substitutions and expansions are done. The special command **eval** is available for this purpose. The form of this command is

```
eval [arg ...]
```

The **eval** command takes a command line as its argument and simply rescans the line performing any variable or command substitutions that are specified. Consider the following situation

```
command=who
output='|wc -l'
eval $command $output
```

This segment of code results in the pipeline **who|wc -l** being executed.

The uses of **eval** can be nested.

exit

A **shell** program may be terminated at any place by using the **exit** command. The form of the **exit** command is

```
exit [n]
```

The **exit** command can also be used to pass a return code (*n*) to the **shell**. By convention, a **0** return code means **true** and a **1** to **255** return code means **false**. The return code can be found by **$?**. For instance, if the executable procedure **testexit** contained

```
exit 5
```

then

```
testexit
```

would execute **testexit**. The command

```
echo $?
```

would return

```
5
```

export

The form of the **export** command is

```
export [name ...]
```

The **export** command places the named variables in the environments of both the **shell** and all its future child processes. Normally, all variables are local to the **shell** program. Commands executed from within the **shell** program do not have access to the local variables. If a variable is **export**ed, then the commands within the **shell** program will be able to access the variable.

To export variables, the following command is used

```
export variable1 variable2 ...
```

To obtain a list of variables **export**ed, the following command is entered

```
export
```

read

A variable may also be set using the **read** command. The **read** command reads one line from the standard input of the **shell** procedure and puts that line in the variables which are its arguments. Leading spaces and tabs are stripped off. The general form of the command is

```
read variable1 variable2 ...
```

The last variable gets what is left over. For example, if **testread** contains the following

```
echo 'Please type your first and last name:\c'
read first_name last_name
echo Your name is ${first_name} ${last_name}
```

then when the program is run the first line would be printed

```
Please type your first and last name:
```

and would wait for the input. (The input would appear on the same line.) Assuming the name is **Jane Doe**, after the input, the following line would be printed

```
Your name is Jane Doe
```

readonly

Variables can be made **readonly**. After becoming readonly, a variable cannot receive a new value. The general form of the command is

```
readonly variable-name variable-name ...
```

To print the names of variables that are readonly, enter

```
readonly
```

return

The return command causes a function to exit with a specified return value. The form of the command is

```
return [n]
```

where *n* is the desired return value. When *n* is omitted, the return status of the last command executed is displayed.

shift

The **shift**[**sh**(1)] command reassigns the positional parameters. Positional parameter **$1** would receive the value of **$2**, **$2** would receive the value of **$3**, etc. Notice that **$0** (the procedure name) is unchanged and that the number of positional parameters ($#) is decremented.

If the executable program **shifter** contains the following

```
echo ${#} positional parameters
echo ${*}
echo Now shift
shift
echo ${#} positional parameters
echo ${*}
```

then the command

```
shifter first second third
```

would result in

```
3 positional parameters
first second third
Now shift
2 positional parameters
second third
```

test

The **test**(1) command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. The **test** command also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the *command list* following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotes if there is any chance of their being null or not set.

The square brackets ([]) may be used as an alias for **test**; e.g., [*expression*] has the same effect as **test** *expression*.

The following is a partial list of the primaries that can be used to construct a conditional expression:

—r *file*	*true* if the named file exists and is readable by the user.
—w *file*	*true* if the named file exists and is writable by the user.
—x *file*	*true* if the named file exists and is executable by the user.
—s *file*	*true* if the named file exists and has a size greater than zero.
—d *file*	*true* if the named file exists and is a directory.
—f *file*	*true* if the named file exists and is an ordinary file.
—p *file*	*true* if the named file exists and is a named pipe (*fifo*).
—z *s1*	*true* if the length of string "s1" is zero.
—n *s1*	*true* if the length of the string "s1" is nonzero.
—t *fildes*	*true* if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor **1** is used by default.
s1 = *s2*	*true* if strings "s1" and "s2" are identical.
s1 != *s2*	*true* if strings "s1" and "s2" are *not* identical.
s1	*true* if "s1" is *not* the null string.

n1 **—eq** *n2*	*true* if the integers *n1* and *n2* are algebraically equal. Other algebraic comparisons are indicated by **—ne**, **—gt**, **—ge**, **—lt**, and **—le**.

These primaries may be combined with the following operators:

!	unary negation operator.
—a	binary logical *and* operator.
—o	binary logical *or* operator. The **—o** has lower precedence than **—a**.
(*expr*)	parentheses for grouping; they must be escaped to remove their significance to the **shell**. When parentheses are absent, the evaluation proceeds from left to right.

Note that all primaries, operators, file names, etc. are separate arguments to **test**.

For example, the procedure **nametest**

```
if test —d $1
   then echo $1 is a directory
elif test —f $1
   then echo $1 is a file
else echo $1 does not exist
fi
```

then if the file bucket existed, then

```
bucket is a file
```

would be returned.

times

The **times** command prints the accumulated user and system times for processes run from the **shell**. The **times** command is entered on a line by itself. For example, the command

```
times
```

returns

```
0m3s 0m10s
```

trap

A **shell** program may handle interrupts by using the trap command. The trap command interfaces with the underlying ICON/UXV operating system mechanism for handling interupts.

The ICON/UXV operating system provides signals that tell a program when some unusual condition has occurred. These signals may be from the keyboard or from other programs.

By default, if a program receives a signal, the program will terminate. However, these signals may be caught, the program suspended, the interrupt routine run, and the program restarted at the point it was suspended. Or these signals may be ignored.

trap *arg signal-list*

is the form of the **trap** command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers [as described in **signal**(2)].

The following signals are used in the ICON/UXV operating system:

01	hangup
02	interrupt
03	quit
04	illegal instruction
05	trace trap
06	IOT instruction
07	EMT instruction
08	floating point exception
09	kill
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 signal
16	user defined signal 1
17	user defined signal 2
18	death of a child
19	power fail.

The commands in *arg* are scanned at least once when the **shell** first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The single quotes inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the **shell**. The following procedure will print the name of the current directory on the file **errdirect** when it is interrupted, thus giving the user information as to how much of the job was done

```
trap 'echo 'pwd' >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
     cd $i
        commands to be executed in directory $i here
done
```

while the same procedure with double (rather than single) quotes (**trap "echo 'pwd' >errdirect" 2 3 15**) will, instead, print the name of the directory from which the procedure was executed.

Signal 11 (SEGMENTATION VIOLATION) may never be trapped because the **shell** itself needs to catch it to deal with memory allocation. Zero is not a ICON/UXV system signal. Zero is effectively interpreted by the **trap** command as a signal generated by exiting from a **shell** (either via an **exit** command or by "falling through" the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string ('' or ""), then the signals in *signal-list* are *ignored* by the **shell**.

The most frequent use of **trap** is to assure removal of temporary files upon termination of a procedure. The second example of "Predefined Special Variables" in subpart "D. Shell Variables" would be written more typically as follows

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
    commands, some of which use $temp, go here
```

In this example whenever signals 1 (HANGUP), 2 (INTERRUPT), 3 (QUIT), or 15 (SOFTWARE TERMINATION) are received by the **shell** procedure or whenever the **shell** procedure is about to exit, the commands enclosed between the single quotes will be executed. The **exit** command must be included or else the **shell** continues reading commands where it left off when the signal was received. The **trap 0** turns off the original trap on exits from the **shell** so that the **exit** command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the **shell** continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, thus terminating the **while** loop and restarting the cycle for the next directory. The entire procedure is terminated if interrupted when waiting for input; but during the execution of a command, an interrupt terminates *only* that command.

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

Several **traps** may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type

```
trap
```

It is important to understand some things about the way the **shell** implements the **trap** command in order not to be surprised. When a signal (other than 11) is received by the **shell**, it is passed on to whatever child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the **shell** *then* polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward except in the case of traps set at the command (outermost or login) level. In this case, it is possible that no child process is running, so the **shell** waits for the termination of the first process spawned *after* the signal is received before it polls the traps.

For internal commands, the **shell** normally polls traps on completion of the command. An exception to this rule is made for the **read**, **hash**, and **echo** commands.

wait

The **wait** command has the following form

```
wait [n]
```

With this command, the **shell** waits for the child process whose process number is *n* to terminate. The exit status of the wait command is that of the process waited on. If *n* is omitted or is not a child of the current **shell**, then *all* currently active processes are waited for and the return code of the **wait** command is zero. For example, the executable program **format**

```
while test "$1" != ""
nroff $1>>junk&
shift
wait $!
done
echo ***nroff complete***
```

envokes the nroff formatter for each file specified and informs the user when it is finished. If the files *chapter1* and *chapter2* required formatting, the entry

```
format chapter1 chapter2
```

would format the two chapters and when they are finished return

```
***nroff complete***
```

COMMAND GROUPING

Commands may be grouped in two ways

{ *command-list* ; }

and

(*command-list*)

The first form, *command-list*, is simply executed. The second form executes *command-list* as a separate process. If a list of commands is enclosed in a pair of parentheses, the list is executed as a subshell. The subshell inherits the environment of the main **shell**. The subshell does not change the environment of the main **shell**. 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*.

A COMMAND'S ENVIRONMENT

All the variables (with their associated values) known to a command at the beginning of execution of that command constitute its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a **shell** passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the **shell** *and* its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example

```
#       key_command
echo $a $b
```

is a simple procedure that **echo**es the values of two variables. If it is invoked as

```
a=key1 b=key2 key_command
```

then the output is

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count $#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are *not* reflected in the environment. The changes are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made **export**able from the current **shell**, type

```
export
```

To get a list of name-value pairs in the current environment, type

```
env
```

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 may be invoked without changing the procedure by entering

```
sh —v proc ...
```

where *proc* is the name of the **shell** procedure. This flag may be used with the **—n** flag to prevent execution of later commands. (Note that typing "**set —n**" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set —x
```

will produce an execution trace with flag **—x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

```
set —
```

and the current setting of the **shell** flags is available as $—.

Chapter 11

EXAMPLES OF SHELL PROCEDURES

Chapter 11

EXAMPLES OF SHELL PROCEDURES

Some examples in this subpart are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.

copypairs

```
#       usage: copypairs file1 file2 ...
#       copy file1 to file2, file3 to file4, ...
while test "$2" != ""
do
       cp $1 $2
       shift; shift
done
if test "$1" != ""
then
    echo "$0: odd number of arguments"
fi
```

Note: This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop because you can adjust the positional parameters via **shift** to handle related arguments.

copyto

```
#       usage: copyto dir file ...
#       copy argument files to 'dir',
#       making sure that at least
#       two arguments exist and that 'dir'
#       is a directory
if test $# -lt 2
then
     echo "$0: usage: copyto directory file ..."
elif test ! -d $1
then
     echo "$0: $1 is not a directory";
else
     dir=$1; shift
  for eachfile
  do
        cp $eachfile $dir
  done
fi
```

Note: This procedure uses an **if** command with two tests in order to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first. The original **$1** is shifted off.

distinct

```
#       usage: distinct
#       reads standard input and reports
#       list of alphanumeric strings
#       that differ only in case,
#       giving lower-case form of each
tr -cs '[A-Z][a-z][0-9]' '[\012*]' | sort -u |
       tr '[A-Z]' '[a-z]' | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. It may not be immediately obvious how this works. [See **tr**(1), **sort**(1), and **uniq**(1) if you are completely unfamiliar with these commands.] The **tr** translates all characters except letters and digits into newline characters and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The **uniq -d** prints (once) only those lines that occur more than once yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged. The two lines below are equivalent assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1>temp1;cmd2<temp1>temp2;cmd3<temp2;rm temp[12]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic **distinct** with such a step-by-step process using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, exercise some restraint lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

draft

```
# usage: draft file(s)
# prints the draft (—rC3) of a document on a DASI 450
# terminal in 12-pitch using memorandum macros (MM).
nroff —rC3 —T450—12 —cm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags. These flags cannot be given default values that are reasonable for all (or even most) users.

edfind

```
# usage: edfind file arg
# find the last occurrence in 'file' of a line whose
# beginning matches 'arg', then print 3 lines (the one
# before, the line itself, and the one after)
ed - $1 <<!
H
?^$2?;-,+p
!
```

Note: This procedure illustrates the practice of using editor (**ed**) inline input scripts into which the **shell** can substitute the values of variables. It is a good idea to turn on the **H** option of **ed** when embedding an **ed** script in a **shell** procedure [see **ed**(1)].

edlast

```
# usage: edlast file
# prints the last line of file, then deletes that line
ed - $1 <<-\eof # no variable substitutions in "ed" script
        H
        $p
        $d
        w
        q
eof
echo Done.
```

Note: This procedure contains an in-line input document or script; it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, **eof**) of the input redirection. If this had not been done, **$p** and **$d** would have been treated as **shell** variables.

fsplit

```
# usage: fsplit file1 file2
# read standard input and divide it into three parts:
# append any line containing at least one letter
# to file1, any line containing at least one digit
# but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
      total="`expr $total + 1`"
      case "$next" in
      *[A–Za–z]*)
                echo "$next" >> $1 ;;
      *[0–9]*)
                echo "$next" >> $2 ;;
      *)
                lost="`expr $lost + 1`"
      esac
done
echo "$total lines read, $lost thrown away"
```

Note: In this procedure, each iteration of the **while** loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file.

Do not use the **shell** to read a line at a time unless you must—it can be grotesquely slow.

initvars

```
#       usage: . initvars
#       use carriage return to indicate "no change"
echo "initializations? \c"
read response
if test "$response" = y
then
        echo "PS1=\c"; read temp
                PS1=${temp:-$PS1}
        echo "PS2=\c"; read temp
                PS2=${temp:-$PS2}
        echo "PATH=\c"; read temp
                PATH=${temp:-$PATH}
        echo "TERM=\c"; read temp
                TERM=${temp:-$TERM}
fi
```

Note: This procedure would be invoked by a user at the terminal or as part of a *.profile* file. The assignments are effective even when the procedure is finished because the **dot** command is used to invoke it. To better understand the **dot** command, invoke **initvars** as indicated above and check the values of **PS1**, **PS2**, **PATH**, and **TERM**; then make **initvars** executable, type **initvars**, assign different values to the three variables, and check again the values of these three **shell** variables after **initvars** terminates. It is assumed that **PS1**, **PS2**, **PATH**, and **TERM** have been **export**ed, presumably by your *.profile*.

merge

```
#       usage:  merge src1 src2 [ dest ]
#       merge two files, every other line.
#       the first argument starts off the merge,
#       excess lines of the longer
#       file are appended to
#       the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}# default destination file is named $1.m
while true
do
          # alternate reading from the files;
          # 'more' represents the file descriptor
          # of the longer file
      line <&4 >>$dest || { more=5; break ;}
      line <&5 >>$dest || { more=4; break ;}
done
          # delete the last line of destination
          # file, because it is blank.
ed - $dest <<\eof
      H
      $d
      w
      q
eof
while line <&$more >> $dest
do :; done # read the remainder of the longer
        # file—the body of the 'while' loop
        # does nothing; the work of the loop
        # is done in the command list following
        # 'while'
```

Note: This procedure illustrates a technique for reading sequential lines from a file or files without creating any subshells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used **src1** and **src2** explicitly rather than the associated file descriptors, this procedure would never terminate because the *first* line of each file would be read over and over again.

mkfiles

```
# usage: mkfiles pref [ quantity ]
# makes 'quantity' (default = 5) files,
# named pref1, pref2, ...
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
      > $1$i
      i="`expr $i + 1`"
done
```

Note: This procedure uses input/output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop. Compare this procedure with procedure **null** below.

mmt

```
if test "$#" = 0; then cat <<\!
Usage: "mmt [ options ] files" where "options" are:
—a     => output to terminal
—e     => preprocess input with eqn
—t     => preprocess input with tbl
—Tst   => output to STARE phototypesetter by Honeywell
—T4014 => output to 4014 manufactured by Tektronix
—Tvp   => output to printer manufactured by Versatec
— => use instead of "files" when mmt used inside a pipeline.
Other options as required by TROFF and the MM macros.
!
        exit 1
fi
PATH='/bin:/usr/bin'; O='—g'; o='|gcat —ph';
# Assumes typesetter is accessed via gcat(1)
# If typesetter is on-line, use O=''; o=''
while test —n "$1" —a ! —r "$1"
do
   case "$1" in
    —a)         O='—a';        o='' ;;
    —Tst)        O='—g';       o='|gcat —st';;
#          Above line for STARE only
    —T4014)     O='—t';        o='|tc';;
    —Tvp)        O='—t';       o='|vpr —t';;
    —e)         e='eqn';;
    —t)         f='tbl';;
    —)          break;;
    *)            a="$a $1";;
   esac
   shift
done
if test —z "$1"
then
                    echo 'mmt: no input file'
                    exit 1
fi
if test "$O" = '—g'
then
                    x="—f$1"
fi
d="$*"
if test "$d" = '—'
then
                    shift
                    x=''
                    d=''
fi
if test —n "$f"
```

```
then
              f="tbl $*|"
              d=''
fi
if test —n "$e"
then
             if test —n "$f"
                then e='eqn|'
                else e="eqn $*|"
                d=''
             fi
fi
eval "$f $e troff $O —cm $a $d $o $x";    exit 0
```

Note: This is a slightly simplified version of an actual UNIX system command. It uses many of the features available in the **shell**. If you can follow through it without getting lost, you have a good understanding of **shell** programming. Pay particular attention to the process of building a command line from **shell** variables and then using **eval** to execute it.

null

```
#       usage: null file
#       create each of the named files
#       as an empty file
for eachfile
do
      > $eachfile
done
```

Note: This procedure uses the fact that output redirection creates the (empty) output file if that file does not already exist. Compare this procedure with procedure **mkfiles** above.

phone

```
#       usage: phone initials
#       prints the phone number(s) of person
#       with given initials
echo 'inits     ext      home'
grep "^$1" <<\!
abc      1234      999-2345
def      2234      583-2245
ghi      3342      988-1010
xyz      4567      555-1234
!
```

Note: This procedure is an example of using an inline input document or *script* to maintain a *small* data base.

writemail

```
#   usage: writemail message user
#   if user is logged in, write message on terminal;
#   otherwise, mail it to user
echo "$1" | { write "$2" || mail "$2" ;}
```

Note: This procedure illustrates command grouping. The message specified by **$1** is piped to the **write** command and, if **write** fails, to the **mail** command.

Chapter 12

GRAPHICS OVERVIEW

PAGE

Chapter 12

GRAPHICS OVERVIEW

Chapter Introduction

The ICON/UXV System Graphics, or **graphics**, is the name given to a collection of numerical and graphical commands available as part of the ICON/UXV operating system. In the current release, **graphics** includes commands to construct and edit numerical data plots and hierarchy charts. This chapter will help a user get started using **graphics** and show where to find more information. The examples below assume that the user is familiar with the ICON/UXV operating system shell.

Basic Concepts

The basic approach taken with **graphics** is to generate a drawing by describing it rather than by drafting it. Any drawing is seen as having two fundamental attributes—its underlying logic and its visual layout. The layout encompasses one representation of the logic. For example, consider the attributes of a drawing that consists of a plot of the function $y=x^2$ for x between 0 and 10:

- The logic of the plot is the description as just given, namely $y=x^2$, for x between 0 and 10.
- The layout consists of an x-y grid, axes labeled perhaps 0 to 10 and 0 to 100, and lines drawn connecting the x-y pairs 0,0 to 1,1 to 2,4 etc.

The way to generate a picture in **graphics** is:

```
gather data | transform the data | generate a layout |
    display the layout
```

The command to generate the plot, $y=x^2$, for x between 0 and 10 and display it on a TEKTRONIX display terminal would be

```
gas –s0,t10 | af "x^2" | plot | td
```

where:

- The **gas** command generates sequences of numbers, in this case starting at 0 and terminating at 10.
- The **af** command performs general arithmetic transformations.
- The **plot** command builds x-y plots.

- The **td** command displays drawings on TEKTRONIX terminals.

The resulting drawing is shown in *Figure 12-1.*

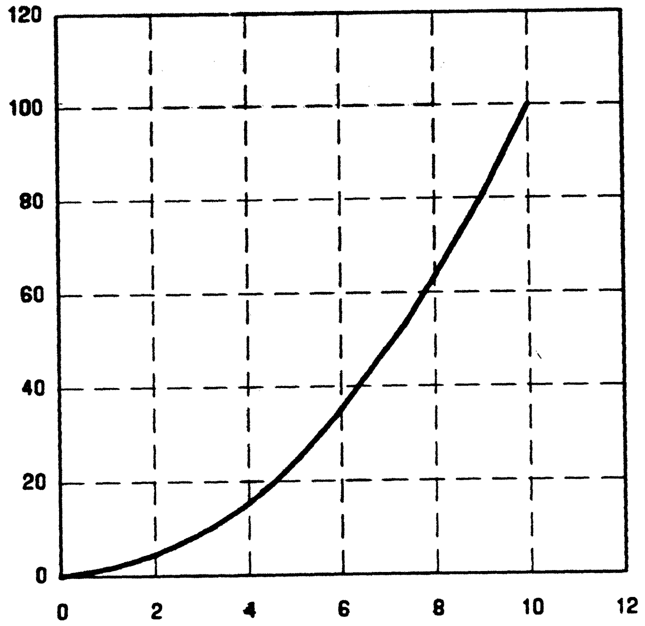


Figure 12-1. Plot of gas -s0,t10 ¦ af "x^2" ¦ plot ¦ td

The layout generated by a **graphics** program may not always be precisely what is wanted. There are two ways to influence the layout. Each drawing program accepts options to direct certain layout features. For instance, in the previous example, it may be desired to have the x-axis labels indicate each of the numbers plotted and not have any y-axis labels at all. To achieve this the **plot** command would be changed to:

```
gas -s0,t10 | af "x^2" | plot -xi1,ya | td
```

producing the drawing of Figure 12-2.

The output from any drawing command can also be affected by editing it directly at a display terminal using the graphical editor, **ged**. To edit a drawing really means to edit the computer representation of the drawing. In the case of **graphics** the representation is called a graphical primitive string, or GPS. All of the drawing commands (e.g., **plot**) write GPS, and all of the device filters (e.g., **td**) read GPS. **ged** allows manipulation of GPS at a display terminal by interacting with the drawing the GPS describes.

The GPS describes graphical objects drawn within a Cartesian plane, 65,534 units on each axis. The plane, known as the *universe*, is partitioned into 25 equal-sized square regions. Multidrawing displays can be produced by placing drawings into adjacent regions and then

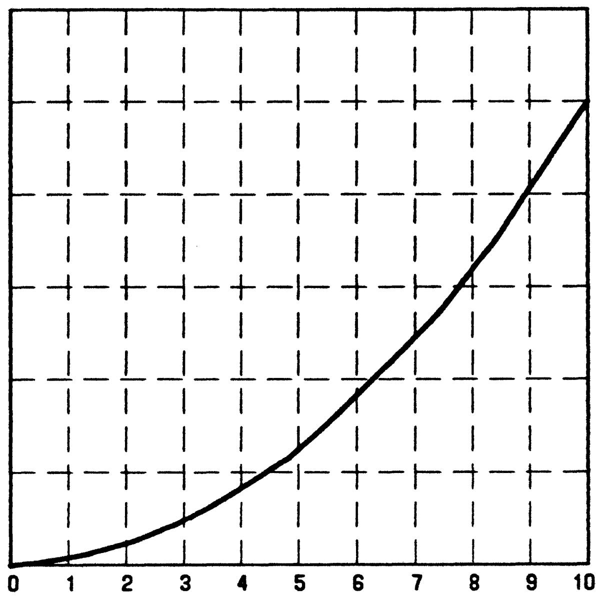


Figure 12-2. Plot of gas -s0,t10 ¦ af "x^2" ¦ plot -xi1,ya ¦ td

displaying each region.

Getting Started

To access the **graphics** commands when logged in on a ICON/UXV system, type **graphics**. The shell variable *PATH* will be altered.to include the **graphics** commands and the shell primary prompt will be changed to^. Any command accessible before typing **graphics** will still be accessible; **graphics** only adds commands, it does not take any away. Exception, the 3B*20 computers **list** command cannot be accessed in the **graphics** mode. Once in **graphics**, a user can find out about any of the **graphics** commands using **whatis**. Typing **whatis** by itself on a command line will generate a list of all the commands in **graphics** along with instructions on how to find out more about any of them.

All of the **graphics** commands accept the same command line format:

- A *command* is a *command-name* followed by *argument*(s).
- A *command-name* is the name of any of the **graphics** commands.
- An *argument* is a *file-name* or an *option-string.*
- A *file-name* is any file name not beginning with —, or a — by itself to reference the standard input.
- An *option-string* is a — followed by *option*(s).
- An *option* is a letter(s) followed by an optional value. Options may be separated by commas.

OVERVIEW

The **graphics** commands will produce the best results when used with a display terminal such as the TEKTRONIX display terminal. **Tplot**(1G) filters can be used in conjunction with **gtop** (see **gutil**(1G)) to get somewhat degraded drawings on Versatec* printers and Dasi-type terminals. Since GPS can be stored in a file, it can be created from any terminal for later display on a graphical device.

The **graphics** commands can be removed from user's *PATH* shell variable by typing an end-of-file indication (control-d on most terminals). To log off the ICON/UXV operating system from **graphics**, type **quit**.

Examples Of What You Can Do

Numerical Manipulation and Plotting

Stat is a collection of numerical and plotting commands. All of these commands operate on vectors. A vector is a text file that contains numbers separated by delimiters, where a delimiter is anything that is not a number.

