305-506
Issue 1

AT&T 3B2 Computer
UNIX™ System V Release 2.0

Utilities – Volume 1

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

TRADEMARKS

The following is a listing of the trademarks that are used in these three volumes:

- APS — Trademark of Autologic, Inc
- DIABLO — Registered Trademark of XEROX Corporation
- DOCUMENTER'S WORKBENCH — Trademark of AT&T
- Dimension is a Trademark of AT&T.
- HP — Trademark of Hewlett-Packard, Inc
- Hayes is a Trademark of Hayes Microcomputer Products, Inc.
- Micom is a Registered Trademark of Micom Systems, Inc.
- Penril is a Registered Trademark of Penril Corporation, Inc.
- Rixon is a Registered Trademark of Rixon, Inc.
- TEKTRONIX — Registered Trademark of Tektronix, Inc
- TELERAY — Trademark of Research Incorporated
- TELETYPE — Trademark of AT&T
- UNIX — Trademark of AT&T
- Ventel is a Registered Trademark of Ven-Tel, Inc.
- Versatec — Registered Trademark of Tektronix, Inc.

ORDERING INFORMATION

Additional copies of this document can be ordered by calling

1-800-432-6600 Inside the U.S.A.

OR

1-317-352-8557 Outside the U.S.A.

OR by writing to:

AT&T Customer Information Center (CIC)
Attn: Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219

Replace this

page with the

INTRODUCTION

tab separator.

Chapter 1

INTRODUCTION

The Utilities Guides for thirteen utilities are packaged in these three-volumes, titled *AT&T 3B2 Computer Utilities - Volume 1 through 3.*

Security Administration capability and supporting documentation is restricted to United States customers only. The *Security Administration Utilities Guide* will be provided to United States customers as a stand-alone document. Instructions at the beginning of the stand-alone guide tell you to place the guide, and the TAB provided, in Volume 3 of the AT&T 3B2 Computer Utilities.

Source Code Control System (SCCS) Utilities are optional utilities. If ordered, you will receive an SCCS Utilities Guide. Instructions at the beginning of the SCCS Utilities Guide tell you to place the guide, and the TAB provided, anywhere you wish in the three-volume Utilities Guide.

The overall index, located at the end of Volume 3, includes keyword entries from the Security Administration and the SCCS documents.

The Manual Pages for each utilities are located in the "*AT&T 3B2 Computer User Reference Manual*," the "*AT&T 3B2 Computer Programmer Reference Manual*," or the "*AT&T 3B2 Computer System Administration Reference Manual*," whichever is appropriate.

The three-volume set contains the following:

Volume 1	INTRODUCTION BASIC NETWORKING CARTRIDGE TAPE DIRECTORY AND FILE MANAGEMENT EDITING
Volume 2	GRAPHICS HELP INTER-PROCESS COMMUNICATION
Volume 3	LINE PRINTER SPOOLING PERFORMANCE MEASUREMENTS SPELL TERMINAL FILTERS TERMINAL INFORMATION USER ENVIRONMENT INDEX

The Index for all utilities are located at the end of Volume 3. The index will have the Item Name, Utilities Code, Chapter Number, and Page Number.

The Utilities Codes are:

UTILITIES NAME	UTILITIES CODE
Basic Networking	BN
Cartridge Tape	CT
Directory and File Management	DF
Editing	ED
Graphics	GR
Help	HP
Inter-Process Communication	IP
Line Printer Spooling	LP
Performance Measurements	PM
Security Administration	SA
Source Code Control System (SCCS)	SC
Spell	SP
Terminal Filters	TF
Terminal Information	TI
User Environment	UE

An example would be:

ACU; problems ..... BN 6-2

This means you can find ACU problems under the BASIC NETWORKING TAB in Chapter 6 on Page 2, or page BN 6-2.

Another example would be:

Joining Lines in vi ..... ED 4-21

Here, if you turn to the TAB marked EDITING, go to the page marked ED 4-21, you will find information on joining lines using the vi editor.

GUIDE BASELINE

This three-volume set contains sample displays that will help you understand described procedures. The sample displays in this document and the displays on your terminal screen may differ slightly due to improvements in the product after this three-volume set was finalized. Therefore, use the displays in the various utilities as samples of the type of data available. However, the data displayed on your terminal screen accurately reflects the sofware on your computer.