For example:

```
1 2 3 4 5, and
arf tty47 Mar 5 09:52
```

are both vectors. The latter is the vector:

```
47 5 9 52.
```

Here is an easy way to generate a Celsius-Fahrenheit conversion table using **gas** to generate the vector of Celsius values:

```
gas –s0,t100,i10 | af "C,9/5*C+32"       2>/dev/null
```

* Registered trademark of Versatec Corporation.

The output is:

```
0           32
10          50
20          68
30          86
40          104
50          122
60          140
70          158
80          176
90          194
100         212
```

where:

- **gas –s0,t100,i10** generates a sequence that starts at 0, terminates at 100, and the increment between successive elements is 10.

- **af "C,9/5*C+32"** generates the table. Arguments to **af** are expressions. Operands in an expression are either constants or file names. If a file name is given that does not exist in the current directory it is taken as the name for the standard input. In this example **C** references the standard input. The output is a vector with odd elements coming from the standard input and even elements being a function of the preceding odd element.

- **2>/dev/null** suppresses the printing of warning messages. It redirects error message to **/dev/null**.

Here is an example that illustrates the use of vector titles and multiline plots:

```
gas | title –v"first ten integers" >N
root N >RN
root –r3 N >R3N
root –r1.5 N >R1.5N
plot –FN,g N R1.5N RN R3N | td
```

where:

- **title –v"*name*" associates a** *name* with a vector. In this case, **first ten integers** is associated with the vector output by **gas**. The vector is stored in file N.

- **root –r*n*** outputs the *n*th root of each element on the input. If –r*n* is not given, then the square root is output. Also, if the input is a titled vector, the title will be transformed to reflect the root function.

- **plot –F***X,g Y(s)* generates a multiline plot with *Y(s)* plotted versus *X*. The **g** option causes tick marks to appear instead of grid lines.

The resulting plot is shown in Figure 12-3.

The next example generates a histogram of random numbers:

```
rand –n100 | title –v"100 random numbers" | qsort |
    bucket | hist | td
```

where:

- **rand –n100** outputs random numbers using **rand**(3C). In this case 100 numbers are output in the range 0 to 1.
- **qsort** sorts the elements of a vector in ascending order.
- **bucket** breaks the range of the elements in a vector into intervals and counts how many elements from the vector fall into each interval. The output is a vector with odd elements being the interval boundaries and even elements being the counts.
- **hist** builds a histogram based on interval boundaries and counts.

The output is shown in Figure 12-4.

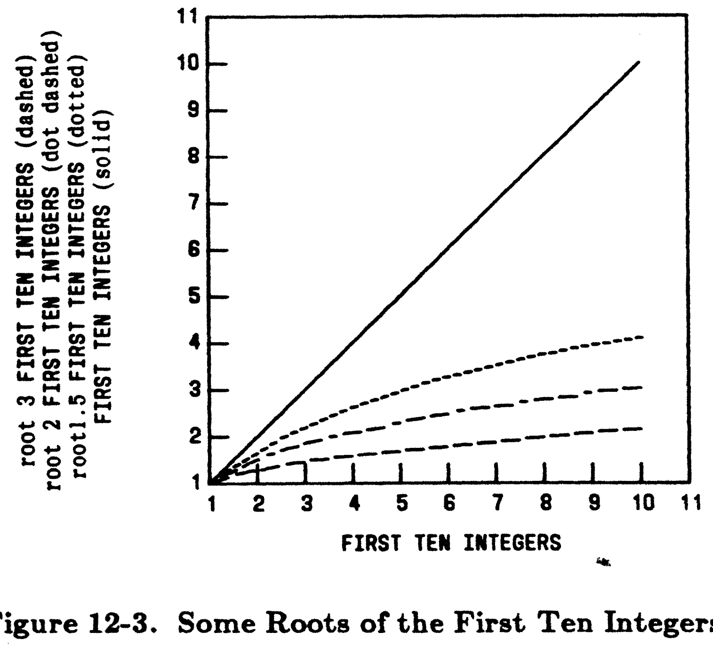


Figure 12-3. Some Roots of the First Ten Integers

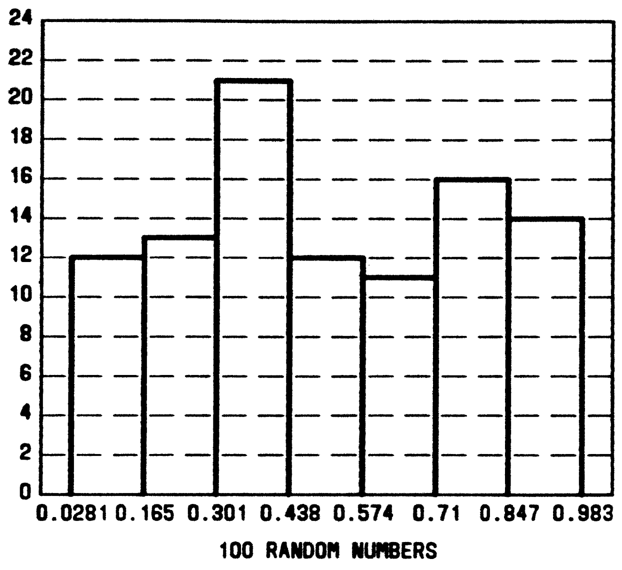


Figure 12-4. Histogram of 100 Random Numbers

Drawings Built From Boxes

There is a large class of drawings composed from boxes and text. Examples are structure charts, configuration drawings, and flow diagrams. In **graphics** the general procedure to construct such box drawings is the same as that for numerical plotting; namely, gather and transform the data, build and display the layout.

As an example, for hierarchy charts, the command line

```
dtoc | vtoc | td
```

outputs drawings representing directory structures.

- The **dtoc** command outputs a table of contents that describes a directory structure (Figure 12-5). The fields from left to right are level number, directory name, and the number of ordinary readable files contained in the directory.

- The **vtoc** command reads a (textual) table of contents and outputs a visual table of contents, or hierarchy chart (Figure 12-6). Input to **vtoc** consists of a sequence of entries, each describing a box to be drawn. An entry consists of a level number, an optional style field, a text string to be placed in the box, and a mark field to appear above the top right-hand corner of the box.

```
0.        "source"      2
1.        "glib.d"      1
1.1.      "gpl.d"       12
1.2.      "gsl.d"       14
2.        "gutil.d"     6
2.1.      "cvrtopt.d"   7
2.2.      "gtop.d"      8
2.3.      "ptog.d"      5
3.        "stat.d"      54
4.        "tek4000.d"   5
4.1       "ged.d"       37
4.4.      "td.d"        8
5.        "toc.d"       3
5.1.      "ttoc.d"      3
5.2.      "vtoc.d"      22
6.        "whatis.d"    108
```

Figure 12-5. Output of dtoc Command

Where To Go From Here

The best way to learn about **graphics** is to log onto a ICON/UXV operating system and use it. Other chapters in this guide contain tutorials for **stat**(1G) and **ged**(1G) and administrative information for **graphics**. Additional information can be found in the *ICON/UXV User Reference Manual* in the following manual entries:

gdev(1G), a collection of commands to manipulate TEKTRONIX 4000 series terminals; and

ged(1G), the graphical editor;

graphics(1G), the entry point for **graphics**;

gutil(1G), a collection of graphical utility commands;

stat(1G), numerical manipulation and plotting commands;

toc(1G), routines to build tables of contents;

gps(5), a description of a graphical primitive string.

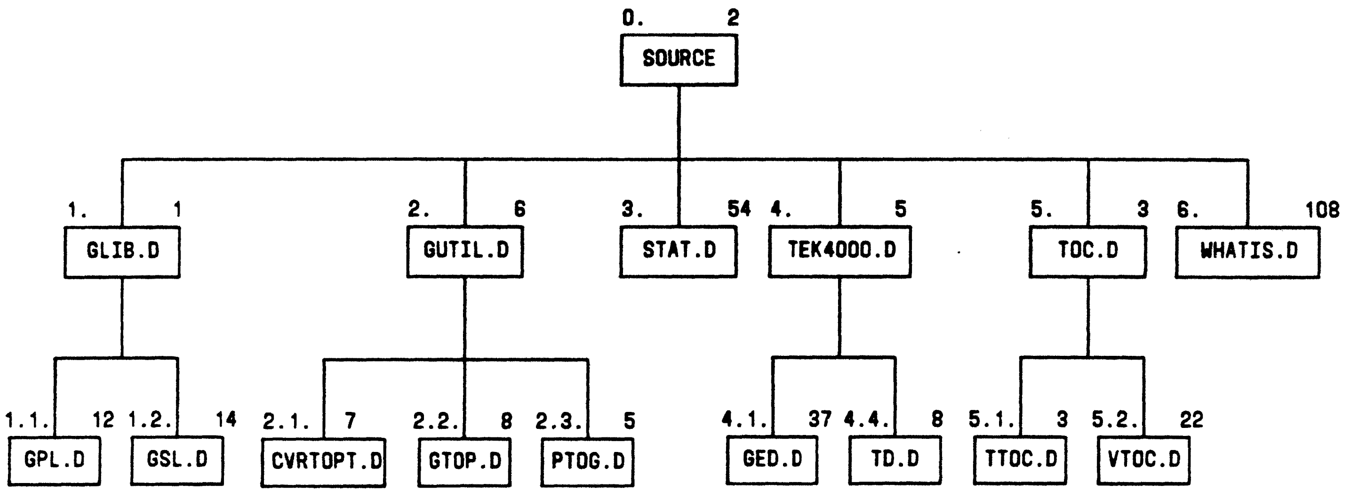


Figure 12-6. Output of vtoc Command

Chapter 13

STAT-A TOOL FOR ANALYZING DATA

PAGE

Chapter 13

STAT-A TOOL FOR ANALYZING DATA

Chapter Introduction

This chapter introduces **stat** concepts and commands through a collection of examples. Also, a complete definition of each command is provided.

Stat is a collection of numerical programs that can be interconnected using the ICON/UXV shell to form processing networks. Included within **stat** are programs to generate simple statistics and pictorial output.

Much of the power for manipulating text in the ICON/UXV operating system comes from the DOCUMENTER'S WORKBENCH* software text processing package. The general interface is an unformatted text string. The interconnection mechanism is usually the ICON/UXV shell. The programs are independent of one another, new functions can easily be added and old ones changed. Because the text editor operates on unformatted text, arbitrary text manipulation can always be performed even when the more specialized routines are insufficient.

Stat uses the same mechanisms to bring similar power to the manipulation of numbers. It consists of a collection of numerical processing routines that read and write unformatted text strings. It includes programs to build graphical files that can be manipulated using a graphical editor. And since **stat** programs process unformatted text, they can readily be connected with other ICON/UXV operating system command-level (i.e., callable from shell) routines.

It is useful to think of the shell as a tool for constructing processing networks in the sense of data flow programming. Command-level routines are the nodes of the network, and pipes and tees are the links. Data flows from node to node in the network via data links. Throughout this chapter, the operator := means *defined as*.

Basic Concepts

All numerical data in **stat** are stored in vectors. A *vector* is a sequence of numbers separated by delimiters. Vectors are processed by command-level routines called *nodes*.

* Trademark of AT&T Technologies.

Transformers

A *transformer* is a node that reads an input element, operates upon it, and outputs the resulting value. For example, suppose file **A** contains the vector

```
1 2 3 4 5
```

then the command

root A (typed input is **bold**)

produces

```
1    1.41421    1.73205    2    2.23607
```

the square root of each input element. Also,

log A

produces

```
0    0.693147    1.09861    1.38629    1.60944
```

the natural logarithm of each element of vector **A**.

af, for arithmetic function, is a particularly versatile transformer. Its argument is an expression that is evaluated once for each complete set of input values. A simple example is

af "2*A^2"

which produces

```
2       8       18       32       50
```

twice the square of each element from **A**. Expression arguments to **af** are usually surrounded by quotes since some of the operator symbols have special meaning to the shell.

Summarizers

A *summarizer* is a node that calculates a statistic for a vector. Typically, summarizers read in all of the input values, then calculate and output the statistic. For example, using the vector **A** from the previous example,

```
mean A
```

produces

```
3
```

and

```
total A
```

produces

```
15
```

Parameters

Most nodes accept parameters to direct their operation. Parameters are specified as command-line options. **root**, for example, is more general than just square root, any root may be specified using the **r** option. For example,

```
root –r3 A
```

produces

```
1    1.25992    1.44225    1.5874    1.70998
```

the cube root of each element from **A**.

Building Networks

Nodes are interconnected using the standard ICON/UXV shell concepts and syntax. A pipe is a linear connector that attaches the output of one node to the input of another. As an example, to find the mean of the cube roots of vector **A** is simply

```
root –r3 A | mean
```

which produces

```
    1.39991
```

Often the required network is not so simple. Tees and sequence can be used to build nonlinear networks. To find the mean and median of the transformed vector **A** is

```
    root -r3 A | tee B | mean; point B
```

which produces

```
    1.39991
    1.44225
```

Beware of the distinction between the sequence operator, (;), and the linear connector, the pipe (¦). Because processes in a pipeline run concurrently, each file written to in the pipeline should be unique. Sequence implies run to completion (so long as & is not used) hence files may be used more than once.

There is a special case of nonlinear networks where the result of one node is used as command-line input for another. Command substitution makes this easy. For example, to generate residuals from the mean of **A** is simply

```
    af "A-`mean A`"
```

which results in

```
    -2    -1    0    1    2
```

Vectors, a Closer Look

Thus far vectors have been used but not created. One way to create a vector is by using a generator. A *generator* is a node that accepts no input and outputs a vector based upon definable parameters. **gas** is a generator that produces additive sequences. One of the parameters to **gas** is the number of elements in the generated vector. As an example, to create the vector **A** that we have been using is

```
    gas -n5
```

which produces

```
    1    2    3    4    5
```

Vectors are, however, merely text files. Hence, the text editor can be used to create and modify the same vector.

A useful property of vectors is that they consist of a sequence of numbers surrounded by delimiters, where a delimiter is anything that is not a number. Numbers are constructed in the usual way

```
[sign](digits)(.digits)[e[sign]digits]
```

where fields are surrounded by brackets and parentheses. All fields are optional, but at least one of the fields surrounded by parentheses must be present. Thus vector **A** could also be created by building the file **B** in the text editor as

```
1partridge,2tdoves,3frhens,4cbirds,5gldnrings,
```

which, when read by

```
list B
```

produces

```
1       2       3       4       5
```

A note should be made as to the size of a vector. A vector is a stream containing numbers terminated by an end of file (control-d from the keyboard). A good illustration of this is to use the keyboard as the source of the input vector, as in

```
cusum -c1
2<cr>
2
16.3<cr>
18.3
25.4<cr>
43.7
-14<cr>
29.7
<control d>
```

which implements a running accumulator. Since no vector was given to **cusum**, the input is taken from the standard input until an end of file.

A Simple Example: Interacting with a Data Base

When used in conjunction with the ICON/UXV operating system tools for manipulating text, **stat** provides an effective means for exploring a numerical data base. Suppose, for example, there is a subdirectory called **data** containing data files that include the lines:

```
path length = nn     (nn is any number)
node count = nn
```

To access the value for **node count** from each file, sort the values into ascending order, store the resulting vector in file **A**, and get a copy on the terminal by typing

```
grep "node count" data* | qsort | tee A
17     19     22     32     39
50     68     78     125    139
```

If some of the data files have numbers in their name, we must protect those numbers from being considered data. Using **cat**, this is easy:

```
cat data/* | grep "node count" | qsort | tee A
```

To get a feel for the distribution of node counts, shell iteration can be used to advantage.

```
for i in .25 .5 .75
do point -p$i A
done
24.5
44.5
75.5
```

This generates the lower hinge, the median, and the upper hinge of the sorted vector **A**.

Translators

Translators are used to view data pictorially. A *translator* is a node that produces a stream of a different structure from that which it consumes. Graphical translators consume vectors and produce pictures in a language called GPS. Among the programs that understand GPS is **ged**, the graphical editor, which means that the graphical output of any translator can be edited at a display terminal. **hist** is an example of a translator; it produces a GPS that describes a histogram from input consisting of interval limits and counts. The summarizer **bucket** produces limits and counts, thus

```
bucket A | hist | td
```

generates a histogram of the data of vector **A** and displays it on a display terminal (Figure 13-1). **td** translates the GPS into machine code for TEKTRONIX 4010 series display terminals.

A wide range of X-Y plots can be constructed using the translator **plot**. For example, to build a scatter plot of **path length** with **node count** (Figure 13-2) is

```
grep "path length" data/* | title -v"path length" >A
grep "node count" data/* | title -v"node count" | plot
    -FA,dg | td
```

A vector may be given a title using **title**. When a titled vector is plotted, the appropriate axis is labeled with the vector title. When a titled vector is passed through a transformer, the title is altered to reflect the transformation. Thus in a graph of log **node count** versus the cube root of **path length**, i.e.,

```
grep "node count" | title -v"node count" | log >B
root -r3 A | plot -F-,dg B | td
```

the axis labels automatically agree with the vectors plotted (Figure 13-3).

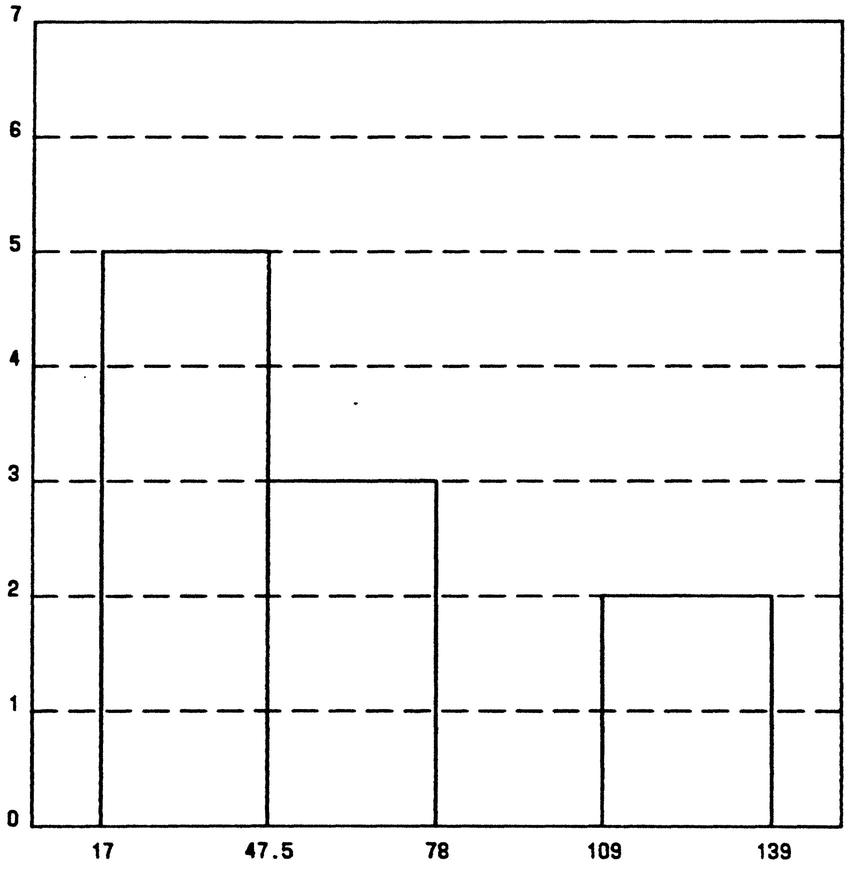


Figure 3-1. bucket A | hist | td

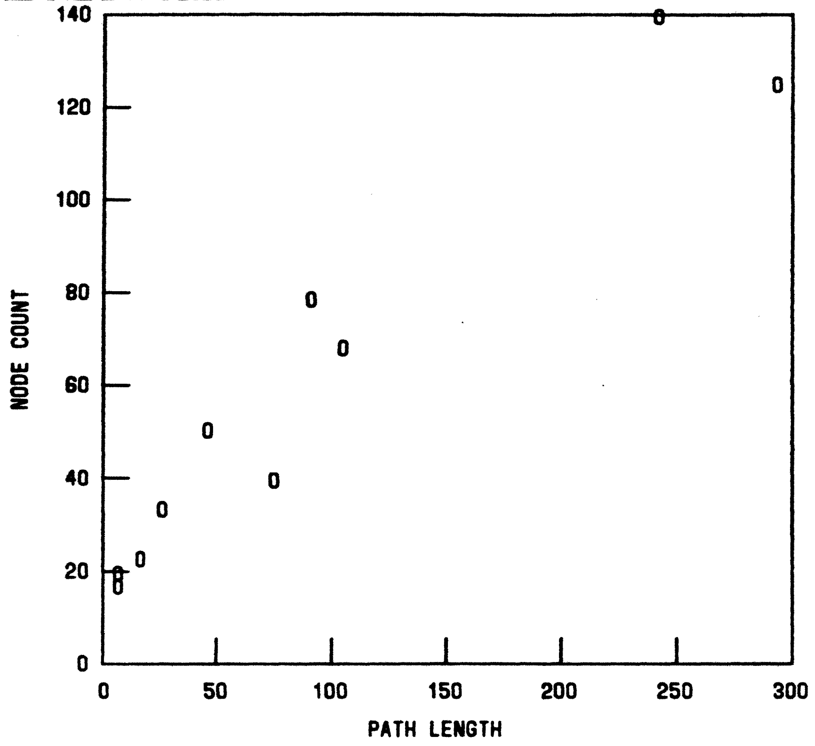


Figure 13-2. Scatter Plot

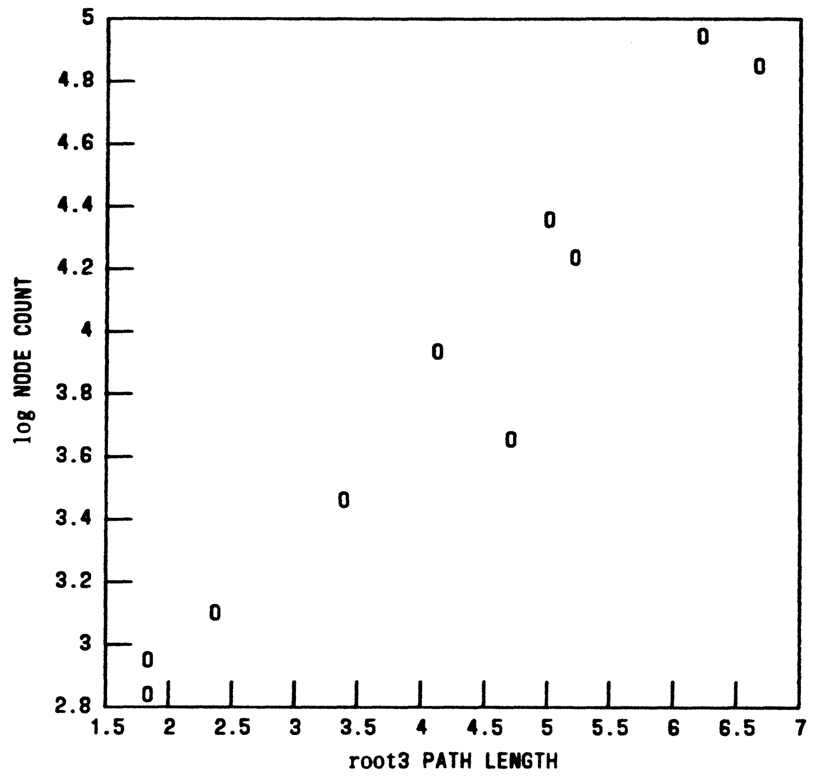


Figure 13-3. Transformed Scatter Plot

Node Descriptions

In this section a more formal description of each node is given. The mathematical formula given with each description corresponds to the algorithm implemented by the command. The descriptions are organized by node class. The **stat** nodes are divided into these four classes:

- *Transformers*
- *Summarizers*
- *Translators*
- *Generators*

All of the nodes accept the same command-line format:

- A *command* is a *command-name* followed by zero or more *arguments*.
- A *command-name* is the name of any **stat** node.

- An *argument* is a *file-name* or an *option-string*.
- An *option-string* is a — followed by one or more *options*.
- An *option* is one or more letters followed by an optional value. Options may be separated by commas.
- A *file-name* is any name not beginning with –, or a – by itself (to reference the standard input).

Each file argument to a node is taken as input to one occurrence of the node. That is, the node is executed from its initial state once per file. If no files are given, the standard input is used. All nodes, except generators, accept files as input, hence it is not made explicit in the synopses that follow.

Most nodes accept command-line *options* to direct the execution of the node. Some *options* take values. In the following synopses, to indicate the type of value associated with an *option*, the *option* key-letter is followed by:

i	to indicate integer,
f	to indicate floating point or integer,
string	to indicate a character string, or
file	to indicate a file-name.

Thus, the *option* **c***i* implies that **c** expects an integer value (**c** := integer).

Transformers

Transformers have the form

V_{in} *transform* V_{out}

where, by convention, V_{in} is a vector Y, with elements y_1 through y_k ($y_{1:k}$) and V_{out} is a vector Z, $z_{1:m}$. All transformers have a **c***i* option, where **c** specifies the number of columns per line in the output. By default, **c** := 5.

abs – absolute value

$z_i := |y_i|$

af [–**t** **v**] – arithmetic function

The command-line format of **af** is an extension of the command-line description given above, with *expression* replacing *file-name*; an *expression* consists of *operands* and *operators.*

An *operand* is either a *vector, function, constant,* or *expression:*

- A *vector* is a file name with the restriction that file names begin with a letter and are composed only of letters, digits, ".", and "_". The first unknown file name (one not in the current directory) references the standard input. A warning will appear if a file cannot be read.
- A *function* is the name of a command followed by its arguments in parentheses. Arguments are written in command-line format.
- A *constant* is an integer or floating point (but not "E" notation) number.

The *operators* are listed below in order of decreasing precedence. Parentheses may be used to alter precedence.

The x_i (y_i) represents the start element from X (Y) for the expression.

- $'Y$ — reference y_{i+1}. y_{i+1} is consumed; the next value from Y is y_{i+2}. Y is a vector.
- X ^ Y $-Y$ — x_i raised to the y_i power, negation of y_i. Association is right to left. X and Y are expressions.
- $X*Y$ X/Y $X\%Y$ — x_i multiplied by, divided by, modulo y_i. Association is left to right. X and Y are expressions.
- $X+Y$ $X-Y$ — x_i plus, minus y_i. Association is left to right. X and Y are expressions.
- X,Y — yields x_i, y_i. Association is left to right. X and Y are expressions.

Options:

t	causes the output to be titled from the vector on the standard input.
v	causes function expansions to be echoed.

ceil — ceiling

z_i := smallest integer greater than y_i

cusum — cumulative sum

$$z_i := \sum_{j=1}^{i} y_j$$

exp — exponential function

$$z_i := e^{y_i}$$

floor — floor

z_i := largest integer less than y_i

gamma — gamma function

$z_i := \Gamma(y_i)$

list [—**d***string*] — list vector elements

$z_i := y_i$

If **d** is not specified, then any character that is not part of a number is a delimiter. If **d** is specified, then the white space characters (space, tab, and new-line) plus the character(s) of *string* are delimiters. Only numbers surrounded by delimiters are listed.

log [—**b***f*] — logarithmic function

$z_i := \log_b y_i$

By default, **b** := e (e ≈2.71828...)

mod [—**m***f*] — modulus

$z_i := y_i$ modulo **m**

By default, **m** := 2

pair [—**F***file* **x***i*] — pair elements

F is a vector X, $x_{1:j}$, and **x** is
the number of elements per group from X. Let %
denote modulo and / denote integer division, then

$$z_i := \begin{cases} y_{(i/(x+1))} & \text{if } i\%(x+1) = 0 \\ x_{(i-i/(x+1))} & \text{if } i\%(x+1) \neq 0 \end{cases}$$

$rank(Z) = (x+1)\text{minimum}(k, j/x)$

If **F** is not specified, then X comes from the standard input. If both X and Y come from the standard input, X precedes Y.

By default, **x** := 1

power [−**p***f*] — raise to a power

$z_i := y_i^{\mathbf{p}}$

By default, **p** := 2

root [−**r***f*] — extract a root

$z_i := \sqrt[\mathbf{r}]{y_i}$

By default, **r** := 2

round [−**p***i* **s***i*] — round off values

if **s** is specified, then
$z_i := y_i$ rounded up to **s** significant digits,

else if **p** is specified, then
$z_i := y_i$ rounded up to **p** digits beyond the decimal point.

By default, **p** := 0

siline [−**i***f* **n***i* **s***f*] — generate a line, given a slope and intercept

$z_i = \mathbf{s}\, y_i + \mathbf{i}$

if **n** is specified, then

$Y \equiv 0, 1, 2, 3, \cdots, \mathbf{n}$.

By default, **i** := 0, **s** := 1

sin — sine function

$z_i := \sin(y_i)$

spline [−options] — interpolate smooth curve

Y and *Z* are sequences of X,Y coordinates (like that produced by **pair**).

For more information about **spline**, see **spline**(1) in the *ICON/UXV User Reference Manual.*

subset [**−a***f* **b***f* **F***file* **i***i* **l***f* **nl np p***f* **s***i* **t***i*] — generate a subset

Z consists of elements selected from *Y*. Selection occurs as follows:

Let C(w) be true if

$(w > \mathbf{a}$ or $w < \mathbf{b}$ or $w = \mathbf{p})$ and $w \neq \mathbf{l}$

is true. If neither **a**, **b**, nor **p** are specified, C(w) is true if $w \neq \mathbf{l}$ is true.

CASE 1 — **nl** or **np** not specified.

If **F** is specified, then $key_i = x_i$
else $key_i = y_i$

For $r = \mathbf{s}, \mathbf{s} + \mathbf{i}, \mathbf{s} + 2\mathbf{i}, \ldots$ with $r =\!=< \mathbf{t}$,
y_r becomes an element of *Z* if C(key_r) is true.

By default, $\mathbf{i} := 1$, $\mathbf{s} := 1$, $\mathbf{t} := 32767$.

CASE 2 — **np** is specified.

F is a vector *X*, $x_{1:j}$.
For $r = x_1, x_2, \ldots, x_j$,
y_r becomes an element of *Z* if C(y_r) is true.

CASE 3 — **nl** is specified.

F is a vector *X*, $x_{1:j}$.
For $r \neq x_1, x_2, \cdots, x_j$,
y_r becomes an element of *Z* if C(y_r) is true.

For cases 2 and 3, if **F** is not specified, then the standard input is used for *X*. Either *X* or *Y* may come from the standard input, but not both.

Summarizers

Summarizers have the form

V_{in} *summarize* V_{out}

where, again, V_{in} is a vector *Y*, $y_{1:k}$, and V_{out} is a vector *Z*, $z_{1:m}$. For many summarizers,

$rank(Z) = 1$.

bucket [**–a***i* **c***i* **F***file* **h***f* **i***i* **l***f* **n***i*] — break into buckets

Y must be a sorted vector. *Z* consists of odd elements (parenthesized) which are bucket limits and even elements which are bucket counts.

The count is the number of elements from *Y* greater than the lower limit (greater than or equal to for the lowest limit), and less than or equal to the higher limit. If specified, the limit values are taken from **F**. Otherwise the limits are evenly spaced between **l** and **h** with a total of **n** buckets. If **n** is not specified, the number of buckets is determined as follows:

$$\mathbf{n} := \begin{cases} \dfrac{\mathbf{h} - \mathbf{l}}{\mathbf{i}} & \text{if } \mathbf{i} \text{ is specified} \\ \dfrac{k}{\mathbf{a} + 1} & \text{if } \mathbf{a} \text{ is specified} \\ 1 + \log_2 k & \text{if neither } \mathbf{a} \text{ nor } \mathbf{i} \text{ are specified.} \end{cases}$$

c specifies the number of columns in the output.

By default:

c := 5
h := largest element of *Y*
l := smallest element of *Y*

cor [**–F***file*] — correlation coefficient

If **F** is a vector *X*, $x_{1:k}$,

let $\bar{x} = \dfrac{\sum_{i=1}^{k} x_i}{k}$ and

$\bar{y} = \dfrac{\sum_{i=1}^{k} y_i}{k}$, then

$$z_1 := \frac{\sum_{i=1}^{k}(x_i-\bar{x})(y_i-\bar{y})}{\left[\sum_{i=1}^{k}(x_i-\bar{x})^2\right]^{1/2}\left[\sum_{i=1}^{k}(y_i-\bar{y})^2\right]^{1/2}}$$

X and *Y* must have the same rank. If **F** is not specified, the standard input is used for *X*. If both *X* and *Y* come from the standard input, *X* precedes *Y*.

hilo [**—h l o ox oy**] — high and low values

z_1 := lowest value across all input vectors

z_2 := highest value across all input vectors

Options to control output:

h	Only output high value.
l	Only output low value.
o	Output high, low values in option form (suitable for **plot**).
ox	Output high, low values with "x" prefixed.
oy	Output high, low values with "y" prefixed.

lreg [**—F***file* **i o s**] — linear regression

If **F** is a vector X, $x_{1:k}$, let $\bar{x} = \frac{\sum_{i=1}^{k} x_i}{k}$ and $\bar{y} = \frac{\sum_{i=1}^{k} y_i}{k}$, then

$$z_1 := \bar{y} - z_2\bar{x} \qquad \text{(intercept)}$$

and

$$z_2 := \frac{\frac{\sum_{i=1}^{k} x_i y_i}{k} - \bar{x}\,\bar{y}}{\frac{\sum_{i=1}^{k} x_i^2}{k} - \bar{x}^2} \qquad \text{(slope)}$$

X and Y must have the same rank.
If **F** is not specified, then

$$X \equiv 0, 1, 2, \cdots, k$$

Options to control output:

i	Only output the intercept.
o	Output the slope and intercept in option form (suitable for **siline**).
s	Only output the slope.

mean [−**f***f* **n***i* **p***f*] – (trimmed) mean

$$z_1 := \frac{\sum_{i=1}^{k} y_i}{k}$$

Y may be trimmed by

(1/**f**)k	elements from each end,
p k	elements from each end, or
n	elements from each end.

By default, **n** :=0

point [−**f***f* **n***i* **p***f* **s**] – empirical cumulative density function point

z_1 := linearly interpolated *Y* value corresponding to the

100(1/**f**)	percent point, the
100**p**	percent point, or the
nth	element.

Negative option values are taken from the high end of *Y*. Option **s** implies *Y* is sorted.

By default, **p** := .5 (median)

prod – product

$$z_1 := \prod_{i=1}^{k} y_i$$

qsort [−**c***i*] – quicksort

z_i := *i*th smallest element of *Y*.

By default, **c** := 5

rank – rank

z_1 := number of elements in *Y*.

total – sum

$$z_1 := \sum_{i=1}^{k} y_i$$

var — variance

$$z_1 := \frac{\sum_{i=1}^{k}(y_i - \bar{y})^2}{k - 1}$$

Translators

Translators have the form

F_{in} translate F_{out}

where F_{in} may be a vector or a GPS depending upon the translator. F_{out} is a GPS. A GPS is a format for storing a picture. A picture is defined in a Cartesian plane of 64K points on each axis. The plane, or universe, is divided into 25 square regions numbered 1 to 25 from the lower left to the upper right. Various commands exist that can display and edit a GPS. For more information, see **graphics**(1) in the *ICON/UXV User Reference Manual* and in this manual.

bar [**–a b f g r*i* w*i* x*f* xa y*f* ya yl*f* yh*f***] — build a bar chart

F_{in} is a vector, each element of which defines the height of a bar. By default, the x-axis will be labeled with positive integers beginning at 1; for other labels, see **label**.

Options:

a	Suppress axes.
b	Plot bar chart with bold weight lines, otherwise use medium.
f	Do not build a frame around plot area.
g	Suppress background grid.
r*i*	Put the bar chart in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
w*i*	*i* is the ratio of the bar width to center-to-center spacing expressed as a percentage. Default is 50, giving equal bar width and bar space.
x*f* (y*f*)	Position the bar chart in the GPS universe with x-origin (y-origin) at *f*.
xa (ya)	Do not label x-axis (y-axis).
yl*f*	*f* is the y-axis low tick value.
yh*f*	*f* is the y-axis high tick value.

hist [**–a b f g r***i* **x***f* **xa y***f* **ya yl***f* **yh***f*] – build a histogram

F_{in} is a vector (of the type produced by **bucket**) of odd rank, with odd elements being limits and even elements being bucket counts.

Options:

a	Suppress axes.
b	Plot histogram with bold weight lines, otherwise use medium.
f	Do not build a frame around plot area.
g	Suppress background grid.
r*i*	Put the histogram in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
x*f* (**y***f*)	Position the histogram in the GPS universe with x-origin (y-origin) at *f*.
xa (**ya**)	Do not label x-axis (y-axis).
yl*f*	*f* is the y-axis low tick value.
yh*f*	*f* is the y-axis high tick value.

label [**–b c F***file* **h p r***i* **x xu y yr**] – label the axis of a GPS file

F_{in} is a GPS of a data plot (like that produced by **hist, bar,** and **plot**). Each line of the label *file* is taken as one label. Blank lines yield null labels. Either the GPS or the label *file*, but not both, may come from the standard input.

Options:

b	Assume the input is a bar chart.
c	Retain lower case letters in labels, otherwise all letters are upper case.
F*file*	*file* is the label *file*.
h	Assume the input is a histogram.
p	Assume the input is an x-y plot. This is the default.
r*i*	Labels are rotated *i* degrees. The pivot point is the first character.
x	Label the x-axis. This is the default.
xu	Label the upper x-axis, i.e., the top of the plot.
y	Label the y-axis.
yr	Label the right y-axis, i.e., the right side of the plot.

pie [**−b o p pn***i* **pp***i* **r***i* **v x***i* **y***i*] − build a pie chart

F_{in} is a vector with a restricted format. Each input line represents a slice of pie and is of the form:

[< **i e f c***color* >] value [label]

with brackets indicating optional fields. The control field options have the following effect:

i	The slice will not be drawn, though a space will be left for it.
e	The slice is "exploded" or moved away from the pie.
f	The slice is filled. The angle of fill lines depends on the color of the slice.
c*color*	The slice is drawn in *color* rather than the default black. Legal values for *color* are **b** for black, **r** for red, **g** for green, and **u** for blue.

The pie is drawn with the value of each slice printed inside and the label printed outside.

Options:

b	Draw pie chart in bold weight lines, otherwise use medium.
o	Output values around the outside of the pie.
p	Output value as a percentage of the total pie.
pn*i*	Output value as a percentage, but total of percentages equals *i* rather than 100. The option **pn**100 is equivalent to **p**.
pp*i*	Only draw *i* percent of a pie.
r*i*	Put the pie chart in region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
v	Do not output values.
x*i* (**y***i*)	Position the pie chart in the GPS universe with x-origin (y-origin) at *i*.

plot [**−a b c***string* **d f F***file* **g m r***i* **x***f* **xa xh***f* **xi***f* **xl***f* **xn***i* **xt**
y*f* **ya yh***f* **yi***f* **yl***f* **yn***i* **yt**] − plot a graph

F_{in} is a vector(s) which contains the y values of an x-y graph. Values for the x-axis come from **F**. Axis scales are determined from the first vector plotted.

Options:

a	Suppress axes.
b	Plot graph with bold weight lines, otherwise use medium.
c*string*	The character(s) of *string* are used to mark points. Characters from *string* are used, in order, for each separately plotted graph included in the plot. If the number of characters in *string* is less than the number of plots, the last character will be used for all remaining plots. The **m** option is implied.
d	Do not connect plotted points, implies option **m**.

f	Do not build a frame around plot area.
F*file*	Use *file* for x-values, otherwise the positive integers are used. This *option* may be used more than once, causing a different set of x-values to be paired with each input vector. If there are more input vectors than sets of x-values, the last set applies to the remaining vectors.
g	Suppress the background grid.
m	Mark the plotted points.
r*i*	Put the graph in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
x*f* (**y***f*)	Position the graph in the GPS universe with x-origin (y-origin) at *f*.
xa (**ya**)	Omit x-axis (y-axis) labels.
xh*f* (**yh***f*)	*f* is the x-axis (y-axis) high tick value.
xi*f* (**yi***f*)	*f* is the x-axis (y-axis) tick increment.
xl*f* (**yl***f*)	*f* is the x-axis (y-axis) low tick value.
xn*i* (**yn***i*)	*i* is the approximate number of ticks on the x-axis (y-axis).
xt (**yt**)	Omit x-axis (y-axis) title.

title [**–b c l***string* **v***string* **u***string*] – title a vector or GPS

F_{in} can be either a GPS or a vector with F_{out} being of the same type as F_{in}. **Title** prefixes a **title** to a vector or appends a **title** to a GPS.

Options apply as indicated:

b	Make the GPS **title** bold.
c	Retain lower case letters in **title**, otherwise all letters are upper case.
l*string*	For a GPS, generate a lower **title** := *string*.
u*string*	For a GPS, generate an upper **title** := *string*.
v*string*	For a vector, **title** := *string*.

Generators

Generators have the form

generate V_{out}

where V_{out} is a vector Z, $z_{1:k}$. All generators have a **c***i* option where **c** specifies the number of columns per line in the output. By default, **c** := 5.

gas [**–i***f* **n***i* **s***f* **t***f*] – generate additive sequence

Z is constructed as follows:

$$z_1 := \mathbf{s}$$

$$z_{i+1} := \begin{cases} z_i+\mathbf{i} & \text{if } |z_i| =\!=\!< \mathbf{t} \\ z_1 & \text{otherwise} \end{cases}$$

$rank(Z) = \mathbf{n}$.

By default, **i** := 1, **n** := 10, **s** := 1, **t** := ∞

prime [−**h***i* **l***i* **n***i*] – generate prime numbers

The elements of *Z* are consecutive prime numbers with

$$\mathbf{l} =\!=\!< z_i =\!=\!< \mathbf{h}$$

$rank(Z) =\!=\!< \mathbf{n}$.

By default, **n** := 10, **l** := 2, **h** := ∞.

rand [−**h***f* **l***f* **m***f* **n***i* **s***i*] – generate random sequence

The elements of *Z* are random numbers generated by a multiplicative congruential generator with **s** acting as a seed, such that

$$\mathbf{l} =\!=\!< z_i =\!=\!< \mathbf{h}$$

If **m** is specified, then

$$\mathbf{h} = \mathbf{m} + \mathbf{l}$$

$rank(Z) = \mathbf{n}$.

By default, **h** := 1, **l** := 0, **n** := 10, **s** := 1.

Examples

Example 1:

PROBLEM

Calculate the total value of an investment held for a number of years at an interest rate compounded annually.

SOLUTION

```
Principal=1000
echo Total return on $Principal units compounded annually
echo "rates:\t\t\c"; gas -s.05,t.15,i.03 | tee rate
for Years in 1 3 5 8
do
   echo "$Years year(s):\t\c"; af "$Principal*(1+rate)^$Years"
done
```

```
Total return on 1000 units compounded annually
rates:          0.05            0.08            0.11            0.14
1 year(s):      1050            1080            1110            1140
3 year(s):      1157.62         1259.71         1367.63         1481.54
5 year(s):      1276.28         1469.33         1685.06         1925.41
8 year(s):      1477.46         1850.93         2304.54         2852.59
```

NOTES

Notice the distinction between vectors and constants as operands in the expression to **af**. The shell variables $*Principal* and $*Years* are constants to **af**, while the file *rate* is a vector. **af** executes the expression once per element in *rate*.

Example 2:

PROBLEM

Given are three ordered vectors (**A**, **B**, and **C**) of scores from a number of tests. Each vector is from one test-taker, each element in a vector is the score on one test. There are missing scores in each vector indicated by the value −1. Generate three new vectors containing scores only for those tests where no data is missing.

SOLUTION

```
echo Before:
gas −n`rank A` | tee N | af "label,A,B,C"

for i in N B C A
do subset −FA,l−1 $i >s$i; done
for i in N A C B
do subset −FsB,l−1 s$i | yoo s$i; done
for i in N A B C
do subset −FsC,l−1 s$i | yoo s$i; done

echo "\nAfter:"
af "sN,sA,sB,sC"
```

```
Before:
1         5         6         −1
2         7         10        10
3         −1        10        9
4         10        −1        8
5         6         5         −1
6         5         7         5
7         −1        7         8
8         −1        −1        8
9         3         −1        8
10        6         10        10
11        7         5         7


After:
2         7         10        10
6         5         7         5
10        6         10        10
11        7         5         7
```

NOTES

The approach is to eliminate those elements in all vectors that correspond to —1 in the base vector. Each of the three vectors takes a turn at being the base. It is important that the base be subsetted last. The command **yoo** (see **gutil**(1) in the *ICON/UXV User Reference Manual*) takes the output of a pipeline and copies it into one of the files used in the pipeline. This cannot be done by redirecting the output of the pipeline as this would cause a concurrent read and write on the same file.

The printing of the "Before" matrix illustrates a useful property of **af**. The first name in an expression that does not match any name in the present working directory is a reference to the standard input. In this example, **label** references the input coming through the pipe.

Example 3:

PROBLEM

Generate a bar chart of the percent of execution time consumed by each routine in a program.

SOLUTION

```
prof | cut −c1−15 | sed −e 1d −e "/ 0.0/d" −e "s/^ *//" >P
echo These are the execution percentages; cat P
title P −v"execution time in percent" | bar −xa −yl0,
        yh100 | label −br−45,FP | td
```

```
These are the execution percentages
_fork  32.9
_creat  14.3
_sbrk  14.3
_read  14.3
_open  14.3
_prime  9.9
```

NOTES

prof is a ICON/UXV operating system command that generates a listing of execution times for a program (see **prof**(1)). **Cut** and **sed** are used to eliminate extraneous text from the output of **prof**. (It is because verbiage can get in the way that **stat** nodes say very little.) Notice that **P** is a vector to **title** while it is a text file to **cat** and **label**.

Figure 13-4 shows the output of these commands.

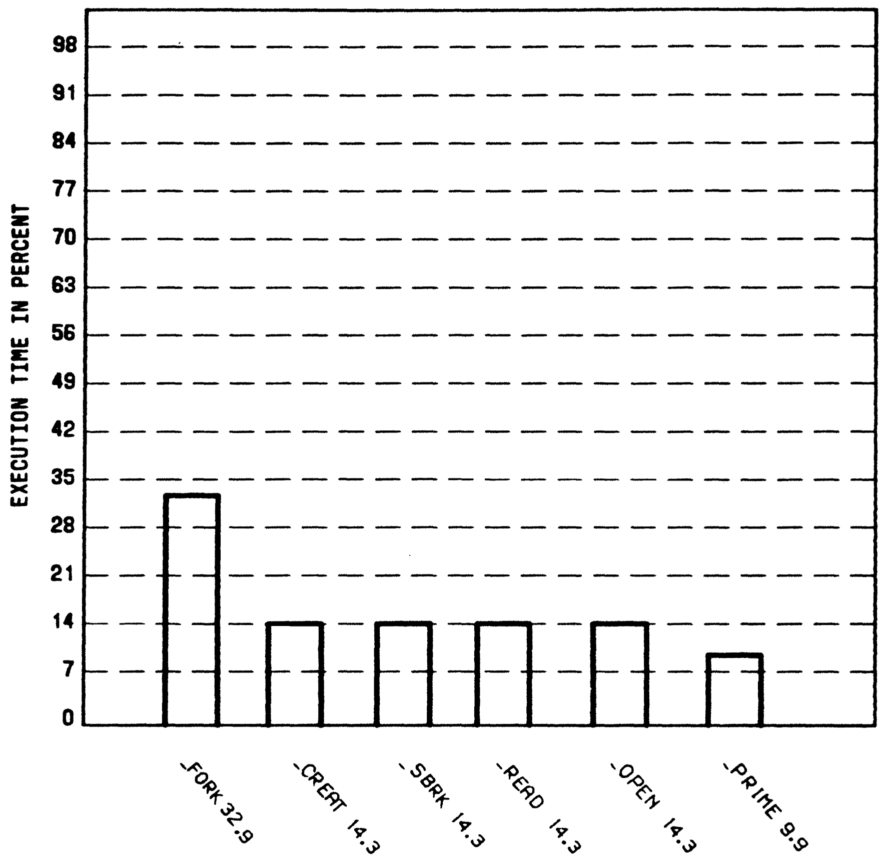


Figure 13-4. Bar Chart Showing Execution Profile

Example 4:

PROBLEM

Plot the relationship between the execution time of a program and the number of processes in the process table.

SOLUTION

```
# The first program generates the performance data

for i in `gas –n12`
do
        ps –ae | wc –l >>Procs&
        time prime –n1000 >/dev/null 2>>Times
        sleep 300
done

# The second program analyzes and plots the data

for i in real user sys
do
grep $i Times | sed "s/$i//" |
    awk –F: "{ if(NF==2) print \$1*60+\$2; else
        print }" | title –v"$i time in seconds" >$i
    siline –` lreg –o,FProcs $i ` Procs >$i.fit
done
title –v"number of processes" Procs | yoo Procs

plot –dg,FProcs real –r12 >R12
plot –ag,FProcs real.fit –r12 >>R12
plot –dg,FProcs sys –r13 >R13
plot –ag,FProcs sys.fit –r13 >>R13
plot –dg,FProcs user –r8 >R8
plot –ag,FProcs user.fit –r8 >>R8
ged R12 R13 R8
```

NOTES

The performance data is the execution time, as reported by the ICON/UXV operating system **time** command, to generate the first 1000 prime numbers. **Times** outputs three times for each run:

- The time in system routines
- The time in user routines

- Total real time.

The output of the **time** command is saved in the file **Times**. Each of these types of time is treated separately by the analysis program.

In the file **Procs** are the number of processes running on the system during each execution of **prime**. The short **awk** program converts "minutes:seconds" format to "seconds." **lreg** does a linear regression of the time vectors on the size of the process table. **siline** generates a line based on the parameters from the regression. One plot is generated for each type of time. Each plot is put into a different region so that they can be displayed and manipulated simultaneously in **ged**.

Figure 13-5 shows the output of these commands.