Replace this

page with the

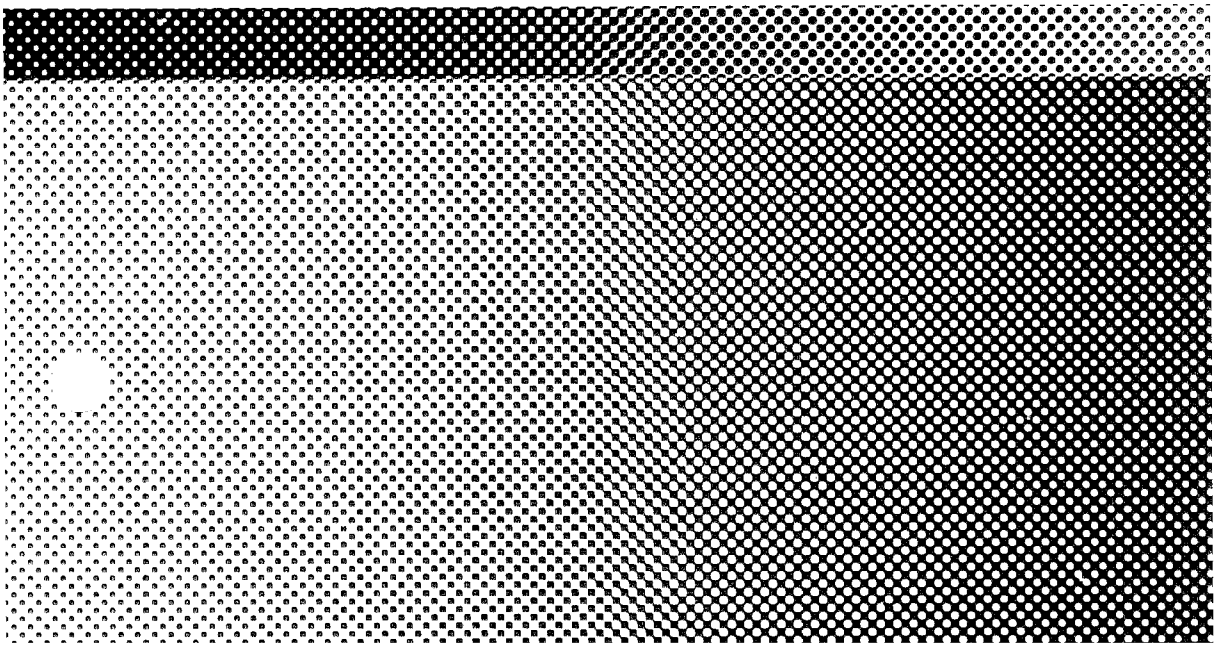
BASIC NETWORKING

tab separator.

AT&T

AT&T 3B2 Computer

UNIX™ System V Release 2.0 Basic Networking Utilities Guide



CONTENTS

Chapter 1

INTRODUCTION

PAGE

Chapter 1

INTRODUCTION

GENERAL

This guide describes the operation, administration, and command format (syntax) of the Basic Networking Utilities. Most of the material discussed is intended to be used by a sophisticated user who needs to know how to administer and maintain the utilities. Lower level (Novice) users may find the tutorial information (Chapter 7) helpful in understanding how the various commands are used to communicate with other machines.

Refer to the *AT&T 3B2 User Reference Manual* for **UNIX*** System V manual pages supporting the commands described in this guide.

---

* Trademark of AT&T

GUIDE ORGANIZATION

This guide is structured so you can easily find information without having to read the entire text. The remainder of this guide is organized as follows:

- Chapter 2, "OVERVIEW," contains an overview of the Basic Networking Utilities. It discusses the hardware and software associated with the operation of the networking system. After the hardware and software are introduced, a description of how the various programs work together to communicate with other machines is presented.

- Chapter 3, "ADMINISTRATION," discusses the administration of the Basic Networking Utilities. The discussion includes the files associated with the everyday operation of the system, supporting data base files, and administrative tasks.

- Chapter 4, "SIMPLE ADMINISTRATION," describes the Simple Administration feature of the Basic Networking Utilities (**uucpmgmt**).

- Chapter 5, "DIRECT LINKS," contains information on establishing a direct link from your AT&T 3B2 Computer to another 3B Computer. This includes some general information, as well as the specifics needed to set up the Basic Networking software and establish the direct link.

- Chapter 6, "MAINTENANCE," contains information on the maintenance of the Basic Networking Utilities. This includes some suggestions on debugging and a listing of error messages.

- Chapter 7, "COMMAND DESCRIPTIONS," describes the command format (syntax) of each command in the command set. The descriptions include the purpose of the command, a discussion on the command syntax and options, and examples of using the command.

DEFINITION OF TERMS

There may be terms used in this guide that you are not familiar with. Below is a list of such terms:

local machine	Refers to the machine on the "near" end of a communication link; normally, your 3B2 Computer.
remote machine	Refers to a machine on the "far" end of a communication link; normally, a machine that your 3B2 Computer calls.
active machine	A machine with the Basic Networking Utilities *and* the hardware required to establish communication links (i.e. Auto Dial Modem).
passive machine	A machine that has the Basic Networking Utilities, but does *not* have the hardware required to establish communication links.
network	A group of machines that are set up to exchange information and resources.
node	A terminating point (machine) on a network.
UUCP	This term (all caps) is used to show a group of programs and files that allow "unix-to-unix copy" capability between UNIX Systems. In general, it refers to all Basic Networking Utilities except the **cu** and **ct** programs. If "uucp" is shown in the text with bold type (**uucp**), this is referring specifically to the **uucp** program or login ID.

Chapter 2

OVERVIEW

PAGE

Chapter 2

OVERVIEW

This chapter contains an overview of the Basic Networking Utilities. The hardware and software of the utilities is introduced before discussing how it operates.