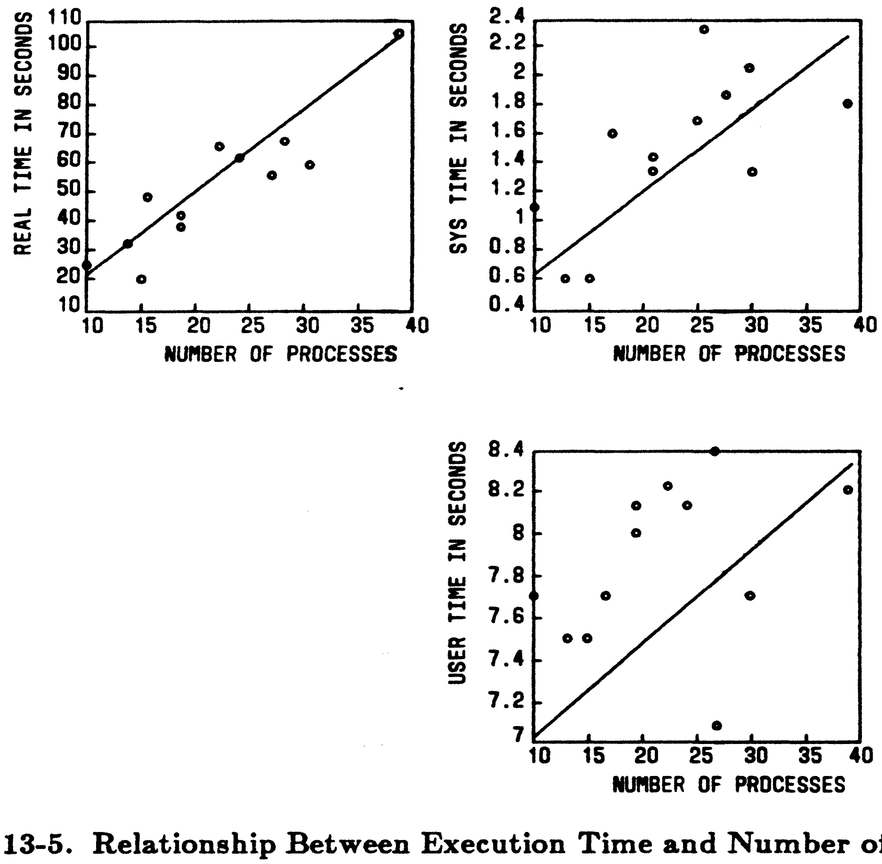


Figure 13-5. Relationship Between Execution Time and Number of Processes

Chapter 14

GRAPHICS EDITOR

PAGE

Chapter 14

GRAPHICS EDITOR

Chapter Introduction

The graphics editor, (**ged**), is an interactive graphical editor used to display, edit, and construct drawings on TEKTRONIX 4010 series display terminals. The drawings are represented as a sequence of objects in a token language known as GPS (graphical primitive string). A GPS is produced by the drawing commands in the ICON/UXV Graphics such as **vtoc** and **plot**, as well as by **ged** itself.

Drawings are built from objects consisting of lines, arcs, and text. Using the editor, the objects can be viewed at various magnifications and from various locations. Objects can be created, deleted, moved, copied, rotated, scaled, and modified.

The examples in this tutorial illustrate how to construct and edit simple drawings. Try them to become familiar with how the editor works, but keep in mind that **ged** is intended primarily to edit the output of other programs rather than to construct drawings from scratch.

As for notation, literal keystrokes are printed in **boldface**. Meta-characters are also in boldface and are surrounded by angled brackets. For example, **<cr>** means return and **<sp>** means space. In the examples, output from the terminal is printed in Roman (normal) type. In-line comments are in Roman and are surrounded by parentheses. Section 2 contains an introduction to the commands understood by the **ged**. A summary of these editor commands and options is given in Section 3 under Command Summary.

Commands

To start, it is assumed that while logged in at a display terminal the **graphics** environment (as described in **graphics**(1G) in the *ICON/UXV User Reference Manual*) has been successfully entered.

> **Note**: In order for **ged** to work properly, the standard strap options need to be set on the terminal. See Section 5.2 under Administrative Information for these standard settings.

To enter **ged** type:

ged<cr>

After a moment the screen should be clear except for the **ged** prompt, *, in the upper left corner. The * indicates that **ged** is ready to accept a command.

Each command passes through a sequence of stages during which you describe what the command is to do. All commands pass through a subset of these stages:

1. Command line
2. Text
3. Points
4. Pivot
5. Destination

As a rule, each stage is terminated by typing **<cr>**. The **<cr>** for the last stage of a command triggers execution.

Command Line

The simplest commands consist only of a *command line.* The command line is modeled after a conventional command line in the shell. That is

command name [*—option*(s)] [*filename*]**<cr>**

A question mark (**?**) is an example of a simple command. It lists the commands and options understood by **ged**. To generate the list, type

***?<cr>** (type a question mark followed by a return)

A command is executed by typing the first character of its name. The **ged** will echo the full name and wait for the rest of the command line. For example, **e** references the **erase** command. As **erase** consists only of stage 1, typing **<cr>** causes the erase action to occur. Typing

***<rubout>**

after a command name and before the final **<cr>** for the command aborts the command. Thus, while

*erase**<cr>**

erases the display screen,

*erase**<rubout>**

brings the editor back to the **ged** prompt, *.

Following the command name, *options* may be entered. Options control such things as the width and style of lines to be drawn or the size and orientation of text. Most options have a default value that applies if a value for the option is not specified on the command line. The **set** command allows examination and modification that the default values. Type

*set**<cr>**

to see the current default values.

The option value is one of three types: integer, character, or Boolean. Boolean values are represented by + (for true) and − (for false). A default value is modified by providing it as an option to the **set** command. For example, to change the default text height to 300 units, type

*set **−h300<cr>**

Arguments on the command line, but not the command name, may be edited using the erase and kill characters from the shell. This applies whenever text is being entered.

Constructing Graphical Objects

Drawings are stored as a GPS in a display buffer internal to the editor. Typically, a drawing in **ged** is composed of instances of three graphical primitives: *arcs*, *lines*, and *text*.

Generating Text

To put a line of text on the display screen use the **Text** command.

First enter the *command line* (stage 1):

```
*Text<cr>
```

Next enter the text (stage 2):

```
a line of text<cr>
```

And then enter the starting *point* for the text (stage 3).

```
<position cursor><cr>
```

Positioning of the graphic cursor is done either with the thumbwheel knobs on the terminal keyboard or with an auxiliary joystick. The **<cr>** establishes the location of the cursor to be the starting point for the text string. The **Text** command ends at stage 3, so this **<cr>** initiates the drawing of the text string.

The **Text** command accepts options to vary the angle, height, and line width of the characters, and to either center or right justify the text object. The text string may span more than one line by escaping the **<cr>** (i.e., \**<cr>**) to indicate continuation. To illustrate some of these capabilities, try the following:

```
*Text -r<cr>               (right justify text)
top\<cr>
right<cr>
<position cursor><cr>
*Text -a90<cr>     (rotate text 90 degrees)
lower\<cr>
left<cr>
<position cursor><cr>   (pick a point below and left of
                          the previous point)
```

Results of these commands are shown in Figure 14-1.

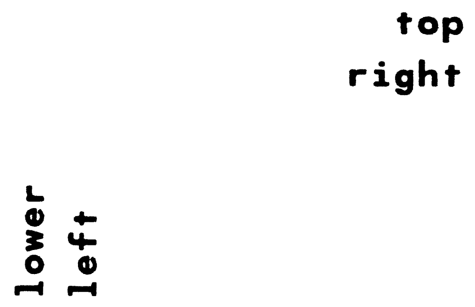


Figure 14-1. Generating Text Objects

Drawing Lines

The Lines command is used to construct objects built from a sequence of straight lines. It consists of stages 1 and 3. Stage 1 is straightforward:

```
*Lines options<cr>
```

The Lines command accepts options to specify line style and line width.

Stage 3, the entering of *points*, is more interesting. *Points* are referenced either with the graphic cursor or by name. We have already entered a point with the cursor for the Text command. For the Lines command it is more of the same. As an example, to build a triangle, type

```
*Lines<cr>
<position cursor><sp>   (locate the first point)
<position cursor><sp>   (the second point)
<position cursor><sp>   (the third point)
<position cursor><sp>   (back to the first point)
<cr>                (terminate points, draw triangle)
```

Results of these commands are shown in Figure 14-2.

Typing **<sp>** enters the location of the crosshairs as a point. **Ged** identifies the point with an integer and adds the location to the current *point set.* The last point entered can be erased by typing #. The current point set can be cleared by typing @. On receiving the final **<cr>** the points are connected in numerical order.

Accessing Points by Name

The points in the current point set may be referenced by name using the $ operator. For instance, $*n* references the point numbered *n*. By using $ the triangle above can be redrawn by entering:

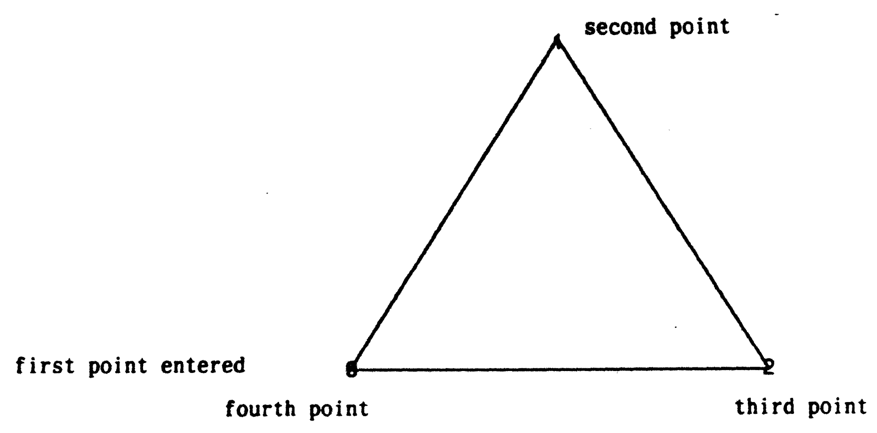


Figure 14-2. Building a Triangle

```
*Lines<cr>
<position cursor><sp>
<position cursor><sp>
<position cursor><sp>
$0<cr>              (reference point 0)
<cr>
```

At the start of each command that includes stage 3, *points*, the current point set is empty. The point set from the previous command is saved and is accessible using the . operator. The . swaps the points in the previous point set with those in the current set. The = operator can be used to identify the current points. To illustrate, use the triangle just entered as the basis for drawing a quadrilateral.

```
*Lines<cr>
.      (access the previous set)
=      (identify the current points)
#      (erase the last point)
<position cursor><sp>  (add a new point)
$0<cr>       (close the figure)
<cr>
```

Results of these commands are shown in Figure 14-3.

Individual points from the previous point set can be referenced by using the . operator with $. The following example builds a triangle that shares an edge with the quadrilateral.

*Lines**<cr>**
$.1<cr> (reference point 1 from the previous point set)
$.2<cr> (reference point 2)
<position cursor><sp> (enter a new point)
$0<cr>** (or **$.1, to close the figure)
<cr>

Results of these points are shown in Figure 14-4.

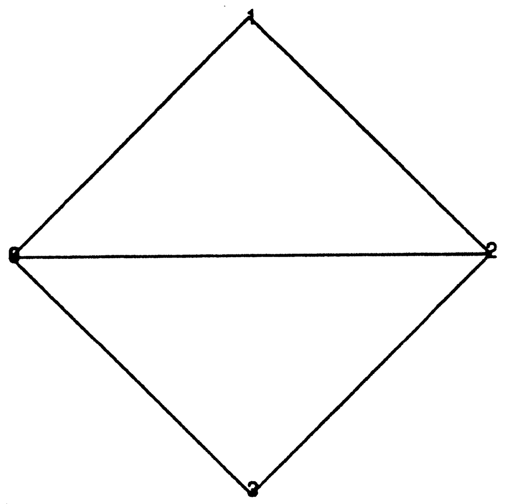


Figure 14-3. Accessing the Previous Point Set

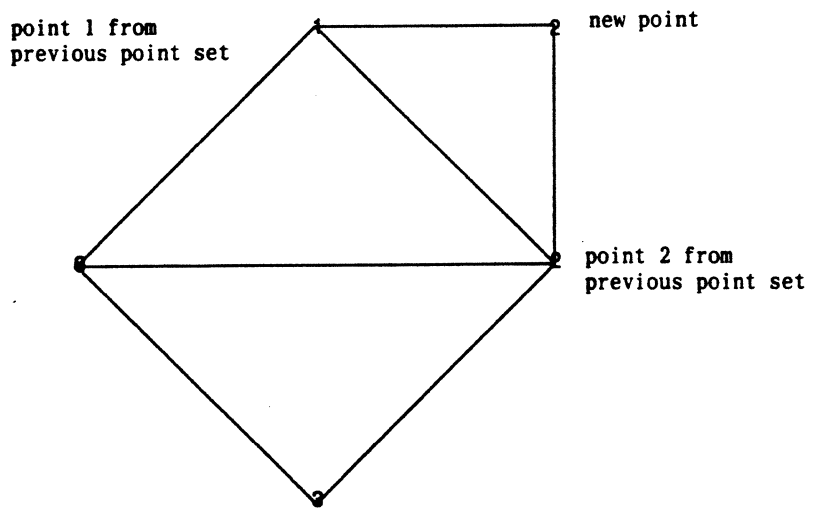


Figure 14-4. Referencing Points from Previous Point Set

A point can also be given a name. The > operator permits an upper case letter to be associated with a point just entered. A simple example is:

```
*Lines<cr>
<position cursor><sp>   (enter a point)
>A<cr>                  (name the point A)
<position cursor><sp>
<cr>
```

In commands that follow, point **A** can be referenced using the **$** operator, as in:

```
*Lines<cr>
$A<cr>
<position cursor><sp>
<cr>
```

Drawing Curves

Curves are interpolated from a sequence of three or more points. The **Arc** command generates a circular arc given three points on a circle. The arc is drawn starting at the first point, through the second point, and ending at the third point. A circle is an arc with the first and third points coincident. One way to draw a circle is thus:

```
*Arc<cr>
<position cursor><sp>
<position cursor><sp>
$0<cr>
<cr>
```

Editing Objects

Addressing Objects An object is addressed by pointing to one of its *handles*. All objects have an *object-handle*. Usually the object-handle is the first point entered when the object was created. The objects command marks the location of each object-handle with an **O**. For example, to see the handles of all the objects on the screen, type

```
*objects –v<cr>
```

Some objects, Lines for example, also have *point-handles*. Typically each of the points entered when an object is constructed becomes a point-handle. (An object-handle is also a point-handle.) The points command marks each of the point-handles.

A handle is pointed to by including it within a *defined-area*. A defined-area is generated either with a command line option or interactively using the graphic cursor. As an example, to delete one of the objects that was created on the screen, type

```
*Delete<cr>
<position cursor><sp>  (above and to the left of some
                       object-handle)
<position cursor><sp>  (below and to the right of the
                       object-handle)
<cr>                   (the defined-area should include the
                       object-handle)
<cr>                   (if all is well, delete the object)
```

The defined-area is outlined with dotted lines. The reason for the seemingly extra **<cr>** at the end of the **D**elete command is to provide an opportunity to stop the command (using **<rubout>**) if the defined-area is not quite right. Every command that accepts a defined-area will wait for a confirming **<cr>**. The **n**ew command can be used to get a fresh copy of the remaining objects.

Defined-areas are entered as **p**oints in the same way that objects are created. Actually, a defined-area may be generated by giving anywhere from 0 to 30 points. Inputting zero points is particularly useful to point to a single handle. It creates a small defined-area about the location of the terminating **<cr>**. Using a zero point defined-area, the **D**elete command would be

```
*Delete<cr>
<position cursor> (center crosshairs on the object-handle)
<cr> (terminate the defined-area)
<cr> (delete the object)
```

A defined-area can also be given as a command line option. For example, to delete everything in the display buffer give the **u**niverse option (**u**) to the **D**elete command. Note the difference between the commands **D**elete **—u**niverse and **e**rase.

Changing the Location of an Object Objects are moved using the **M**ove command. Create a circle using **A**rc, then move it as follows:

```
*Move<cr>
<position cursor><cr>  (centered on the object-handle)
<cr>          (this establishes a pivot, marked with
              an asterisk)
<position cursor><cr>  (this establishes a destination)
```

The basic move operation relocates every point in each object within the defined area by the distance from the *pivot* to the *destination*. In this case, the pivot was chosen to be the object-handle, so effectively the object-handle was moved to the destination point.

Changing the Shape of an Object The **Box** command is a special case of generating lines. Given two points, it creates a rectangle such that the two points are at opposite corners. The sides of the rectangle lie parallel to the edges of the screen. To draw a box, type

```
*Box<cr>
<position cursor><sp>
<position cursor><cr>
```

The **Box** command generates point-handles at each vertex of the rectangle. Use the points command to mark the point-handles. The shape of an object can be altered by moving point-handles. The next example illustrates one way to double the height of a box (shown in Figure 14-5).

```
*Move -p+<cr>
<position cursor><sp>    (left of the box, between the
                         top and bottom edges)
<position cursor><cr>    (right of the box, below the
                         bottom edge)
<position cursor><cr>    (on the top edge)
<position cursor><cr>    (directly below on the bottom
                         edge)
```

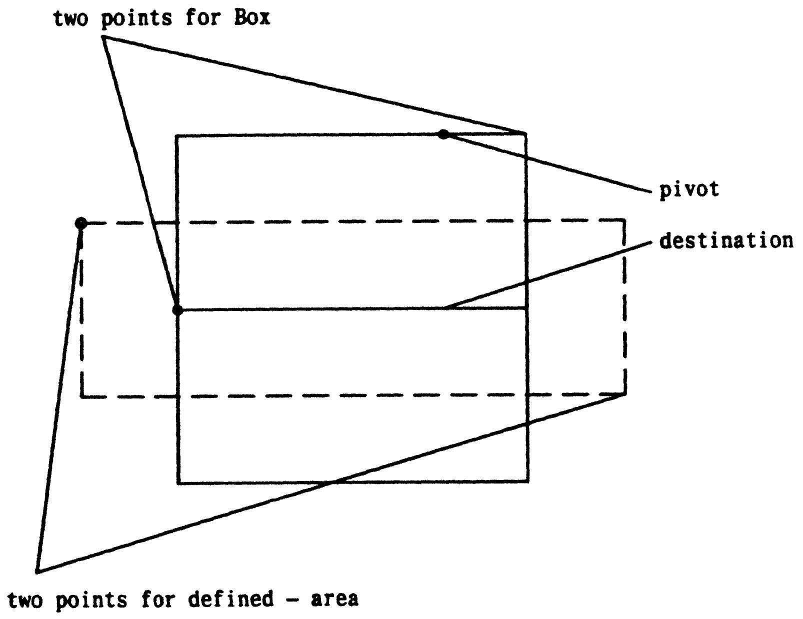


Figure 14-5. Growing a Box

When the points flag (**p**) is true, operations are applied to each point-handle addressed. In this example, the points flag was set to true using the command-line option **—p+** causing each point-handle within the defined-area to be moved the distance from the pivot to the destination. If **p** was false, only the object-handle would have been addressed.

Changing the Size of an Object The size of an object can be changed using the Scale command. The Scale command scales objects by changing the distance from each handle of the object to the pivot by a factor. Put a line of text on the screen and try the following Scale commands (Figure 14-6).

```
*Scale —f200<cr>              (factor is in percent)
<position cursor><cr>         (point to object-handle)
<position cursor><cr>         (set pivot to rightmost character)
<cr>

*Scale —f50<cr>
.<cr> (reference the previous defined-area)
<position cursor><cr>         (set pivot above a character
                               near the middle)
<cr>
```

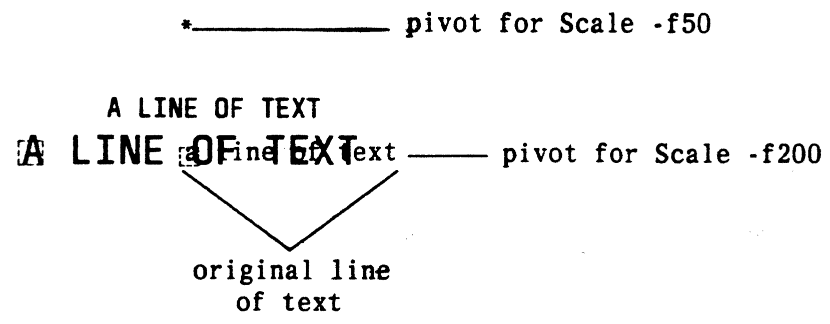


Figure 14-6. Scaling Text

A useful insight into the behavior of scaling is to note that the position of the pivot does not change. Also observe that the defined-area is scaled to preserve its relationship to the graphical objects.

The size of objects can also be changed by moving point-handles. Generate a circle, this time using the Circle command:

```
*Circle<cr>
<position cursor><sp>   (specify the center)
<position cursor><cr>   (specify a point on the circle)
```

The **Circle** command generates an arc with the first and third point at the point specified on the circle. The second point of the arc is located 180 degrees around the circle. One way to change the size of the circle is to move one of the point-handles (using **Move −p+**).

The size of text characters can be changed via a third mechanism. Character height is a property of a line of text. The **Edit** command allows changing character height as follows:

```
*Edit −hheight<cr> (height is in universe units,
                    see Section 2.8 View Command)
<position cursor><cr>   (point to the object-handle)
<cr>
```

Changing the Orientation of an Object The orientation of an object can be altered using the **Rotate** command. The **Rotate** command rotates each point of an object about a pivot by an angle. Try the following rotations on a line of text (Figure 14-7).

```
*Rotate −a90<cr>         (angle is in degrees)
<position cursor><cr>    (point to object-handle)
<position cursor><cr>    (set pivot to rightmost
                         character)
<cr>

*Rotate −a−90<cr>
.<cr> (reference previous defined-area)
<position cursor><cr>    (set pivot to a character near
                         the middle)
<cr>
```

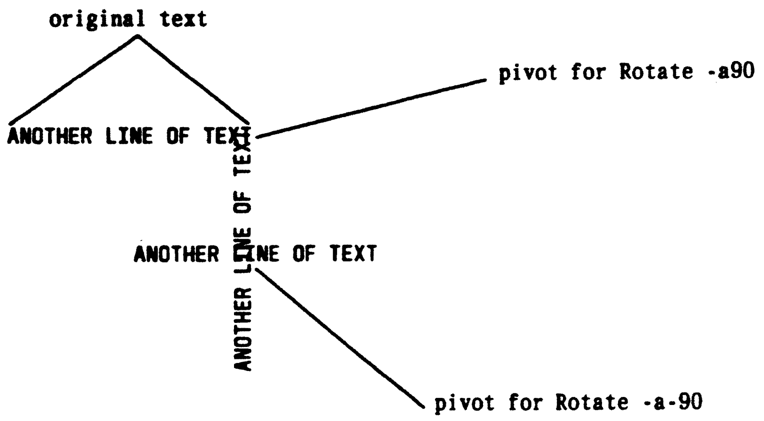


Figure 14-7. Rotating Text

Changing the Style and Width of Lines In the current editor, objects can be drawn from lines in any of five styles: solid (**so**), dashed (**da**), dot-dashed (**dd**), long-dashed (**ld**) and three widths narrow (**n**), medium (**m**), and bold (**b**). Style is controlled by the **s** option and width by **w**. The next example creates a narrow dotted line:

```
*Lines -wn,sdo<cr>
<position cursor><sp>
<position cursor><sp>
<cr>
```

Using the **E**dit command, the line can be changed to bold dot-dashed:

```
*Edit -wb,sdd<cr>
$.0<cr>        (reference the object-handle of the previous line)
<cr>           (complete the defined-area)
<cr>
```

View Commands

All of the objects drawn lie within a Cartesian plane, 65,534 units on each axis, known as the *universe*. Thus far, only a small portion of the universe has been displayed on the screen. The command

```
*view -u<cr>
```

displays the entire universe.

Windowing A mapping of a portion of the universe onto the display screen is called a *window*. The extent or magnification of a window is altered using the **zoom** command. To build a window that includes all of the objects drawn, type

```
*zoom<cr>
<position cursor><sp>   (above and to the left of all
                        the object)
<position cursor><cr>   (below and to the right, also
                        end points)
<cr> (verify)
```

Zooming can be either *in* or *out*. Zooming in, as with a camera lens, increases the magnification of the window. The area outlined by *points* is expanded to fill the screen. Zooming out decreases magnification. The current window is shrunk so that it fits within the defined-area. The direction of the zoom is controlled by the sense of the **out** flag; **o** true means zoom out.

The location of a window is altered using the **view** command. View moves the window so that a given point in the universe lies at a given location on the screen.

```
*view<cr>
<position cursor><cr>   (locate a point in the universe)
<position cursor><cr>   (locate a point on the screen)
```

View also provides access to several predefined windows. As seen earlier, view **−u** displays the entire universe. The view **−h** command displays the *home-window*. The home-window is the window that circumscribes all of the objects in the universe. The result is similar to that of the example using **zoom** given earlier.

Lastly, the **view** command permits selection of a window on a particular *region*. The universe is partitioned into 25 equal-sized regions. Regions are numbered from 1 to 25 beginning at the lower left and proceeding toward the upper right. Region 13, the center of the universe, is used as the default region by drawing commands such as **plot**(1) and **vtoc**(1).

Other Commands

Interacting with Files The **write** command saves the contents of the display buffer by copying it to a file:

```
*write filename<cr>
```

The contents of *filename* will be a GPS, thus it can be displayed using any of the device filters (e.g., **td** (1)) or read back into **ged**.

A GPS is read into the editor using the **read** command:

```
*read filename<cr>
```

The GPS from *filename* is appended to the display buffer and then displayed. Because **read** does not change the current window, only some (or none) of the objects read may be visible. A useful command sequence to view everything read is

```
*read -e- filename<cr>
*view -h<cr>
```

The display function of **read** is inhibited by setting the **echo** flag to false; **view —h** windows on and displays the full display buffer.

The **read** command may also be used to input text files. The form is

```
read [-option(s)] filename<cr>
```

followed by a single point to locate the first line of text. A text object is created for each line of text from *filename*. Options to the **read** command are the same as those for the **T**ext command.

Leaving the Editor The **quit** command is used to terminate an editing session. As with the text editor **ed**, **quit** responds with **?** if the internal buffer has been modified since the last **write** command. A second **quit** command forces exit.

Other Useful Information

One-Line ICON/UXV Escape As in **ed**, the ! provides a temporary escape to the shell.

Typing Ahead Most programs under the ICON/UXV operating system allow input to be typed before the program is ready to receive it. In general, this is not the case with **ged**; characters typed before the appropriate prompt are lost.

Speeding up Things Displaying the contents of the display buffer can be time consuming, particularly if much text is involved. The wise use of two flags to control what gets displayed can make life more pleasant:

- The **echo** flag controls echoing of new additions to the display buffer.
- The **text** flag controls whether text will be outlined or drawn.

Command Summary

In the summary, characters actually typed are printed in boldface. Command stages are printed in italics. Arguments surrounded by brackets (e.g., [...]) are optional. Parentheses surrounding arguments separated by "or" means that exactly one of the arguments must be given.

For example, the **D**elete command accepts the arguments –**u**niverse, –**v**iew, and *points.*

Construct Commands

Arc	[–**e**cho,**s**tyle,**w**idth] *points*
Box	[–**e**cho,**s**tyle,**w**idth] *points*
Circle	[–**e**cho,**s**tyle,**w**idth] *points*
Hardware	[–**e**cho] *text points*
Lines	[–**e**cho,**s**tyle,**w**idth] *points*
Text	[–**a**ngle,**e**cho,**h**eight,**m**idpoint,**r**ightpoint, **t**ext,**w**idth] *text points*

Edit Commands

Delete	(– (**u**niverse or **v**iew) or *points*)
Edit	[–**a**ngle,**e**cho,**h**eight,**s**tyle,**w**idth] (– (**u**niverse or **v**iew) or *points*)
Kopy	[–**e**cho,**p**oints,**x**] *points pivot destination*
Move	[–**e**cho,**p**oints,**x**] *points pivot destination*
Rotate	[–**a**ngle,**e**cho,**k**opy,**x**] *points pivot destination*
Scale	[–**e**cho,**f**actor,**k**opy,**x**] *points pivot destination*

View Commands

coordinates	*points*
erase	
new	
objects	(– (**u**niverse or **v**iew) or *points*)

points	(– (**labelled-points** or **universe** or **view**) or *points*)
view	(– (**home** or **universe** or **region**) or [–**x**] *pivot destination*)
x	[–**view**] *points*
zoom	[–**out**] *points*

Other Commands

quit or Quit	
read	[–**angle**,**echo**,**height**,**midpoint**,**rightpoint**,**text**, **width**] *filename* [*destination*]
set	[–**angle**,**echo**,**factor**,**height**,**kopy**,**midpoint**, **points**,**rightpoint**,**style**,**text**,**width**,**x**]
write	*filename*
!*command*	
?	

Options

Options specify parameters used to construct, edit, and view graphical objects. If a parameter used by a command is not specified as an *option*, the default value for the parameter will be used. The format of command *options* is

–*option* [,*option*]

where *option* is *keyletter*[*value*]. Flags take on the values of true or false indicated by + and –, respectively. If no value is given with a flag, true is assumed. Object options are

angle*n*	Specify an angle of *n* degrees.
echo	When true, changes to the display buffer will be echoed on the screen.
factor*n*	Specify a scale factor of *n* percent.
height*n*	Specify height of text to be *n* universe-units (*n* greater than or equal to 0 and less than 1280).
kopy	The commands Scale and Rotate can be used to either create new objects or to alter old ones. When the **kopy** flag is true, new objects are created.

midpoint	When true, use the midpoint of a text string to locate the string.
out	When true, reduce magnification during **zoom**.
points	When true, operate on points; otherwise operate on objects.
rightpoint	When true, use the rightmost point of a text string to locate the string.
style*type*	Specify line style to be one of the following *types*: **so** solid **da** dashed **dd** dot-dashed **do** dotted **ld** long-dashed
text	Most text is drawn as a sequence of lines. This can sometimes be painfully slow. When the **text** flag (**t**) is false, strings are outlined rather than drawn.
width*type*	Specify line width to be one of the following *types*: **n** narrow **m** medium **b** bold
x	One way to find the center of a rectangular area is to draw the diagonals of the rectangle. When the **x** flag is true, defined-areas are drawn with their diagonals.

Area options are:

home	Reference the home-window.
region*n*	Reference region *n*.
universe	Reference the universe-window.
view	Reference those objects currently in view.

Some Examples of What Can Be Done

The following examples are used to illustrate use of the **ged**.

Example 1--Text Centered Within a Circle

```
*Circle<cr>
<position cursor><sp>   (establish center)
<position cursor><cr>   (establish radius)
*Text -m<cr>            (text is to be centered)
some text<cr>
$.0<cr>            (first point from previous set,
                   i.e., circle center)
<cr>
```

Figure 14-8 shows the output of these commands.

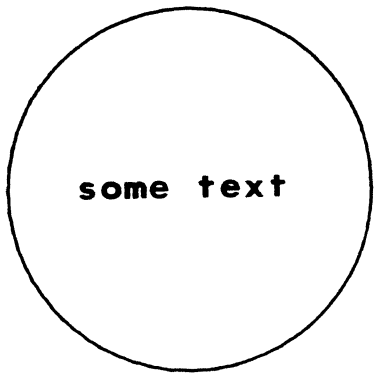


Figure 14-8. Text Centered Within a Circle

NOTES

Example 2--Making Notes on a Plot

```
*! gas | plot -g >A<cr>  (generate a plot, put it in file A)
*read -e- A<cr>  (input the plot, but do not display it)
*view -h<cr>         (window on the plot)
*Lines -sdo<cr>     (draw dotted lines)
<position cursor><sp>   (0,6.5 y-axis)
<position cursor><sp>   (6.5,5.5)
<position cursor><sp>   (5.5,0 x-axis)
<cr>  (end of Lines)
*set -h150,wn<cr>(set text height to 150, line width to
                   narrow)
*Text -r<cr>         (right justify text)
threshold beyond which nothing matters<cr>
<position cursor><cr>   (set right point of text)
*Text -a-90<cr>  (rotate text negative 90 degrees)
threshold beyond which nothing matters<cr>
<position cursor><cr>   (set top end of text)
*x<cr>       (find center of plot)
<position cursor><sp>   (top left corner of plot)
<position cursor><cr>   (bottom right corner of plot)
*Text -h300,wm,m<cr>  (build title: height 300, weight
                   medium, centered)
SOME KIND OF PLOT<cr>
<position cursor><cr>   (set title centered above plot)
*view -h<cr>         (window on the resultant drawing)
```

Figure 14-9 shows the output of these commands.

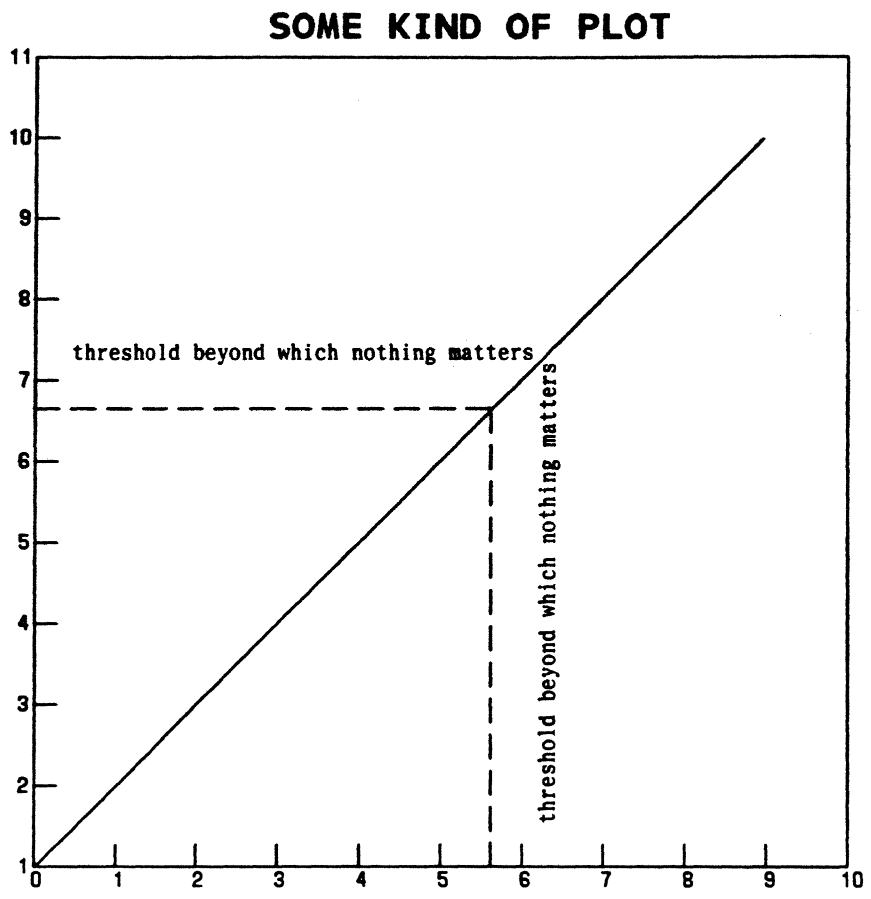


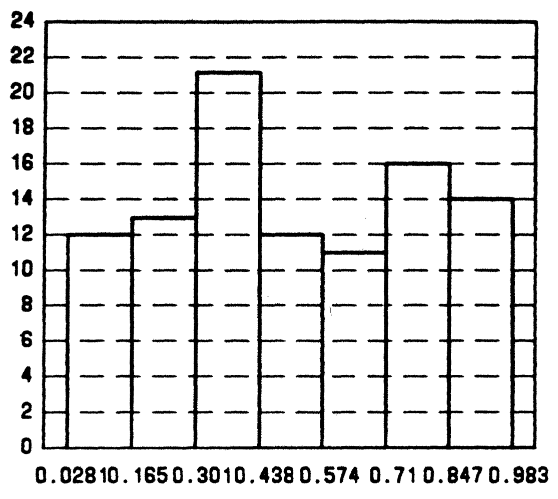
Figure 14-9. Making Notes on a Plot

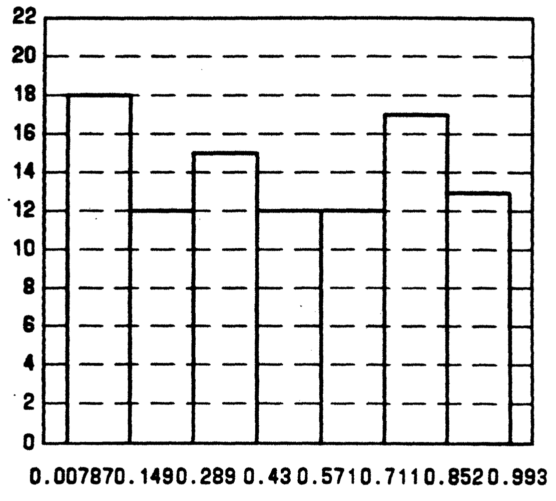
Example 3--A Page Layout with Drawings and Text

```
*! rand –s1,n100 | title –v"seed 1" | qsort | bucket |
        hist –r12 >A<cr> (put a histogram, region
                12, of 100 random numbers in file A)
*! rand –s2,n100 | title –v"seed 2" | qsort | bucket |
        hist –r13 >B<cr> (put another histogram,
                         region 13, into file B)
*! ed<cr>      (create a file of text using the text editor)
a<cr>
On this page are two histograms<cr>
from a series of 40<cr>
designed to illustrate the weakness<cr>
of multiplicative congruential random number
generators.<cr>
.pl 3<cr>      (mark end of page)
.<cr>
w C<cr>        (put the text into file C)
151
q<cr>
*! nroff C | yoo C<cr>      (format C, leave the output
                            in C)
*view –u<cr>        (window on the universe)
*read –e– A<cr
*read –e– B<cr>
*view –h<cr>        (view the two histograms)
*read –h300,wn,m C<cr> (text height 300, line weight
                        narrow, text centered)
<position cursor><cr>   (center text over two plots)
*view –h<cr>        (window on the resultant drawing)
```

Figure 14-10 shows the output of these commands.

ON THIS PAGE ARE TWO HISTORGRAMS FROM A SERIES OF
40 DESIGNED TO ILLUSTRATE THE WEAKNESS OF MULTIPLICATIVE
CONGRUENTIAL RANDOM NUMBER GENERATORS.





SEED 1 SEED 2

Figure 14-10. Page Layout with Drawings and Text

Chapter 15

ADMINISTRATIVE INFORMATION

Chapter 15

ADMINISTRATIVE INFORMATION

Chapter Introduction

This chapter is a reference guide for system administrators using or establishing a graphics facility on a ICON/UXV operating system. It contains information about directory structure, installation, makefiles, hardware requirements, and miscellaneous facilities of the graphics package.

Graphics Structure

Figure 15-1 contains a graphical representation of the directory structure of graphics. In this part, the shell variable *SRC* will represent the parent node for graphics source and is usually set **/usr/src/cmd**.

The **graphics** command (see **graphics**(1G)) resides in **/usr/bin**. All other graphics executables are located in **/usr/bin/graf**; the **/usr/lib/graf** directory contains text for **whatis** documentation (see **gutil**(1G)) and editor scripts for **ttoc** (see **toc**(1G)).

Graphics source resides below the directory **$SRC/graf**; **$SRC/graf** is broken into the following subdirectories:

- **include** — contains the following header files: **debug.h**, **errpr.h**, **gsl.h**, **gpl.h**, **setopt.h**, and **util.h**.
- **src** — contains source code partitioned into subdirectories by subsystem. Each subdirectory contains its own *Makefile* (or *Install* file for **whatis.d**).
  - **glib.d** — contains source used to build the graphical subroutine library, **$SRC/graf/lib/glib.a**.
  - **stat.d** — contains source for numerical manipulation and plotting routines.
  - **dev.d** — contains source code for device filters partitioned into subdirectories.
    - **lolib** and **uplib** — contains source used to create device independent libraries.
    - **hp7220.d** — contains source for **hpd** (a Hewlett-Packard (HP*) Plotter display function).

* HP is a registered trademark of Hewlett-Packard Company.

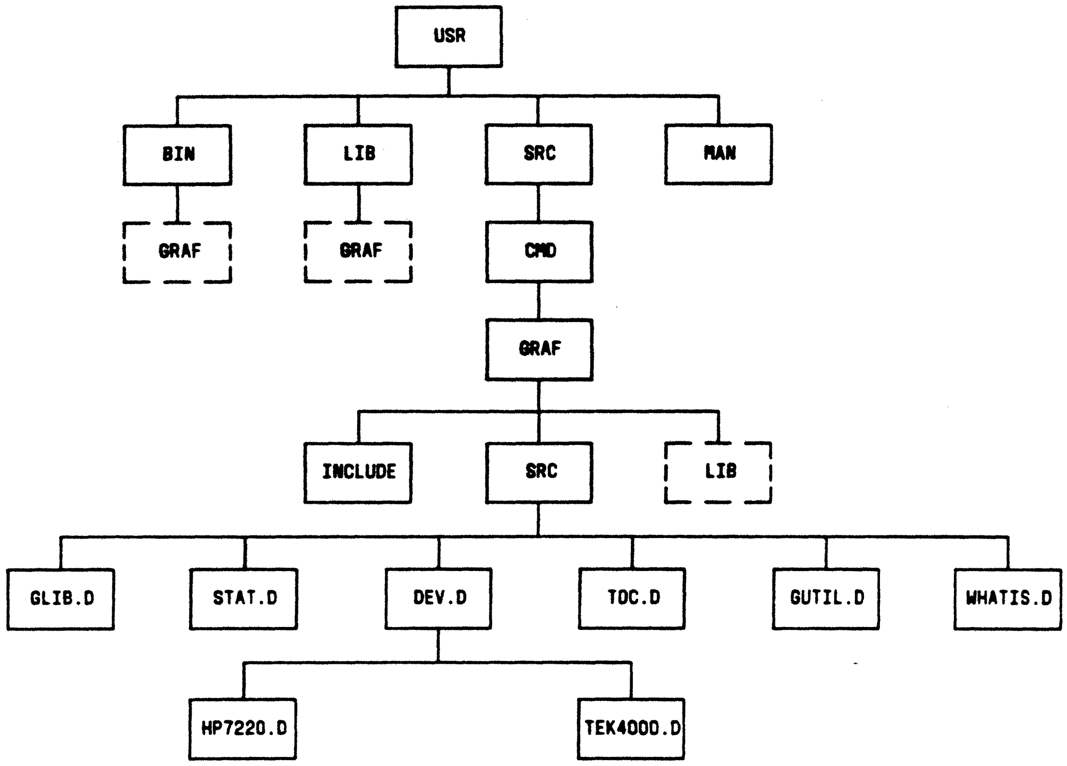


Figure 15-1. Directory Structure of Graphics Program

- **tek4000.d** — contains source for **ged** (the graphical editor), **td** (a TEKTRONIX display function), and other TEKTRONIX dependent routines.

- **gutil.d** — contains source for graphical utility programs.
- **toc.d** — contains source for table of contents drawing routines.
- **whatis.d** — contains **nroff** files and the installation routine for on-line documentation.

- **lib** — contains **glib.a** which contains commonly used graphical subroutines.

The *ICON/UXV Reference Manual* entries for **graphics** consist of the following: **gdev**(1G), **ged**(1G), **graphics**(1G), **gutil**(1G), **stat**(1G), **toc**(1G), and **gps**(5).

Installing Graphics

Procedures for installing **graphics**:

— To build the entire **graphics** package, execute (as super-user):

```
/usr/src/:mkcmd graf
```

— To build a particular **graphics** subsystem, use the shell variable *ARGS*:

```
ARGS=subsystem /usr/src/:mkcmd graf
```

A *subsystem* is either **glib**, **stat**, **dev**, **toc**, **gutil**, or **whatis**. **Glib** must exist before other subsystems can be built. Write permission in **/usr/bin** and **/usr/lib** is needed, and the following libraries are assumed to exist:

- /lib/libc.s — Standard C library, used by all subsystems.
- /lib/libm.a — Math library, used by all subsystems.
- /usr/lib/macros/mm[nt] — Memorandum macros for [**nt**]*roff*, used by the **whatis** subsystem.

The complete build process takes approximately two hours of system time. If the build must be stopped, it is a good idea to restart from the beginning. Upon completion, the following things will be created and owned by **bin**:

- /usr/lib/graf — A directory for data and editor scripts.
- /usr/bin/graf — A directory for executables.
- /usr/bin/graphics — Command entry point for graphics.

whatis.d contains source files for **whatis** and the executable command **Install**.

```
Install command name
```

calls **nroff** to produce **whatis** documentation for *command name* in **/usr/lib/graf**. To install the entire **whatis** subsystem, use: **mkcmd** as described above.

Makefile Parameters

Makefiles use executable shell procedures **cco** and **cca**. **Cco** is used to compile C source and install load modules in **/usr/bin/graf**. The **cca** command compiles C programs and loads object code into archive files.

Makefiles use various macro parameters, some of which can be specified on the command line to redirect outputs or inputs. Parameters specified in higher level *Makefiles* are passed to lower levels. Below is a list of specifiable parameters for *Makefiles* followed by their default values in parentheses and an explanation of their usage:

$SRC/graf/graf.mk:

- BIN (/usr/bin) - installation directory for the **graphics** command.
- BIN (/usr/bin/graf) - installation directory for other **graphic** commands.
- SRC (/usr/src/cmd) - parent directory for source code.

$SRC/graf/src/Makefile:

- BIN1 (/usr/bin) - installation directory for the **graphics** command.
- BIN2 (/usr/bin/graf) - installation directory for other **graphic** commands.
- LIB (/usr/lib/graf) - installation directory for **whatis** documentation.

$SRC/graf/src/stat.d/Makefile:

- BIN (../../bin) - installation directory for executable commands.

$SRC/graf/src/toc.d/Makefile:

- BIN (../../bin) - installation directory for executable commands.

$SRC/graf/src/dev.d/Makefile:

- BIN (../../bin) - installation directory for executable commands.

$SRC/graf/src/dev.d/hp7220.d/Makefile:

- BIN (../../../bin) - installation directory for executable commands.

$SRC/graf/src/dev.d/tek4000.d/Makefile:

- BIN (../../../bin) - installation directory for executable commands.

$SRC/graf/src/gutil.d/Makefile:

- BIN (../../bin) - installation directory for executable commands.

The following example will make a new version of the **ged**, installing it in **/a1/pmt/dp/bin** (assuming that necessary libraries were previously built):

```
cd $SRC/graf/src/dev.d/tek4000.d
make BIN=/a1/pmt/dp/bin ged
```

Hewlett-Packard Plotter

The **graphics** display function, **hpd**, uses graphics plotter that emulates the TEKTRONIX 4014 terminal. The plotter can be connected to the computer in series with a terminal via a dedicated or dial-up line. This arrangement allows the plotter to intercept plotting instructions while passing other data to the terminal unaltered, thus providing for normal terminal operation. Plotter switch settings should match those of the terminal. The plotter operating manual contains a more complete discussion.

TEKTRONIX Terminal

The graphics display function, **td**, and the **ged**, both use TEKTRONIX Series 4010 storage tubes. Below is a list of device considerations necessary for **graphics** operation.

Inittab Entry

When a TEKTRONIX 4010 series terminal is connected to a ICON/UXV operating system via a dedicated 4800 or 9600 baud line, **/etc/inittab** should reference speed table entry **6** (the table may vary locally) of **getty**. Speed table entry **6** is designed specifically for the TEKTRONIX 4014 and, among other things, sets a form-feed delay so that the screen may be cleared without losing information and clears the screen before prompting for a login. See **stty**(1), **inittab**(5) and **getty**(8) for more information.

Strap Options

The standard strap options as listed below should be used (see the Reference Manual for the TEKTRONIX 4014):

- LF effect — LF causes line-feed only.
- CR effect — CR causes carriage return only.
- DEL implies loy — DEL key is interpreted as low-order y value.

- Graphics input terminators — None.

Enhanced Graphics Module

The Enhanced Graphics Module (EGM) for TEKTRONIX terminals is required for **graphics**. The EGM provides different line styles (solid, dotted, dot-dashed, dashed, and long-dashed), right and left margin cursor location, and 12-bit cursor addressing (4096 by 4096 screen points).

Miscellaneous Information

Announcements

The **graphics** command provides a means of printing out announcements to users. To set up an announcement facility, create a readable text file containing the announcements named *announce* in directory **/usr/adm**. If it is desired to use a different directory for announcements, redefine the shell variable *GRAF* in **/usr/bin/graphics**.

Graflog

The **graphics** command also provides a means of monitoring its use by listing users in a file.

Each time a user executes **graphics**, an entry of the login name, terminal number, and system date are recorded in file *graflog* in directory **/usr/adm**, the same directory used for announcements.

Restricted Environments

Restricted environments can be used to limit access to the system [see **sh**(1)]. A restricted environment for **graphics** can be set up by creating the directories **/rbin** and **/usr/rbin** and populating them with restricted versions of regular ICON/UXV system commands, so that the environment cannot be compromised. In particular, **ed**(1), **mv**(1), **rm**(1), and **sh**(1) require restricted interface programs which do not allow users to move or remove files whose names begin with ".".

To create a restricted environment for **graphics**:

- Create a restricted **ged** command in **/usr/rbin** as follows:

  ```
  exec /usr/bin/graf/ged –R
  ```

- Create restricted logins for users or create a community login with a working directory (reached through **.profile**) set up for each user. A restricted login specifies **/bin/rsh** as the terminal interface program and is created by adding **/bin/rsh** to the end of the **/etc/passwd** file entry for that login.

- Call **graphics –r** from **.profile**.

The execution of **graphics –r** changes $*PATH* to look for commands in **/rbin** and **/usr/rbin** before **/bin** and **/usr/bin** and executes a restricted shell. The **–R** option is appended to the **ged** command so that the escape from **ged** to the ICON/UXV operating system (!*command*) will also use a restricted shell.

APPENDICES

Appendix A

SELECTED ICON/UXV SYSTEM DOCUMENTATION

The *ICON/UXV User Guide* is a general introduction to the ICON/UXV system. Several documents are available for follow-up study and for further reference. This appendix highlights documents to which you should refer next for detailed information on the topics presented in this guide. In addition, it provides brief instructions for obtaining these documents.

The documents selected for the appendix conform to the scope of the *ICON/UXV User Guide*; your needs may be different. For example, your documentation requirements will depend upon your use of the ICON/UXV system, the special add-on packages available to you, and the computer on which you run the ICON/UXV system. You may require more advanced or more detailed documentation on such things as support tools, system administration, or error messages. If so, refer to the *Documentation Directory* described in this appendix or contact your Icon International, Inc. Account Representative.

DOCUMENT DESCRIPTIONS

Table A-1 summarizes the additional documentation by select code number (the reference number you must use when ordering any of the documents) and title. Following are brief descriptions of these documents.

ICON/UXV User Reference Manual (172-036-004)
: Gives complete instructions on all standard ICON/UXV system commands, including proper format and all options.

ICON/UXV Programmer Guide (172-036-005)
: Describes the programming languages and language aids available on the ICON/UXV system, including C, FORTRAN-77, and libraries.

ICON/UXV Programmer Reference Manual (172-036-006)
: Gives detailed instructions for programmers using the ICON/UXV system. This manual covers system calls, library functions and subroutines, file formats, and miscellaneous facilities for the programmer.

ICON/UXV Editing Guide (172-036-007)
: Contains beginning and advanced information on the editing programs (including **ed** and **vi**) available with the ICON/UXV system.

TABLE A-1

ICON/UXV System Documentation Arranged by Select Code

Select Code	ICON/UXV System Document Title
172-036-003	User Guide
172-036-004	User Reference Manual
172-036-005	Programmer Guide
172-036-006	Programmer Reference Manual
172-036-007	Editing Guide

ORDERING DOCUMENTS

You may order ICON/UXV system documents through your Icon International, Inc. Account Representative. This appendix does not cover pricing information, which is subject to change. For current and complete document availability and pricing information, contact your Icon International Account Representative.

Appendix B

FILE SYSTEM ORGANIZATION

To make full use of the capabilities of the file system, you must understand its organization of files and directories. This appendix summarizes the standard system directories provided and maintained by the ICON/UXV operating system.

The file structure of the ICON/UXV system is a treelike structure (or *hierarchy*), with directories and files descending and branching out from a single directory. This directory is called the *root* and is designated by a slash (/). One path from the root leads to a directory that, in turn, leads to the directory that you find yourself in when you log in (your *home directory*). Under your home directory, you can establish your own hierarchy of directories and files for organizing information.

Other paths lead from the root to *system directories*. These directories are available to all users. The system directories described in this book are common to all UNIX system installations; they are provided and maintained by the operating system. In addition to this standard set of directories, you may have other system directories available to you. To obtain a complete listing of all the directories and files in the root directory on your ICON/UXV system, type:

ls –l /<CR>

When you understand the organization of directories and files in the ICON/UXV system, you will be able to use path names to move around in the structure and find out what other directories contain. For example, you can move to the directory **/bin** (which contains ICON/UXV system executable files) by typing:

cd /bin<CR>

and then list its contents by inputting one of the following command lines:

ls<CR>	for a list of file and directory names
ls –l<CR>	for a detailed list of the contents

Or, you can use the **ls** command to view the contents of the **/bin** directory from any directory. Type:

ls /bin<CR>	for a short listing
ls –l /bin<CR>	for a detailed listing

You may use the same commands to look at the contents of other system directories, substituting the desired directory name for **/bin**.

Figure B-1 shows the root and the major system directories belonging to it. On the following pages are brief descriptions of each system directory.

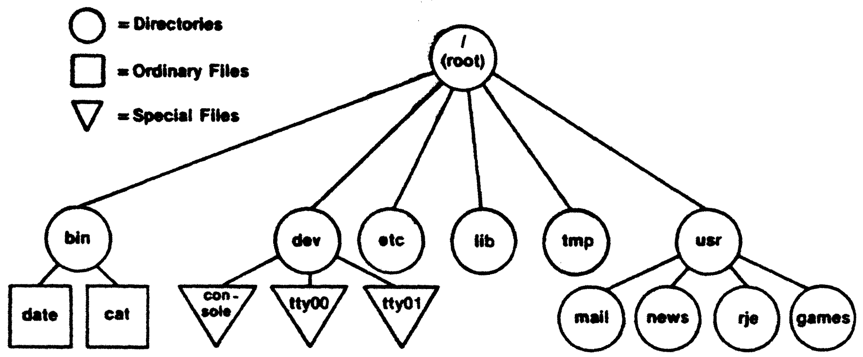


Figure B-1. Sample of file system structure of the ICON/UXV system

ICON/UXV SYSTEM DIRECTORIES

/	Root, the source of the file system.
/bin	Many executable programs and utilities reside in this directory, such as:

cat
date
login
grep
mkdir
who

/lib This directory contains available program and language libraries, such as:

libc.a	system calls, standard I/O)
libm.a	math routines and support for languages such as C, FORTRAN, and BASIC.

/dev This directory contains special files that represent peripheral devices, such as:

console	console
lp	line printer
tty00	user terminal
tty01	user terminal
rp00	disks.

/etc Special programs and data files for system administration reside in this directory.

/tmp This directory holds temporary files, such as the buffers created for editing a file.

/usr This directory is the parent to the following subdirectories:

news	important news items
rje	data sent by the remote job entry
mail	electronic mail
games	electronic games
man	on-line user's manual
spool	files waiting to print on the line printer.

Appendix C

SUMMARY OF ICON/UXV SYSTEM COMMANDS

This appendix consists of two sections.

- *Table C-1* summarizes the ICON/UXV system commands covered in this guide. The table lists these commands alphabetically and supplies a brief definition for each one.

- The remainder of this appendix contains abridged descriptions of the capabilities of these commands.

TABLE C-1

Summary of ICON/UXV System Commands

Name	Description
at	Specify time to run a job
banner	Display posters on the standard output
batch	Run jobs when system load permits
cat	Display contents of a file on the terminal
cd	Change your working directory
chmod	Change permission modes for a file or directory
cp	Copy an existing file to another file
cut	Cut out selected fields in a file
date	Display the current date and time
diff	Find difference(s) between two files
echo	Echo input to the standard output
ed	Edit (or create) a file using line editor
grep	Search a file for a pattern
kill	Terminate a background process
lex	Generate programs for simple lexical tasks
lp	Print a file on the line printer
lpstat	Display current line printer status

(Continued on next page)

TABLE C-1--*continued*

Name	Description
ls	List the contents of a directory
mail	Send or receive electronic mail
mailx	Interactive message handing system
make	Maintain large programs or documents
man	On-line manual
mkdir	Make (create) a new directory
mv	Move and rename a file
nohup	Continue background processes after logoff
pg	Display file contents a page at a time
pr	Display a partially formatted file on terminal
ps	Show status of background processes
pwd	Display the current working directory
rm	Remove (delete) a file
rmdir	Remove (delete) an empty directory
sh	Execute a shell file/program
sort	Sort or merge files
spell	Find spelling errors in a file
stty	Report or set I/O options for a terminal
uname	Print the name of the current ICON/UXV system
uucp	Send a copy of a file directly to another user's login
uuname	List names of known remote systems
uupick	Retrieve a file in the public directory
uuto	Send a copy of a file to another user
uustat	**uuto** status inquiry
vi	Edit (create) a file using full screen editor
wc	Count lines, words, and characters in a file
who	Show who is logged into the system
yacc	Impose a structure on program input

COMMAND DESCRIPTIONS

at

Displays the job numbers of all jobs you have in the **at** or **batch** modes or in the background mode. Followed by a time, submits commands to be run at that time. A sample format for this command is:

```
at 0845am Jun 09<CR>
command1<CR>
command2<CR>
<^d>
```

If you use the **at** command without the date, the command executes within 24 hours at the time specified.

banner
Displays a message (in words up to 10 characters long) in large letters on the standard output.

batch
Submits command(s) to be processed when the system load is at an acceptable level. A sample format of this command is:

```
batch<CR>
command1<CR>
command2<CR>
<^d>
```

You may use a shell script for a command in **batch**. This may be useful and timesaving if you have a set of commands you frequently submit using the **batch** command.

cat
Displays the contents of a specified file at your terminal. To pause the output, use <^s>; <^q> resumes the display. To stop the display and return to the shell prompt, press the <BREAK> key.

cd
Changes your position in the file system from the current directory to your home directory. Followed by a directory name, this command changes your position in the file system from the current directory to the directory specified. You can move up or down in the file system. By using a path name in place of the directory name, you may jump several levels with one command.

cp
Copies a specified file into a new file. The **cp** command leaves the original file intact; if you do not want to retain the file as is, use **mv**.

cut
Cuts out specified fields from each line of a file. This command can be used to cut columns from a table, for example.

date
Displays the current date and time.

diff
Compares two files. The **diff** command reports which lines are different and what changes should be made to the second file to make it the same as the first file.

echo
Displays (echoes) input to the terminal on the standard output, followed by a carriage return.

ed

Edits a specified file using the line editor. If there is no file by that name, the **ed** command creates a file. See *Chapter 5* for detailed instructions on using the **ed** editor.

grep

Searches a specified file or files for a specified pattern and tells you which lines match. If you name more than one file, **grep** also tells you which file contains the pattern.

kill

Terminates a background process specified by its process identification number (**PID**). The **PID** can be found by using the **ps** command.

lex

Generates programs to be used in simple lexical analysis of text, perhaps as a first step in creating a compiler. See the *ICON/UXV System User Reference Manual* for details.

lp

Prints a specified file on the line printer. This gives you a paper copy of the file's contents.

lpstat

Displays the status of any requests made to the line printer system. Options are available to request more detailed information.

ls

Lists the names of all files in the current directory except those whose names begin with a dot (**.**). Options are available to list more detailed information about the files in the directory.

mail

Displays any electronic mail you may have received at your terminal, one message at a time. Each message ends with **?**; type **r** for a list of options available to you at this point. There are options for saving, forwarding, or deleting mail.

When followed by a login name, **mail** sends a message to the user with the specified login name through electronic mail. You may type in as many lines of text as you wish; a dot (.) entered at the beginning of a new line ends the message and sends it to the recipient. Press the **<BREAK>** key to stop the mail session while composing the message or while reading one.

mailx

A more sophisticated, expanded version of electronic mail. See the *Unix System User Reference Manual* for details.

make

Provides a method for maintaining and supporting large programs or documents on the basis of smaller ones. See the *ICON/UXV System User Reference Manual* for details.

man

Displays the manual page for a specified command at your terminal.

mkdir

Makes (creates) a new directory. The new directory becomes a subdirectory of the directory you are in when you issue the command. To create subdirectories or files in the new directory, you should move into the new directory with the **cd** command.

mv

Moves or renames a specified file. Either file name may be a path name. To make a copy of a file use the **cp** command.

nohup

Allows a command placed in the background to continue executing after you log off. Error messages are placed in a file called *nohup.out.*

pg

Displays the contents of a specified file at your terminal, a screenful at a time. After each screenful, the system pauses and waits for your instructions before proceeding.

pr

Displays a partially formatted version of a specified file at your terminal. The **pr** command shows page breaks, but does not implement any macros supplied for text formatter packages.

ps

Shows the status and number of all processes currently running. The **ps** command does not show the status of jobs in the **at** or **batch** queues, but it shows them when they are executing.

pwd

Displays the name of the current working directory. The **pwd** command shows the working directory with its full path name, beginning from the root. For an explanation of the file system organization, see *Appendix B.*

rm

Removes (deletes) a file. You may use metacharacters with the **rm** command, but with caution; a removed file cannot usually be recovered.

rmdir

Removes (deletes) a directory. The directory must be empty before you delete it; you must delete all files and subdirectories in the specified directory first.

sort

Sorts a file by the ASCII sequence and displays the results at your terminal. The sequence is as follows:

numbers before *letters*
capitals before *lower case*
alphabetical order

There are other options for sorting a file. For a complete list of **sort** options, see the *ICON/UXV System User Reference Manual.*

spell

Collects words from a specified file and checks them against a spelling list. Words not on the list or not related to words on the list (with suffixes, prefixes, etc.) are displayed.

stty

By itself, reports the settings of certain input/output options for your terminal. It sets these options when followed with appropriate options and arguments (see the *ICON/UXV System User Reference Manual*).

uname

Displays the name of the ICON/UXV system on which your login resides.

uucp

Sends a specified file directly to a user's login. See the *ICON/UXV System User Reference Manual* for details.

uuname
Lists the names of remote ICON/UXV systems that can communicate with your ICON/UXV system.

uupick
Searches the public directory for files sent to you by the **uuto** command. If files are found, it displays the file name and the system it came from, then prompts you (?) to take action.

uustat
Displays the status of your request to send files to another user with the **uuto** command.

uuto
Sends a specified file to another user. The destination is in the format **system!login** where the **system** must be on the list of systems generated by the **uuname** command.

vi
Edits a specified file using the screen editor. If there is no file by that name, **vi** creates the file. See *Chapter 6* for detailed information on using the **vi** editor.

wc
Counts the number of lines, words, and characters in a specified file and displays the results at your terminal.

who
Displays the login names of the users logged in to the ICON/UXV system on your computer, along with the terminals they are using and the times they logged in.

yacc
Imposes a structure on the input of a program. See the *ICON/UXV System User Reference Manual* for details.

Appendix D

QUICK REFERENCE TO ed COMMANDS

This *Quick Reference to* **ed** *Commands* is organized into two sections.

- The first section lists the commands, with brief descriptions, in alphabetical order.
- The second section groups the commands according to each topic discussed in *Chapter 5.*

The commands are shown as you type them. The general format for **ed** commands is:

[address1,address2]command[parameter]<CR>

where *address1* and *address2* denote line addresses and *parameter(s)* indicates data on which the command operates. You can find complete information on using **ed** commands in *Chapter 5, Line Editor Tutorial.*

COMMANDS IN ALPHABETICAL ORDER

.	Returns to command mode from text input mode.
.	Address of the current line.
.	Matches any single character (in a search pattern and in a substitution).
!command	Temporarily escapes to the shell to execute a shell command.
/	Acts as a delimiter (for **s**, **v**, or **g** commands).
\	Removes the meaning of a special character (in a search pattern and in a substitution).
=	Displays the address of the last line in the buffer.
.=	Displays the current line number.
+*x*	Relative address; add *x* to the current line number.
−*x*	Relative address; subtract *x* from the current line number.
*	Matches zero or more occurrences of the preceding character (in search or substitution patterns).
.*	Matches zero or more occurrences of any characters following the period (in search or substitution patterns).

[...]	Matches the first character of those characters within the brackets.
[^...]	If the caret (^) is the first character in brackets, finds and matches the first character that is *not* within the brackets.
^	The caret (^) matches the beginning of a line (in a search pattern and in a substitution).
/pattern	Searches forward in the buffer and addresses the first line after the current line that contains the **pattern** of text.
?pattern	Searches backward in the buffer and addresses the first line before the current line that contains the **pattern** of text.
$	Denotes the last line in the buffer.
$	Matches the end of a line.
&	Repeats the last pattern to be substituted.
%	Repeats the last replacement pattern.
@	Deletes the current line (text input mode) or a command line (command mode).
#	Deletes the character just entered (text input mode).
a	Creates text after the specified line.
c	Replaces text in the specified lines with new text.
CR	Carriage return; moves down a line in the buffer.
d	Deletes specified lines of text.
ed filename	Enters **ed** line editor to edit a file called *filename*; copies the file into the buffer.
E filename	Replaces the current buffer with the contents of a file called *filename*; deletes present contents of the buffer whether written to a permanent file or not.
f	Displays the name of the file being edited.
f newfile	Changes the current file name associated with the buffer to *newfile*.
g/pattern	Addresses all lines in the buffer that contain the specified **pattern** of text.
G/pattern	Addresses all lines in the buffer that contain the specified **pattern** of text; prints each occurrence for you to deal with separately.
h	Displays a short explanation of the previous diagnostic response (?).

H	Automatically displays explanations of diagnostic responses (?) throughout the editing session.
i	Inserts new text before the specified line.
j	Joins contiguous lines.
l	Displays the specified lines with all nonprinting (hidden) characters.
m	Moves the specified lines after a destination line; deletes the lines at the old location.
n	Displays the specified lines preceded by the line addresses and a tab space.
p	Displays the specified lines in the buffer.
P	Causes **ed** to print an asterisk (*) as a command mode prompt (for more details, see the *ICON/UXV Editing Guide* described in *Appendix A*).
q	Ends an editing session. If changes to the buffer were not written to a file, a warning (?) is given. Typing **q** a second time ends the session without writing to a file.
Q	Ends an editing session whether or not changes to the buffer were written into a file.
r filename	Appends the contents of a file called *filename* to the end of the present buffer contents.
s/old text/new text/	Substitutes the first occurrence of **old text** with **new text** on the current line. A **g** after the final slash changes all occurrences on the current line.
address1,address2s/old text/new text/	Substitutes the first occurrence of **old text** with **new text** on the lines denoted by *address1,address2*.
t	Copies the specified lines and places them after a destination line.
u	Undoes the last command given, except for **w** and **q** (command mode).
v/pattern	Addresses all lines in the buffer that *do not* contain the specified **pattern** of text.
V/pattern	Addresses all lines in the buffer that *do not* contain the specified **pattern** of text; prints each occurrence for you to deal with separately.
w	Copies the buffer contents into the file currently associated with the buffer.
w filename	Copies the buffer contents into a file called *filename*.

COMMANDS BY TOPIC

Commands for Getting Started

ed filename	Enters **ed** line editor to edit a file called *filename*.
a	Appends text after the current line.
.	Ends the text input mode and returns to the command mode.
p	Displays the current line.
d	Deletes the current line.
CR	Moves down one line in the buffer.
–	Moves up one line in the buffer.
w	Writes the buffer contents to the file currently associated with the buffer.
q	Ends an editing session. If changes to the buffer were not written to a file, a warning (**?**) is issued. Typing **q** a second time ends the session without writing to a file.

Line Addressing Commands

1, 2, 3...	Denotes line addresses in the buffer.
.	Address of the current line in the buffer.
.=	Displays the current line address.
$	Denotes the last line in the buffer.
,	Addresses lines 1 through the last line.
;	Addresses the current line through the last line.
+*x*	Relative address; add *x* to the current line number.
–*x*	Relative address; subtract *x* from the current line number.
/abc	Searches forward in the buffer and addresses the first line after the current line that contains the pattern *abc*.
?abc	Searches backward in the buffer and addresses the first line before the current line that contains the pattern *abc*.

g/abc	Addresses all lines in the buffer that contain the pattern *abc*.
v/abc	Addresses all lines in the buffer that *do not* contain the pattern *abc*.

Display Commands

p	Displays the specified lines in the buffer.
n	Displays the specified lines preceded by the line addresses and a tab space.

Text Input

a	Enters text after the specified line in the buffer.
i	Enters text before the specified line in the buffer.
c	Replaces text in the specified lines with new text.
.	On a line by itself, ends the text input mode and returns to the command mode.

Deleting Text

d	Deletes one or more lines of text (command mode).
u	Undoes the last command given (command mode).
@	Deletes the current line (text input mode) or a command line (command mode).
# or **backspace**	Deletes the last character typed in (text input mode).

Substituting Text

address1,address2s/old text/new text/command
Substitutes **new text** for **old text** within the range of lines denoted by ***address1,address2*** (which may be numbers, symbols, or text). The **command** may be **g**, **l**, **n**, **p**, or **gp**.

Special Characters

.	Matches any single character in search or substitution patterns.
*****	Matches zero or more occurrences of the preceding character in search or substitution patterns.
.*	Matches zero or more occurrences of any characters following the period in search or substitution patterns.
^	The caret (^) matches the beginning of the line in search or substitution patterns.
$	Matches the end of the line in search or substitution patterns.
\	Takes away the special meaning of the special character that follows in search and substitution patterns.
&	Repeats the last pattern to be substituted.
%	Repeats the last replacement pattern.
[...]	Matches the first occurrence of a pattern in the brackets.
[^...]	Matches the first occurrence of a character that is *not* in the brackets.

Text Movement Commands

m	Moves the specified lines of text after a destination line; deletes the lines at the old location.
t	Copies the specified lines of text and places the copied lines after a destination line.
j	Joins the current line with the next contiguous line.
w	Copies (writes) the buffer contents into a file.
r	Reads in text from another file and appends it to the buffer.

Other Useful Commands and Information

h	Displays a short explanation for the preceding diagnostic response (?).
H	Turns on the help mode, which automatically displays an explanation for each diagnostic response (?) during the editing session.
l	Displays nonprinting (hidden) characters in the text.
f	Displays the current file name.
f newfile	Changes the current file name associated with the buffer to *newfile*.
!command	Temporarily escapes to the shell to execute a shell command.
ed.hup	If the terminal is hung up before a **write** command, the editing buffer is saved in the file *ed.hup*.

Appendix E

QUICK REFERENCE TO vi COMMANDS

This *Quick Reference to* **vi** *Commands* is organized into two sections.

- The first section lists the commands, with brief descriptions, in alphabetical order.
- The second section lists the commands according to each topic discussed in *Chapter 6.*

Please note the following conventions when using this appendix.

- Typing the control key is denoted by a caret (^) and the key--for example, **^g**.
- "Current line," "current word," and "current character" refer to the line, word, or character denoted by the cursor.

The commands are shown as you type them. The general format for **vi** commands is:

[x]command[argument]

where **x** denotes a number and *argument* indicates data on which the command operates. You can find complete information on using **vi** commands in *Chapter 6, Screen Editor Tutorial.*

COMMANDS IN ALPHABETICAL ORDER

;	Continues the search for the character specified by the **f** command.
.	Repeats the action initiated by the last command.
\	Prints the characters that are normally nonprinting (hidden characters) text input mode.
:	Begins a line editor command.
:$	Moves the cursor to the beginning of the last line in the buffer.
:*n*	Moves the cursor to the beginning of the *n*th line of the buffer (*n* = line number).
–	Moves the cursor up one line.
+	Moves the cursor down one line.
(	Moves the cursor to the beginning of the current sentence.

)	Moves the cursor to the beginning of the next sentence.
{	Moves the cursor to the beginning of the current paragraph.
}	Moves the cursor to the beginning of the next paragraph.
~	Change uppercase to lowercase or lowercase to uppercase.
/pattern	Searches forward in the buffer for **pattern**.
?pattern	Searches backward in the buffer for **pattern**.
@	In text input mode, erases the current line.
CR	Carriage return; in command mode, moves the cursor down one line.
space bar	Moves the cursor to the right one character.
a	Enters text after the cursor.
A	Enters text at the end of the current line.
b	Moves the cursor to the left one word, to the first character of that word.
B	Moves the cursor to the left one word (delimited only by blanks), to the first character of that word.
^b	Scrolls the screen back a full window, revealing the window of text above the current window.
BS	Backspace; in command mode, moves the cursor one character to the left.
BS	Backspace; in text input mode, deletes the current character.
cw	Replaces a word (or part of a word), from the cursor to the next space or punctuation, with new text.
cc	Replaces all the characters in the current line.
ncx	Replaces *n* number of text objects *x*, where *x* can include a sentence or a paragraph.
C	Replaces the characters in the current line from the cursor to the end of the line.
D	Deletes the line from the cursor to the end of the line.
^d	Scrolls the screen down a half window, revealing lines below the current window.
^d	Escapes the temporary return to the shell and returns to **vi** to edit the current window.

dd	Deletes the current line.
dw	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.
n**d*****x***	Deletes *n* number of text objects *x*, where *x* can include a sentence or a paragraph.
:.,$d	Deletes all the lines from the current line to the last line.
e	Moves the cursor to the end of the current word.
E	Moves the cursor to the end of the current word (delimited by blanks only); places the cursor on the last character before the next blank space or at the end of the current line.
ESC	Escape; returns to the command mode from any of the text input modes.
f***x***	Moves the cursor right to the specified character *x*.
F***x***	Moves the cursor left to the specified character *x*.
^f	Scrolls the screen forward a full window, revealing the window of text below the current window.
G	Moves the cursor to the beginning of the last line in the buffer.
n**G**	Moves the cursor to the *n*th line of the file (*n* = line number).
^g	Gives the line number, its position in the buffer (as a percentage of the buffer completed), and status.
:g/text/s//new words/g	Changes every occurrence of **text** to **new words**.
h	Moves the cursor one character to the left.
H	Moves the cursor to the first line on the screen, or "home".
i	Enters text to the right of the cursor.
I	Enters text to the left of the first character that is not a blank on the current line.
j	Moves the cursor down one line from its present position.
J	Joins the line immediately below the current line with the current line.
k	Moves the cursor up one line from its present position.
l	Moves the cursor one character to the right.

L	Moves the cursor to the last line on the screen.
^l	Clears and redraws the current window.
M	Moves the cursor to the middle line on the screen.
n	Repeats the last search command.
N	Repeats the last search command in the opposite direction.
o	Enters text at the beginning of a new line immediately below the current line.
O	Enters text at the beginning of a new line immediately above the current line.
p	Places the contents of the temporary buffer containing the last "delete" or "yank" command into the text after the cursor or below the current line.
"*x*p	Places the contents of register *x* after the cursor or below the current line.
:q	Quits **vi** if changes made to the buffer were written to a file.
:q!	Quits **vi** whether or not changes made to the buffer were written to a file.
r	Replaces the current character.
R	Replaces only those characters that are typed over with new text; continues to append new text after the end of the line is reached.
:r filename	Inserts the contents of a file called *filename* under the current line of the buffer.
s	Deletes the current character and appends text.
S	Replaces all the characters in the current line.
:s/text/new words/	Replaces **text** with **new words** on the current line.
:s/text/new words/g	Changes every occurrence of **text** on the current line to **new words**.
t*x*	Moves the cursor right to the character just before the specified character *x*.
T*x*	Moves the cursor left to the character just after the specified character *x*.
u	Undoes the last command.
U	Erases the last change on the current line.
^u	Scrolls the screen up a half window, revealing the lines of text above the current window.

^v	In text input mode, prints characters that are normally nonprinting (hidden characters).
vi filename	Enters **vi** screen editor to edit the file *filename*.
vi file1 file2 file3	Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.
vi −r file1	Restores the changes made to file *file1* that were lost because of an interrupt in the system.
view file1	Views file *file1* in the read-only mode of **vi**.
w	Moves the cursor forward to the first character in the next word.
W	Ignores all punctuation and moves the cursor forward to the beginning of the next word delimited only by blanks.
:w filename **:n**	When editing more than one file, writes the buffer to the file called *filename* and calls the next file in the buffer (use **:n** only after **:w**).
:w filename **:q**	Writes the buffer to a new file called *filename* and quits **vi**.
:wq	Writes the buffer to a file and quits **vi**.
:*x*,*z*w data	Writes lines from the number *x* through the number *z* into a new file called **data**.
:w! filename **:q**	Overwrites an existing file called *filename* and quits **vi**.
^w	In text input mode, deletes the current word delimited by blanks.
x	Deletes the current character.
nyx	Places (yanks) a copy of *n* numbertext objects *x* into a temporary buffer, where *x* can include a word, line, sentence, or paragraph.
"*lyn*	Places a copy of text object *x* into a register named by a letter *l*.
yy	Places the current line of text into a temporary buffer.
ZZ	Writes the buffer to a file and quits **vi**

SUMMARY OF vi COMMANDS BY TOPIC

Commands for Getting Started

TERM=code	
export TERM	Before entering **vi**, sets the terminal configuration.
vi filename	Enters **vi** screen editor to edit a file called *filename*.
a	Enters text after the cursor.
h	Moves the cursor to the left one character.
j	Moves the cursor down one line.
k	Moves the cursor up one line.
l	Moves the cursor to the right one character.
x	Deletes the current character.
CR	Carriage return; moves the cursor down to the beginning of the next line.
ESC	Escape; leaves text input mode and returns to command mode.
ZZ	Writes to a file and quits **vi**.
:q	Quits **vi** if changes made to the buffer were written to a file.

Commands for Positioning in the Window

Character Positioning

h	Moves the cursor one character to the left.
l	Moves the cursor one character to the right.
BS	Backspace; moves the cursor one character to the left.
space bar	Moves the cursor one character to the right.
f*x*	Moves the cursor right to the specified character *x*.
F*x*	Moves the cursor left to the specified character *x*.
;	Continues the search for the character specified by the **f** command. It will remember the character and seek out the next occurrence of the character on the current line.

t*x*	Moves the cursor right to the character just before the specified character *x*.
T*x*	Moves the cursor left to the character just after the specified character *x*.

Positioning by Lines

j	Moves the cursor down one line from its present position.
k	Moves the cursor up one line from its present position.
+	Moves the cursor down one line.
–	Moves the cursor up one line.
CR	Carriage return; moves the cursor down to the beginning of the next line.

Word Positioning

w	Moves the cursor to the right, to the first character in the next word.
W	Ignores all punctuation and moves the cursor to the right, to the beginning of the next word delimited only by blanks.
b	Moves the cursor to the left one word, to the first character of that word.
B	Moves the cursor to the left one word, (delimited only by blanks) to the first character of that word.
e	Moves the cursor to the end of the current word.
E	Moves the cursor to the end of the current word (delimited by blanks only); places the cursor on the last character before the next blank space or at the end of the current line.

Positioning by Sentences

(	Moves the cursor to the beginning of the current sentence.
)	Moves the cursor to the beginning of the next sentence.

Positioning by Paragraphs

{	Moves the cursor to the beginning of the current paragraph.
}	Moves the cursor to the beginning of the next paragraph.

Positioning in the Window

H	Moves the cursor to the first line on the screen, or "home".
M	Moves the cursor to the middle line on the screen.
L	Moves the cursor to the last line on the screen.

Commands for Positioning in the File

Scrolling

^f	Scrolls the screen forward a full window, revealing the window of text below the current window.
^d	Scrolls the screen down a half window, revealing lines of text below the current window.
^b	Scrolls the screen back a full window, revealing the window of text above the current window.
^u	Scrolls the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

G	Moves the cursor to the beginning of the last line in the buffer.
^g	Gives the line number, position in the buffer (as a percentage of the buffer completed), and status.
***n*G**	Moves the cursor to the *n*th line of the file (*n* = line number).

Searching for a Pattern

/pattern	Searchs forward in the buffer for the next occurrence of the **pattern** of text. Positions the cursor under the first character of the pattern.
?pattern	Searchs backward in the buffer for the first occurrence of **pattern** of text. Positions the cursor under the first character of the pattern.
n	Repeats the last search command.
N	Repeats the search command in the opposite direction.

Create Commands

a	Enters text after the cursor.
A	Enters text at the end of the current line.
i	Enters text to the right of the cursor.
I	Enters text to the right the first character that is not a blank on the current line.
o	Enters text at the beginning of a new line immediately below the current line.
O	Enters text at the beginning of a new line immediately above the current line.
ESC	Escape; returns to the command mode from any of the above text input modes.

Delete Commands

For the TEXT INPUT Mode

BS	Backspace; deletes the current character.
^w	Deletes the current word delimited by blanks.
@	Erases the current line of text.

For the COMMAND Mode

u	Undoes the last command.
U	Erases the last change on the current line.
x	Deletes the current character.
dw	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.
dd	Deletes the current line.
ndx	Deletes *n* number of text objects *x*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
D	Deletes the current line from the cursor to the end of the line.

Change Commands

r	Replaces the current character.
R	Replaces only those characters typed over with new characters; continues to append new text after the end of the line until the **ESC** command is given.
s	Deletes the current character and appends text until the **ESC** command is given.
S	Replaces all the characters in the current line.
cw	Replaces the current word or the remaining characters in the current word with new text, from the cursor to the next space or punctuation.
cc	Replaces all the characters in the current line.
ncx	Replaces *n* number of text objects *x*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
C	Replaces the remaining characters in the current line, from the cursor to the end of the line.

Cut and Paste Commands

p	Places the contents of the temporary buffer containing the last "delete" or "yank" command into the text after the cursor or below the current line.
yy	Places (yanks) a line of text into a temporary buffer.
*n***yx**	Places a copy of *n* number of text objects *x* into a temporary bufffer.
"*lyx*	Places a copy of text object *x* into a register named by a letter *l*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
"*x***p**	Places the contents of register *x* after the cursor or below the current line.

Special Commands

.	Repeats the action initiated by the last command.
J	Joins the line immediately below the current line with the current line.
\	Prints characters that are normally nonprinting (hidden characters) in text input mode.
^v	Prints characters that are normally nonprinting (hidden characters) in text input mode.

^l	Clears and redraws the current window.
~	Change uppercase to lowercase or lowercase to uppercase.

Line Editor Commands

:	Tells **vi** that the next commands are line editor commands.
:sh	Temporarily returns to the shell to perform some shell commands without leaving **vi**.
^d	Escapes the temporary return to the shell and returns to **vi** to edit the current window.
:*n*	Goes to the *n*th line of the buffer.
:*x*,*z*w data	Writes lines from the number *x* through the number *z* into a new file called **data**.
:$	Moves the cursor to the beginning of the last line in the buffer.
:.,$d	Deletes all the lines from the current line to the last line.
:r filename	Inserts the contents of the file *filename* under the current line of the buffer.
:s/text/new words/	Replaces the first instance of **text** on the current line with **new words**.
:s/text/new words/g	Replace every occurrence of **text** on the current line with **new words**.
:g/text/s//new words/g	Changes every occurrence of **text** in the buffer to **new words**.

Quit Commands

ZZ	Writes the buffer to a file and quits **vi**.
:wq	Writes the buffer to a file and quits **vi**.
:w filename **:q**	Writes the buffer to a new file named *filename* and quits **vi**.
:w! filename **:q**	Overwrites an existing file called *filename* with the contents of the buffer and quits vi.
:q!	Quits **vi** whether or not changes made to the buffer were written to a file.

:q	Quits **vi** if changes made to the buffer were written to a file.

Special Options for vi

vi file1 file2 file3
Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.

:w
:n When editing more than one file, writes the buffer to a file called *filename* and calls the next file in the buffer (use **:n** only after **:w**).

vi –r file1
Restores the changes made to *file1* that were lost because of an interrupt in the system.

view file1
Views *file1* in the read-only mode of **vi**. Changes cannot be made to the buffer.

Appendix F

SUMMARY OF SHELL PROGRAMMING INGREDIENTS

This summary of shell programming ingredients discussed in *Chapter 7, Shell Tutorial*, is organized into two sections.

- The first section is a summary of the variables and special symbols of the shell. These are arranged by topic in the order that they were discussed in the chapter.

- The second section shows the shell programming constructs.

SHELL COMMAND LANGUAGE

Special Characters in the Shell

* ? []"	Metacharacters; used as file name shortcuts (file name generation).
&	Executes commands in the background mode.
;	Sequentially executes several commands typed in on one line, each separated by ;.
\	Turns off the meaning of special characters in the shell.
'...' "..."	Single quotes turn off the special meaning of all characters. Double quotes allow $, , and " to retain their special meaning.

Redirecting Input and Output

<	Redirects the contents of a file into a command.
>	Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.
>>	Redirects the output to be added to the end of a file.
|	Directs the output of one command to be the input of the next command.
command	Substitutes the output of the enclosed command line.

Executing and Terminating Processes

batch	Submits the commands that follow to be processed at a time when the system load is at an acceptable level. ^**d** ends the **batch** command.
at	Submits the following commands to be executed at a specified time. ^**d** ends the **at** command.
at −l	Gives the current jobs in the **at** or **batch** queue.
at −r	Removes the **at** or **batch** job from the queue.

ps	Gives the status of the shell processes.
kill PID	Terminates the shell process with the specified process ID (PID).
nohup command list &	Completes background processes after logging off.

Executing A File

sh filename	Executes a shell file that is a program.
chmod u+x filename	Changes the mode of a file to be executable by you.
bin	Your directory for storing executable shell programs that are accessible to all of your other directories.

Variables

positional parameter
A variable defined by its position on the command line.

$#	Gives the number of positional parameters.
$*	Substitutes all positional parameters starting with the first positional parameter.

named variable
A variable that is given a name by you.

Variables Used by the Shell

HOME	Denotes your home directory; the default variable for the **cd** command.
PATH	Defines the path your login shell follows to find commands.
CDPATH	Defines the search path for the **cd** command.
MAIL	Gives the name of the file containing your electronic mail.
PS1 PS2	Defines the primary and secondary prompt strings.
TERM	Defines the type of terminal.
IFS	Defines the internal field separators; normally the space, the tab, and the carriage return.

SHELL PROGRAMMING CONSTRUCTS

Here Document

```
command <<!
input lines
!
```

For Loop

```
for variable<CR>
    in this list of values<CR>
do the following commands<CR>
    command 1<CR>
    command 2<CR>
    .<CR>
    .<CR>
    last command<CR>
done<CR>
```

While Loop

```
while command list<CR>
do<CR>
command1<CR>
command2<CR>
.<CR>
.<CR>
last command<CR>
done<CR>
```

If...Then

```
if this command is successful<CR>
then command1<CR>
command2<CR>
.<CR>
.<CR>
last command<CR>
fi<CR>
```

If...Then...Else

```
if command list<CR>
then command list<CR>
else command list<CR>
fi<CR>
```

Case Construction

```
case characters<CR>
in<CR>
    pattern1)<CR>
    command line 1<CR>
    .<CR>
    .<CR>
    last command line<CR>
    ;;<CR>
    pattern2)<CR>
    command line 1<CR>
    .<CR>
    .<CR>
    last command line<CR>
    ;;<CR>
    pattern3)<CR>
    command line 1<CR>
    .<CR>
    .<CR>
    last command line<CR>
    ;;<CR>
esac<CR>
```

break Statement

This statement forces the program to leave any loop and execute the next command.

Appendix G

AN INTRODUCTION TO THE C SHELL

William Joy
(revised by Mark Seiden)

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX® systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Additional information includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

INTRODUCTION

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX User Reference Manual. The *csh* documentation in section 1 of the manual provides a full description of all features of the shell and is the definitive reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

ACKNOWLEDGEMENTS

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

TERMINAL USAGE OF THE SHELL

The Basic Notion of Commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *built-in* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or*words* interpreted as a *command name* followed by *arguments*. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the*mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
     Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a ^D which sent an end-of-file to the mail program. (Here and throughout this document, the notation "^x" is to be read "control-x" and represents the striking of the x key while the control key is held down.) The mail program then echoed the characters 'EOT' and transmitted our message. The characters '% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '% ' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ^D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '% ' prompt.

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 and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal - the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is the delete key (equivalent to '^?') and the kill character is '^U'. Some people prefer to make the erase character the backspace key (equivalent to '^H'). You can make this be true by typing

```
tset -e
```

which tells the program *tset* to set the erase character to tset's default setting for this character (a backspace).

Flag Arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names, some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '-' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current *working directory* . The option *-s* is the size option, and

```
ls -s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

Output to Files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *meta-character* '>' and the name of the file where output is to be placed. Thus the command

```
      date > now
```

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in the section covering ***Shell Variables.***.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.[1] The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

```
      date > #now
```

Metacharacters in the Shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail,* or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% ' (although we can type our input even before it prompts).

® UNIX is a registered trademark of AT&T.

1 Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-\s-2CRT\s0) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a \s-2CRT\s0, your erase character should be a ^H, as we demonstrated in the section covering **The Basic Notion of Commands** how this could be set up.

Input from Files; Pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ^D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *-n* option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the *ls* command run with the option *-s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *-r* reverse sort option and the *head* command in combination with the previous command doing

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

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

```
    /etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory* , which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

```
    prog.c prog.o prog.errs prog.output
```

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' 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.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

```
      prog.*
```

This expression is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

```
      echo prog.*
```

will echo the names

```
      prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

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

will echo 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 will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
      prog.[co]
```

will match

```
      prog.c prog.o
```

in the example above. We can also place two characters around a '-' in this notation to denote a range. Thus

```
chap.[1-5]
```

might match files

```
      chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list*) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another user's login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a convenient way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp*. If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in the section covering ***Braces { ... } in Argument Expansion***, as it is used less frequently.

Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current *working directory*, or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '-' in an argument word to a command is to enclose it with single quotation characters ''', i.e.

```
echo '*'
```

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within ''' characters. It and the character ''' itself can be preceded by a single '\' to prevent their special meaning. Thus

```
echo \'\!
```

prints

```
'!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

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

which prints

```
'*
```

since the first '\' escaped the first ''' and the '*' was enclosed between ''' characters.

Terminating Commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing ^C on your terminal.[2] Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ^D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ^D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in the section covering ***Shell Variables***.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a ^D. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a ^Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

2 On some older UNIX systems the \s-2DEL\s0 or \s-2RUBOUT\s0 key has the same effect. "stty all will tell you the INTR key value.

```
% mail harold
Someone just copied a big file into my directory and its
name is
^Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1] + Stopped       mail harold
% fg
mail harold
funnyfile.  Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ^Z. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a ^D which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The ^Z should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in the section covering ***Jobs; Background, Foreground, or Suspended.***

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

```
Quit (Core dumped)
```

indicating that a file 'core' has been created containing information about the running program's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in Jobs; Background, Foreground, or Suspended) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See the section on ***Jobs; Background, Foreground, or Suspended*** for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

```
      cat /etc/passwd
```

command will, you can use the command

```
      more /etc/passwd
```

The *more* program pauses after each complete screenful and types '--More--' at which point you can hit a space to get another screenful, a return to get another line, a '?' to get some help on other commands, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

```
      cat /etc/passwd | more
```

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ^S key to stop the typeout. The typeout will resume when you hit ^Q or any other key, but ^Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ^S and ^Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ^O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ^O is a toggle, so flushing can be turned off by typing ^O again while output is being flushed.

What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
      chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. You only have to do this once; it takes effect at next login. You are now ready to try using *csh.*

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

DETAILS ON THE SHELL FOR TERMINAL USERS

Shell Startup and Termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users ; users
alias ts \
  'set noglob ; eval \'tset -s -m dialup:c100rv4pna -m \
  plugboard:?hp2621nl \!*\'';
ts; stty intr ^C kill ^U crt
set time=15 history=10
msgs -f
if (-e $mail)  then
     echo "${prompt}mail"
     mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ^D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set*; this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ^C and the line kill character to ^U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '-f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

Shell Variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set*) in the shell. The default value for path will be shown by *set* to be

```
% set
argv      ()
cwd       /usr/bill
home      /usr/bill
path      (.  /usr/ucb /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
term      c100rv4pna
user      bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

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

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
      set
```

again to see that the value assigned to *path* has changed.

Another directory that might interest you is /usr/new, which contains many useful user-contributed programs provided with Berkeley UNIX.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
      rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset.* Thus to set this variable you simply do

```
      set ignoreeof
```

and to unset it do

```
      unset ignoreeof
```

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail.* The metasyntax

```
    > filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
    set noclobber
```

in your *.login* file. Then trying to do

```
    date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
    date >!  now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.[3]

The Shell's History List

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following example gives a sample session involving typical usage of the history mechanism of the shell.

[3] The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

```
% cat bug.c
main()
{
     printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4:  newline in string or char constant
"bug.c", line 5:  syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
     printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
     printf("hello\n");
w
32
q % !c -o bug
cc bug.c -o bug
% size a.out bug
a.out:  2784+364+1028 = 4176b = 0x1050b
bug:  2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x  1 bill       3932 Dec 19 09:41  a.out
-rwxr-xr-x  1 bill       3932 Dec 19 09:42  bug
% bug
hello
% num bug.c | spp
spp:  Command not found.
% ^spp^ssp
num bug.c | ssp
   1  main()
   3  {
   4       printf("hello\n");
   5  }
% !!  | lpr
num bug.c | ssp | lpr
%
```

In the example on the previous page we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '-o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls -l' command with the same argument list, denoting the argument list '\!*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '^' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmer's Manual.

Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
      alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

```
      mail bill
```

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '-s'. We can do

```
      alias ls ls -s
```

or even

```
      alias dir ls -s
```

creating a new command syntax 'dir' which does an 'ls -s'. If we say

```
      dir ~bill
```

then the shell will translate this to

```
      ls -s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

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

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in ''' characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

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

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 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.

More Redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.[4]

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.[5]

Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

---

[4] A command of the form

```
command >&!  file
```

exists, and is used when *noclobber* is set and *file* already exists.

[5] If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

```
command >>!  file
```

makes it not be an error for file to not exist when *noclobber* is set.

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

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without out waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] - Done          du > usage
%
```

If the job did not terminate normally the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the *notify* variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are re-used.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

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

runs the 'ls' program with the '-s' options, pipes this output into the 'sort' program with the '-n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediates prompts for a new command, leaving the job running simultaneously.

As mentioned in the section covering ***Terminating Commands***, foreground jobs become *suspended* by typing ^Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
^Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

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

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the *bg* command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
^Z
Stopped
% bg
[1] du > usage &
%
```

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job - identified by a '-' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%-' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '-l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls -s | sort -n > myfile &
[2] 3405
% mail bill
^Z
Stopped
% jobs
[1] - Running            du > usage
[2] Running              ls -s | sort -n > myfile
[3] + Stopped            mail bill
% fg %ls
ls -s | sort -n > myfile
% more myfile
```

The *fg* command 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). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

```
    % kill %1
    [1] Terminated           du > usage
    %
```

The *notify* command (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.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```
    % ed bigfile
    120000
    1,$s/thisword/thatword/
    ^Z
    Stopped
    % bg
    [1] ed bigfile &
    %
    ... some foreground commands
    [1] Stopped (tty input)   ed bigfile
    % fg
    ed bigfile
    w
    120000
    q
    %
```

So after the 's' command was issued, the 'ed' job was stopped with ^Z and then put in the background using *bg*. Some time later when the 's' command was finished, *ed* tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The *fg* command returned the 'ed' job to the foreground where it could once again accept commands from the terminal.

The command

```
    stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
... some time later
q
[1] Stopped (tty output)       wc hugefile
% fg wc
wc hugefile
   13371   30123   302577
% stty -tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since 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* can be used in this case to find out about background jobs not started in the current shell.

Working Directories

As mentioned in the section covering ***Filenames***, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir,* creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory *newpaper*. where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just

```
cd
```

with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable *cwd*. The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command *pushd* is used in place of the *cd* command, the shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this list at any time by typing the 'directories' command *dirs*.

```
% pushd newpaper/references
~/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
% dirs
/usr/lib/tmac ~/newpaper/references ~
% popd
~/newpaper/references ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. 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 *csh* manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs -l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
^Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd:  ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no cd command was issued. In the above example the 'ed' job was still in '/mnt/bill/project' even though the shell had changed to '/mnt/bill'. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd:  ~/myproject)
... after some editing
q
(wd now:  ~)  %
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The '-l' option of *jobs* will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

Useful Built-in Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

The *echo* command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*. By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

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

Note that the '!' character had to be *escaped* here even within '' characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

```
cputime        unlimited
filesize       unlimited
datasize       5616 kbytes
stacksize      512 kbytes
coredumpsize   unlimited
```

Limits can be set, e.g.:

```
limit coredumpsize 128k
```

Most reasonable units abbreviations will work; see the *csh* manual page for more details.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable TERM to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect right away.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
   52    178   1347   /etc/rc
   52    178   1347   /usr/bill/rc
   104   356   2694   total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

Conclusion

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the csh manual pages (section1) to become familiar with the other facilities which are available to you.

SHELL CONTROL STRUCTURES AND COMMAND SCRIPTS

Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

Invocation and the *argv* Variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
    echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
    $?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
    $#name
```

expands to the number of elements in the variable *name*. Thus

```
    % set argv=(a b c)
    % echo $?argv
    1
    % echo $#argv
    3
    % unset argv
    % echo $?argv
    0
    % echo $argv
    Undefined variable:  argv.
    %
```

It is also possible to access the components of a variable which has several values. Thus

```
    $argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
    $argv[$#argv]
```

would give 'c', and

```
    $argv[1-2]
```

would give 'a b'. Other notations useful in shell scripts are

```
    $n
```

where *n* is an integer as a shorthand for

```
    $argv[n]
```

the *nth* parameter and

```
    $*
```

which is a shorthand for

```
    $argv
```

The form

```
    $$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
    $<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
    echo 'yes or no?\c'
    set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (^D) was typed.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if *n* is not in the range '1-$#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n-'; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm-n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the 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 *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
    -?  filename
```

where '?' is replace by a number of single characters. For instance the expression primitive

```
    -e filename
```

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

Sample Shell Script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

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

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

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

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

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

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

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

The placement of the keywords here is **not** flexible due to the current implementation of the shell.[6]

The shell does have another form of the if statement of the form

```
if ( expression )  command
```

which can be written

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

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif,* e.g.:

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

else
     commands
endif
```

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier ':r' here to extract a root of a filename or ':e' to extract the *extension.* Thus if the variable *i* has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

6 The following two formats are not currently acceptable to the shell:

```
if ( expression )       # Won't  work!
then
     command
     ...
endif
```

and

```
if ( expression )  then command endif   # Won't  work
```

shows how the ':r' modifier strips off the trailing '.bar' and the the ':e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the User's Reference Manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.[7] Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using ''' or '\' to place it in an argument word.

Other Control Structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

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

and

```
switch ( word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

7 It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
     commands
     goto loop
```

Supplying Input to Commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

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

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in ''' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

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

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
    onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a ‘goto label’ and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
    exit(1)
```

e.g. to exit with status ‘1’.

Other Shell Features

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can be used to help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character ‘#’, that is shell scripts that do not begin with a comment. Similarly, the ‘/bin/sh’ on your system may well defer to ‘csh’ to interpret shell scripts which begin with ‘#’. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using ‘"’ which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as ‘’’ does.

OTHER, LESS COMMONLY USED, SHELL FEATURES

Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$'  '-v sh$')
?  grep -c $i /etc/passwd
?  end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(\'ls\`)
% echo $a
csh.n csh.rm
% ls
csh.n csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '\'' characters is converted by the shell to a list of words. You can also place the '\'' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

Braces { ... } in Argument Expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ...  AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. 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 expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

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

Command Substitution

A command enclosed in '\'' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=\'pwd\'
```

to save the current directory in the variable *pwd* or to do

```
ex \'grep -l TRACE 8.c\'
```

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.[8]

[8] Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

Other Details Not covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the csh(1) manual section for a list of these options.

GLOSSARY

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX User Reference manual in section 1. You can look at an online copy of its manual page by doing

```
man 1 pr
```

References of the form (See ***More Redirection; >> and >&***) indicate that more information can be found in the section covering More Redirection; >> and >& of this manual.

.

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 which are in '.' are found first (See ***Shell Variables***). 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 (See ***Filenames***).

..

Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir*, i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (See ***Working Directories***).

a.out

Compilers which create executable images create them, by default, in the file *a.out.* for historical reasons (See ***The Shell's History List***).

absolute pathname

A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system - called the *root* directory. *Pathname s* which are not *absolute* are called *relative* (see definition of *relative pathname*) (See ***Filenames***).

alias

An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (See ***Aliases***).

argument Commands in UNIX receive a list of *argument* words. Thus the command

```
echo a b c
```

consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (See ***The Basic Notion of Commands***).

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 (See ***Variable Substitution***).

background Commands started without waiting for them to complete are called *background* commands (See ***Jobs; Background, Foreground, or Suspended***).

base A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* - the part after the '.'. See the definitions for *filename* and *extension* and basename (1)(See ***Filenames***).

bg The *bg* command causes a *suspended* job to continue execution in the *background* (See ***Jobs; Background, Foreground, or Suspended***).

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' which 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 shell (See ***Other Control Structures***).

breaksw The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (See ***Other Control Structures***).

builtin A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the 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 shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh (1)' (See ***Other Control Structures***).

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' (See ***Terminating Commands*** and ***The Shell's History List***).

cd The *cd* command is used to change the *working directory* . With no arguments, *cd* changes your *working directory* to be your *home* directory (See ***Aliases*** and ***Working Directories***).

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 which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing

```
chsh your-login-name /bin/csh
```

Thus I would do

```
chsh bill /bin/csh
```

It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (See the ***What Now?*** section under ***TERMINAL USAGE OF THE SHELL***).

cmp *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (See ***Sample Shell Script***). For comparing text files the program *diff* , described in 'diff (1)' 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 within the UNIX system, is called a *command* (See ***The Basic Notion of Commands***).

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 on UNIX is that the first word of a command names the function to be performed (See ***The Basic Notion of Commands***).

command substitution
The replacement of a command enclosed in '\`' characters by the text output by that command is called *command substitution* (See ***Command Substitution***).

componenet A part of a *pathname* between '/' characters is called a *component* of that *pathname* . A variable which has multiple strings as value is said to have several *component s;* each string is a *component* of the variable.

continue	A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (See ***Sample Shell Script***).
control–	Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control- c* is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an caret (^) followed by the corresponding letter when you type a *control* character (e.g. '^C' for *control- c* (See ***Terminating Commands***).
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 (1)' or 'sdb (1)' in order to determine what went wrong with the program (See ***Terminating Commands***). 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 UNIX commands (See ***Filenames***).
csh	The name of the shell program that this document describes.
.cshrc	The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (See ***Shell Startup and Termination***).
cwd	The *cwd* variable in the shell holds the *absolute pathname* of the *current working directory* . It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (See ***Shell Variables***).
date	The *date* command prints the current date and time (See ***Output to Files***).
debugging	*Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging*.
default:	The label *default:* is used within 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 (See ***Other Control Structures***).

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 ^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 (See Output to Files, See Input from Files; Pipelines). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (See ***More Redirection; >> and >&***).
directory	A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (See ***The Basic Notion of Commands*** and ***Working Directories***).
directory stack	The 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 (See ***Working Directories***).
dirs	The *dirs* command prints the shell's *directory stack* (See ***Working Directories***).
du	The *du* command is a program (described in 'du (1)') which prints the number of disk blocks is all directories below and including your current *working directory* (See ***Jobs; Background, Foreground, or Suspended***).
echo	The *echo* command prints its arguments (See ***Filenames*** and ***Sample Shell Script***).
else	The *else* command is part of the 'if-then-else-endif' control command construct (See ***Sample Shell Script***).
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 (See ***Sample Shell Script***).

EOF An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which 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 shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (See ***The Basic Notion of Commands***, ***Terminating Commands***, and ***Supplying Input to Commands***).

escape A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escape s* '*' (See ***Quotation***). There is also a non-printing character called *escape* , usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended* . Most systems use control-s to stop the output and control-q to start it.

/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 (See ***Terminating Commands***). You can look at this file by saying

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See 'finger (1)', 'passwd(5)', and 'grep (1)' for more details.

exit The *exit* command is used to force termination of a shell script, and is built into the shell (See ***Supplying Input to Commands***).

exit status A command which discovers a problem may reflect this back to the command (such as a shell) which 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 shell command script to give a non-zero *exit status* (See ***Sample Shell Script***).

expansion The replacement of strings in the shell input which 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* (See ***Filenames***, ***Variable Substitution***, and ***Braces { ... } in Argument Expansion***).

expressions *Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (See ***Expressions***).

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 of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (See ***Filenames***).

fg The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (See ***Terminating Commands*** and ***Jobs; Background, Foreground, or Suspended***).

filename Each file in UNIX 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* (See ***Filenames***).

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 which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (See ***Filenames*** and ***Braces { ... } in Argument Expansion***).

flag Many UNIX commands accept arguments which 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 '-' (See ***Flag Arguments***). 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 shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (See ***Sample Shell Script*** and ***Loops at the Terminal; Variables as Vectors***).

foreground When commands are executing in the normal way such that the 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 (See ***Terminating Commands*** and ***Jobs; Background, Foreground, or Suspended***).

goto The shell has a command *goto* used in shell scripts to transfer control to a given label (See ***Other Control Structures***).

grep The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print 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 (1)' and 'ex (1)'. *Grep* stands for 'globally find *regular expression* and print' (See ***Aliases***).

head The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (See ***Input from Files; Pipelines***).

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 (See ***Sample Shell Script***).

history The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (See ***The Shell's History List***).

home directory
Each user has a *home directory* , which is given in your entry in the password file, */etc/passwd* . This is the directory which 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 '~' (See ***Filenames***).

if A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (See ***Sample Shell Script***).

ignoreeof Normally, your shell will exit, printing ‘logout’ if you type a control-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 control-d characters, logging yourself off (See ***Shell Variables***).

input Many commands on UNIX take information from the terminal or from files which 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 will also read from a file specified as argument. Commands placed in *pipelines* will 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 (See ***Input from Files; Pipelines*** and ***Supplying Input to Commands***).

interrupt An *interrupt* is a signal to a program that is generated by typing ^C. (On older versions of UNIX the RUBOUT or DELETE key were used for this purpose.) It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (See ***Terminating Commands*** and ***Supplying Input to Commands***).

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* (See ***Jobs; Background, Foreground, or Suspended***).

job control The builtin functions that control the execution of jobs are called *job control* commands. These are *bg, fg, stop, kill* (See ***Jobs; Background, Foreground, or Suspended***).

job number When each job is started it is assigned a small number called a *job number* which 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 (See ***Jobs; Background, Foreground, or Suspended***).

jobs The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (See ***Jobs; Background, Foreground, or Suspended***).

kill	A command which sends a signal to a job causing it to terminate (See ***Jobs; Background, Foreground, or Suspended***).
.login	The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (See ***Shell Startup and Termination***).
login shell	The shell that is started on your terminal when you login is called your *login shell* . It is different from other shells which 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 (See ***Shell Startup and Termination***).
logout	The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end\fI*-of\f1-file, but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (See ***Useful Built-In Commands***).
.logout	When you log off of UNIX the shell will execute 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 UNIX 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* (See ***The Shell's History List***).
ls	The *ls* (list files) command is one of the most commonly used UNIX 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 (See ***Flag Arguments***).
mail	The *mail* program is used to send and receive messages from other UNIX users (See ***The Basic Notion of Commands*** and ***Shell Startup and Termination***), whether they are logged on or not.
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 (See ***Make***).
makefile	The file containing commands for *make* is called *makefile* or *Makefile* (See ***Make***).

manual The *manual* often referred to is the 'UNIX manual'. It contains 8 numbered sections with a description of each UNIX program (section 1), system call (section 2), subroutine (section 3), device (section 4), special data structure (section 5), game (section 6), miscellaneous item (section 7) and system administration program (section 8). There are also supplementary documents (tutorials and reference guides) for individual programs which require explanation in more detail. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

If you can't decide what manual page to look in, try the *apropos (1)* command. The supplementary documents are in subdirectories of /usr/doc.

metacharacter
Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. 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 '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words (See Metacharacters in the Shell). The appendix to this user's manual lists the *metacharacters* in groups by their function.

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 (See ***Sample Shell Script***).

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 (See ***Terminating Commands***).

noclobber The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (See ***Shell Variables*** and ***More Redirection; >> and >&***).

noglob The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (See ***Sample Shell Script***).

notify The *notify* command tells the 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 shell to always report the termination of *background* jobs exactly when they occur (See ***Jobs; Background, Foreground, or Suspended***).

onintr The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (See ***Supplying Input to Commands***).

output Many commands in UNIX result in some lines of text which 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 shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file (See ***Output to Files***). 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 (See Input from Files; Pipelines). 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 (See ***The Shell's History List***). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (See ***The Shell's History List***). 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 (See ***More Redirection; >> and >&***).

path The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the 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 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 shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (See Shell Variables). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

```
chmod 755 script
```

was executed to turn this execute permission on (See Invocation and the argv Variable). If you add new commands to a directory in the *path* , you should issue the command *rehash* (See ***Shell Variables***).

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* which 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* may name a directory, but usually names a file.

pipeline A group of commands which 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 '|' (See ***Input from Files; Pipelines*** and ***The Shell's History List***).

popd The *popd* command changes the 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 (See ***Working Directories***).

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 (See ***The Shell's History List***).

printenv The *printenv* command is used to print the current setting of variables in the environment (See ***Useful Built-In Commands***).

process An instance of a running program is called a *process* (See ***Jobs; Background, Foreground, or Suspended***). UNIX 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 which performs a useful function is often called a *program* .

prompt Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex (1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (See ***The Basic Notion of Commands***). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt* . This is mostly used when debugging the shell (See ***Useful Built-In Commands***).

pushd The *pushd* command, which means 'push directory', changes the 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 (See ***Working Directories***).

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 (See ***Jobs; Background, Foreground, or Suspended***). 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 control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (See ***Terminating Commands***).

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* (See ***Quotation***).

redirection The routing of input or output from or to a file is known as *redirection* of input or output (See ***Output to Files***).

rehash The *rehash* command tells the 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 (See ***Useful Built-In Commands***).

relative pathname
A *pathname* which 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 *working directory* . *Pathnames* that are not *relative* are called *absolute pathnames* (See ***Filenames***).

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 (See ***Filenames***).

RUBOUT	The RUBOUT or DELETE key is often used to erase the previously typed character; some users prefer the BACKSPACE for this purpose. On older versions of UNIX this key served as the INTR character.
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 (See ***Output to Files***).
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 (See ***Invocation and the 'argv' Variable*** and ***Other Shell Features***).
set	The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (See ***Shell Startup and Termination***).
setenv	Variables in the environment 'environ (5)' can be changed by using the *setenv* builtin command (See ***Useful Built-In Commands***). 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. This manual deals with the details of one particular *shell* , called *csh*.
shell script	See *script* (See ***Invocation and the 'argv' Variable*** and ***Other Shell Features***).
signal	A *signal* in UNIX is a short message that is sent to a running program which 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 (See ***Terminating Commands*** and ***Jobs; Background, Foreground, or Suspended***).
sort	The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (See ***Input from Files; Pipelines***).
source	The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (See ***Useful Built-In Commands***).
special character	See *metacharacters* and the appendix to this manual.

standard	We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (See ***Output to Files*** and ***Supplying Input to Commands***).
status	A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (See ***Sample Shell Script***).
stop	The *stop* command causes a *background* job to become *suspended* (See ***Jobs; Background, Foreground, or Suspended***).
string	A sequential group of characters taken together is called a *string* . *Strings* can contain any printable characters (See ***Shell Variables***).
stty	The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty (1)' for a complete description (See ***Jobs; Background, Foreground, or Suspended***).
substitution	The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (See ***Variable Substitution***).
suspended	A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control -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 (See ***Jobs; Background, Foreground, or Suspended***).
switch	The *switch* command of the shell allows the 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 (See ***Other Control Structures***).
termination	When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end\fI*-of\f1-file from their *standard input* . It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (See ***Terminating Commands***). The *kill* program terminates specified jobs (See ***Jobs; Background, Foreground, or Suspended***).
then	The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (See ***Sample Shell Script***).

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 (See ***Shell Startup and Termination*** and ***Useful Built-In Commands***).

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 (See ***Shell Startup and Termination***).

tty The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.

unalias The *unalias* command removes aliases (See ***Useful Built-In Commands***).

UNIX UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.

unset The *unset* command removes the definitions of shell variables (See ***Shell Variables*** and ***Useful Built-In Commands***).

variable expansion
See *variables* and *expansion* (See ***Shell Variables*** and ***Variable Substitution***).

variables *Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing shell programs (shell command scripts) (See ***Shell Variables***).

verbose The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's *-v* command line option (See ***Other Shell Features***).

wc The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (See ***Jobs; Background, Foreground, or Suspended***).

while The *while* builtin control construct is used in shell command scripts (See ***Other Control Structures***).

word

A sequence of characters which forms an argument to a command is called a *word* . Many characters which are neither letters, digits, ‘-’, ‘.’ nor ‘/’ form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with ‘‘’ characters except for the characters ‘‘’ and ‘!’ which require special treatment (See ***The Basic Notion of Commands***). 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 an obsolete way of communicating with other users who are logged in to UNIX (you have to take turns typing). If you are both using display terminals, use *talk*(1), which is much more pleasant.

GLOSSARY

This glossary defines terms and acronyms used in the *ICON/UXV User Guide* that may not be familiar to you.

acoustic coupler
A device that permits transmission of data over an ordinary telephone line. When you place a telephone handset in the coupler, you link a computer at one end of the phone line to a peripheral device, such as a user terminal, at the other.

address
Generally, a number that indicates the location of information in the computer's memory. In the ICON/UXV system, the address is part of an editor command that specifies a line number or range.

append mode
A text editing mode where you enter (append) text after the current position in the buffer. See **text input mode**, compare with **command mode** and **insert mode**.

argument
Special instructions on the command line that specify data on which a command is to operate. Arguments usually follow the command and can include numbers, letters, or text strings. For instance, in the command **lp —m myfile**, **lp** is the command and **myfile** is the argument. See **option**.

ASCII
(pronounced **as'-kee**) American Standard Code for Information Interchange, a standard for data transmission that is used in the ICON/UXV system. ASCII assigns sets of 0s and 1s to represent 128 characters, including alphabetical characters, numerals, and standard special characters, such as #, $, %, and &.

AT&T 3B computers
Computers manufactured by AT&T Technologies, Inc.

background
A type of program execution where you request the shell to run a command away from the interaction between you and the computer ("in the background"). While this command runs, the shell prompts you to enter other commands through the terminal.

baud rate
A measure of the speed of data transfer from a computer to a peripheral device (such as a terminal) or from one device to another. Common baud rates are 300, 1200, 4800, and 9600. As a general guide, divide a baud rate by 10 to get the approximate number of English characters transmitted each second.

buffer
A temporary storage area of the computer used by text editors to make changes to a copy of an existing file. When you edit a file, its contents are read into a buffer, where you make changes to the text. For the changes to become a part of the permanent file, you must write the buffer contents back into the file. See **permanent file**.

child directory
See **subdirectory**.

command
The name of a file that contains a program that can be processed or executed by the computer on request.

command file
See **executable file**.

command language interpreter
A program that acts as a direct interface between you and the computer. In the ICON/UXV operating system, a program called the **shell** takes commands and translates them into a language understood by the computer.

command line
A line containing one or more commands, ended by typing a carriage return (<CR>). The line may also contain options and arguments. You type this line to the shell to instruct the computer to perform one or more tasks.

command mode
A text editing mode in which each character you type is interpreted as an editing command. This mode permits actions such as moving around in the buffer, deleting text, or moving lines of text. See **text input mode**, compare with **append mode** and **insert mode**.

context search
A technique for locating a specified pattern of characters (called a string) when in a text editor. Editing commands that cause a context search scan the buffer, looking for a match with the string specified in the command. See **string**.

control character
A nonprinting character that is entered by holding down the control key and typing a character. A control character transmits a special command to the computer. For instance, when viewing a long file on your screen with the **cat** command, typing control-s (^**s**) stops the display so you can read it, and typing control-q (^**q**) continues the display.

current directory
The directory in which you are presently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is a dot (.).

cursor
A cue printed on the terminal screen that indicates the position at which you enter or delete a character. It is usually a rectangle or a blinking line.

default
An automatically assigned value or condition that exists unless you explicitly change it. For example, the shell prompt string has a default value of $ unless you change it.

delimiter
A character that logically separates items or arguments on a command line. Two frequently used delimiters in the ICON/UXV operating system are the space and the tab. Another is the slash character (/) that separates directories from subdirectories and files in a path name.

diagnostic
A message printed at your terminal to indicate an error encountered while trying to execute some command or program. Generally, you need not respond directly to a diagnostic message.

directory
A type of file used to group and organize other files or directories. You cannot enter text or other data into a directory. (For more detail, see *Appendix B, File System Organization.*)

disk
A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.

electronic mail
The feature of an operating system that allows computers users to exchange written messages via the computer. The ICON/UXV operating system **mail** command provides electronic mail in which the addresses are the login names of users.

environment
The conditions under which you work while using the ICON/UXV operating system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the ICON/UXV system and the computer. For example, your shell environment includes such things as your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

erase character
The character you type to delete the previous character on the current line. The ICON/UXV system default erase character is #.

escape
A means of getting into the shell from within a text editor or another program.

execute
The computer's action of interpreting a programmed instruction or command and performing the indicated operation(s).

executable file
A file that can be processed or executed by the computer without any further translation. When you type in the file name, the commands in the file are executed. See **shell procedure**.

field
A word or a group of characters treated as one word on a command line. Fields are usually a fixed number of character positions in size, but they may also vary.

file
A collection of information. Files may contain data, programs, or other text. You access ICON/UXV files by name. See **ordinary file**, **permanent file**, and **executable file**.

file name
A sequence of characters that denotes a file. (In the ICON/UXV system, a slash character (/) cannot be used as part of a file name.)

file system
A collection of files and the structure that links them together. The file system is a hierarchical structure--that is, a ranked system of files. (For more detail, see *Appendix B, File System Organization.*)

filter
A command that reads the standard input, acts on it in some way, and then prints the result as standard output.

final copy
The completed, printed version of a file of text.

foreground
The normal type of program execution. In foreground mode, the shell waits for a command to end before prompting you for another command. In other words, you enter something into the computer and the computer "replies" before you enter something else.

full-duplex
A type of data communication in which a computer system can transmit and receive data simultaneously. Terminals and modems usually have settings for half-duplex (one-way) and full-duplex communication; the ICON/UXV system uses the full-duplex setting.

full path name
A path name that originates at the root directory of the ICON/UXV system and leads to a specific file or directory. Each file and directory in the ICON/UXV system has a unique full path name, sometimes called an absolute path name. See **path name**.

global
A qualifier that indicates the complete or entire file. While normal editor commands commonly act on only the first instance of a pattern in the file, global commands perform the action on all instances in the file.

hardware
The physical machinery of a computer and any associated devices.

hidden character
One of a group of characters within the standard ASCII set, but not normally printed as visible symbols. Control characters, such as backspace and escape, are examples.

home directory
The directory in which you are located when you log in to the ICON/UXV system; also known as your login directory.

ICON/UXV system
A general-purpose, multiuser, interactive, time-sharing operating system developed by ICON International, Inc. The ICON/UXV system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

input/output
The path by which information enters a computer system (input) and leaves the system (output). An input device that you use is the keyboard and an output device is the terminal monitor.

insert mode
A text editing mode in which you enter (insert) text before the current position in the buffer. See **text input mode**, compare with **append mode** and **command mode**.

interactive
Describes an operating system (such as the ICON/UXV system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

line editor
An editing program in which text is operated upon on a line-by-line basis within a file. Commands for creating, changing, and removing text use line addresses to determine where in the file the changes are made. Changes can be viewed after they are made by displaying the lines changed. See **text editor**, compare with **screen editor**.

login
The procedure used to gain access to the ICON/UXV operating system.

login directory
See **home directory**.

login name
A string of characters used to identify a user. Your login name is different from other login names.

log off
The procedure used to exit from the ICON/UXV operating system.

metacharacter
One of a group of characters with a special meaning to the **shell**, such as < > * ? ¦ & $; ( ) \ " ´ [ ] .

mode
In general, a particular type of operation (for example, an editor's append mode). In relation to the file system, a mode is an octal number used to determine who can have access to your files and what kind of access they can have. See **permissions**.

modem
A device that connects a terminal and a computer by way of a telephone line. A modem converts digital signals to tones and converts tones back to digital signals, allowing a terminal and a computer to exchange data over standard telephone lines.

multitasking
The ability of an operating system to execute more than one program at a time.

multiuser
The ability of an operating system to support several users on the system at the same time.

nroff
A text formatter available as an add-on to the ICON/UXV system. You can use the **nroff** program to produce a formatted on-line copy or a printed copy of a file. See **text formatter**.

operating system
The software system on a computer under which all other software runs. The ICON/UXV system is an operating system.

option
Special instructions that modify how a command runs. Options are a type of argument that follow a command and are preceded by a minus sign (—). You can specify more than one option for any command given in the ICON/UXV system. For example, in the command **ls —l —a directory**, **—l** and **—a** are options that modify the **ls** command. See **argument**.

ordinary file
A collection of one to several thousand characters. Ordinary files may contain text or other data but are not executable. See **executable file**.

output
Information processed in some fashion by a computer and delivered to you by way of a printer, a terminal, or a similar device.

parameter
Generally, a value that determines the characteristics or behavior of something. In the ICON/UXV system, a type of variable found only on the command line. See **variable**.

parent directory
The directory immediately above a subdirectory or file in the file system organization. The shorthand notation for the parent directory is two dots (..).

parity
A method used by a computer for checking that the data received matches the data sent.

password
A code word known only to you that is called for in the login process. The computer uses the password to verify that you may indeed use the system.

path name
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The path name defines the connection path between some directory and a file.

peripheral device
Auxiliary devices under the control of the main computer, used mostly for input, output, and storage functions. Some examples include terminals, printers, and disk drives.

permanent file
The data stored permanently in the file system structure. To change a permanent file, you must make use of a text editor, which maintains a temporary work space, or buffer, apart from the permanent files. Once changes have been made to the buffer, they must be written to the permanent file to make the changes permanent. See **buffer**.

permissions
Access modes, associated with directories and files, that permit or deny system users the ability to read, write, and/or execute the directories or files. You determine the permissions for your directories or files by changing the mode for each one with the **chmod** command.

pipe
A method of redirecting the output of one command to be the input of another command. It is named for the character (¦) that redirects the output. For example, the shell command **who ¦ wc –l** pipes output from the **who** command to the **wc** command, telling you the total number of people logged into your ICON/UXV system.

pipeline
A series of filters separated by the pipe character (¦). The output of each filter becomes the input of the next filter in the line. The last filter in the pipeline writes to its standard output. See **filter**.

positional parameters
Variables that hold arguments supplied with a shell procedure. They are placed into variable names, such as **$1**, **$2**, and **$3** when the shell calls for the shell procedure. The name of the shell procedure is positional parameter **$0**. See **variable** and **shell procedure**.

prompt
A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The ICON/UXV system default prompt is the dollar sign character ($).

printer
An output device that prints the data it receives from the computer on paper.

process
Generally a program that is at some stage of execution. In the ICON/UXV system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current directory, status of files, information recorded at login time, and various other items.

program
The instructions given to a computer on how to do a specific task. Programs are user-executable software.

read-ahead capability
The ability of the ICON/UXV system to read and interpret your input while sending output information to your terminal in response to previous input. The ICON/UXV system separates input from output and processes each correctly.

relative path name
The path name to a file or directory which varies in relation to the directory in which you are currently working.

remote system
A system other than the one on which you are working.

root
The source of all files and directories in the file system, designated by a slash character (/).

screen editor
An editing program in which text is operated on relative to the position of the cursor on a visual display. Commands for entering, changing, and removing text involve moving the cursor to the area to be altered and performing the necessary operation. Changes are viewed on the terminal display as they are made. See **text editor**, compare with **line editor**.

search pattern
See **string**.

search string
See **string**.

secondary prompt
A cue displayed at your terminal by the shell to tell you that the command typed in response to the primary prompt is incomplete. The ICON/UXV system default secondary prompt is the "greater than" character (>).

shell
An ICON/UXV program that handles the communication between you and the computer. The shell is also known as a command language interpreter because it translates your commands into a language understandable by the computer. The shell accepts commands and causes the appropriate program to be executed.

shell procedure
An executable file that is not a compiled program. A shell procedure calls the shell to read and execute commands contained in a file. This lets you store a sequence of commands in a file for repeated use. It is also called a command file. See **executable file**.

silent character
See **hidden character**.

software
Instructions and programs that tell the computer what to do. Contrast with **hardware**.

source code
The English-language version of a program. The source code must be translated to machine language by a program known as a compiler before the computer can execute the program.

special character
See **metacharacter**.

special file
A file (called a device driver) used as an interface to an input/output device, such as a user terminal, a disk drive, or a line printer.

standard input
An open file that is normally connected directly to the keyboard. Standard input to a command normally goes from the keyboard to this file and then into the shell. You can redirect the standard input to come from another file instead of from the keyboard; use an argument in the form < **file**. Input to the command will then come from the specified file.

standard output
An open file that is normally connected directly to a primary output device, such as a terminal printer or screen. Standard output from the computer normally goes to this file and then to the output device. You can redirect the standard output into another file instead of to the printer or screen; use an argument in the form > **file**. Output will then go to the specified file.

string
Designation for a particular group or pattern of characters, such as a word or phrase, that may contain special characters. In a text editor, a context search interprets the special characters and attempts to match a specified string with an identical string in the editor buffer.

string value
A specified group of characters that is symbolized to the shell by a variable. See **variable**.

subdirectory
A directory pointed to by a directory one level above it in the file system organization; also called a child directory.

system administrator
The person who monitors and controls the computer on which your ICON/UXV system runs; sometimes referred to as a super-user.

terminal
An input/output device connected to a computer system, usually consisting of a keyboard with a video display or a printer. A terminal allows you to give the computer instructions and to receive information in response.

text editor
Software for creating, changing, or removing text with the aid of a computer (known as text processing). Most text editors have two modes--an input mode for typing in text, and a command mode for moving or modifying text. Two examples are the ICON/UXV editors **ed** and **vi**. See **line editor** and **screen editor**.

text formatter
A program that prepares a file of text for printed output. To make use of a text formatter, your file must also contain some special commands for structuring the final copy. These special commands tell the formatter to justify margins, start new paragraphs, set up lists and tables, place figures, and so on. Two text formatters available as add-ons to your ICON/UXV system are **nroff** and **troff**.

text input mode
A text editing mode where the text you type is added into the buffer. To execute a command, you must leave the input mode. See **command mode**, compare with **append mode** and insert mode.

timesharing
A method of operation in which several users share a common computer system seemingly simultaneously. The computer interacts with each user in sequence, but the high-speed operation makes it seem that the computer is giving each user its complete attention.

tool
A package of software programs.

troff
A text formatter available as an add-on to the ICON/UXV system. The **troff** program drives a phototypesetter to produce high-quality printed text from a file. See **text formatter**.

tty
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

user
Anyone who uses a computer or an operating system.

user-defined
Something determined by the user.

user-defined variable
A shell name given by the user for the value of a string of characters. See **variable**.

utility
Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.

variable
A symbol whose value may change within a program or a repetition of a program. In the shell, a variable is a name representing some string of characters (a **string value**). Some variables are normally set only on a command line and are called **parameters** (**positional parameters** and **keyword parameters**). Other variables are simply names to which the user (**user-defined variables**) or the shell itself may assign string values. (Keyword parameters are discussed fully in *ICON/UXV System Shell Commands and Programming*; see description in *Appendix A*.)

video display terminal
A terminal that uses a televisionlike screen (a monitor) to display information. A video display terminal can display information much faster than printing terminals.

visual editor
See **screen editor**.

working directory
See **current directory**.

© Copyright1988
Icon International, Inc.
All rights reserved worldwide.

Printed in the U.S.A. 171-063-003 A1