WHAT IS BASIC NETWORKING

The Basic Networking Utilities allow machines using the UNIX System to communicate with one another. The utilities allow you to:

- Transfer files and send electronic mail to other UNIX System machines as background processes.
- Interactively communicate with UNIX System machines and possibly non-UNIX System machines.
- Execute commands (restrictive) on a remote machine without logging in.

- Call a remote terminal and allow the user of the terminal to log in on your 3B2 Computer.

The later part of this chapter discusses the variety of ways information is transferred from one machine to another. It also discusses how commands are executed remotely, and how your 3B2 Computer can be requested to call a remote terminal. But first, you should become familiar with the hardware and software associated with the Basic Networking Utilities.

WHAT HARDWARE IS NEEDED

Before your 3B2 Computer can communicate with a remote machine, a communication link must be established to the remote machine. There are three types of hardware used to establish a communication link to another machine. The first is a direct link from a serial port on the 3B2 Computer to a serial port on the other machine. This type of connection is useful when two machines communicate with each other on a regular basis. Direct links allow data to be transferred at rates as high as 19200 bits per second (bps). Even though the RS-232 standard recommends that direct links be limited to 50 feet or less, two machines may be separated by several hundred feet if noise on the direct link does not become a problem. If noise becomes a problem or greater distance is needed between the two machines, the transfer rate may need to be decreased or limited distance modems placed at each end of the connection.

The Basic Networking Utilities does not contain the hardware needed to link your 3B2 Computer directly to remote machines. However, information on establishing a direct link to another machine and the parts needed can be found in Chapter 5, "DIRECT LINKS."

The second type of communication link uses the telephone network. In this type of link, the machine that establishes the connection (local machine) must have an automatic call unit (ACU). The ACU dials the specified phone number on request from the Basic Networking Utilities.

The called (remote) machine must have a telephone modem capable of answering incoming calls so other machines can contact it through the telephone network. If you wish to use the telephone network for establishing communication links, contact your AT&T Service Representative or authorized dealer for information on the AT&T Automatic Dial Modem.

The third type of communication link is established through a Local Area Network (LAN). Here, the 3B2 Computer must be a node on a LAN switch. This will allow the 3B2 Computer to establish a link to any machine connected to that LAN.

If a machine can establish a link to and request communication with another machine, it is considered an "active machine." Active machines must be able to establish links using one of the type of hardware mentioned above. A "passive machine" cannot establish a link to (call) a remote machine. However, a connection can be established *to* a passive computer if the passive computer has:

- A telephone modem that can automatically answer a call, or
- A direct link dedicated to serving incoming calls.

Note: If a machine is connected to a LAN, it is considered to be a active machine since it can call other machines on the LAN.

When a passive machine is called by an active machine, this is referred to as "polling." Polling passive machines is discussed later in more detail.

THE BASIC NETWORKING SOFTWARE

The Basic Networking Utilities is composed of software programs, daemons (background routines), and a supporting data base. The supporting data base contains support files that store information such as telephone numbers, location of the devices (hardware) used to establish links, security restrictions, etc. The software programs and a skeleton data base is supplied on the Basic Networking Utilities floppy diskette that comes with this guide. Since each 3B2 Computer will have an uncommon supporting data base for Basic Networking, the files that make up this data base are empty (except for comments) when they are loaded onto the hard disk. The Basic Networking Simple Administration feature is used to create unique entries in some of these files. The Simple Administration subcommands are discussed in Chapter 4, "SIMPLE ADMINISTRATION."

The Directories and Their Purpose

There are several directories that contain the programs and support files of Basic Networking. Some of these directories are uncommon to Basic Networking, while others are common to the UNIX System and the 3B2 Computer. The directories used by Basic Networking are listed below:

/usr/bin — This directory is used by the UNIX System to store executable programs and is used by Basic Networking for the same purpose.

/usr/admin/menu — This directory contains the Simple Administration subcommands for certain utilities. The subcommands for the Basic Networking Utilities are in the directory **/usr/admin/menu/packagemgmt/uucpmgmt**.

/usr/lib/uucp — This directory is the "HOME" directory for the **uucp** administrative log in. It contains the files of the supporting data base and some executable programs.

/usr/spool/locks	This directory contains the lock (LCK) files for the Basic Networking hardware devices. Lock files are discussed in Chapter 3.
/usr/spool/uucp	This directory is the "spool directory" for "work" that is to be processed by the Basic Networking Utilities. It contains a tree-like structure of subdirectories associated with remote machines that your 3B2 Computer wishes to communicate with (has communicated with recently). These subdirectories are also used for administrative purposes such as storing log and status information.
/usr/spool/uucppublic	This directory is the "public" directory for UUCP transfers. The public directory is used to store files that have been sent to your 3B2 Computer. Some remote machines may be restricted to placing files in this directory, while others may have permission to place files elsewhere.

The Software Programs and Their Purpose

There are several types of programs associated with the Basic Networking Utilities. Some of these programs are used by normal users to transfer data and obtain status information, while others are used for administrative purposes, or are executed internally. The following paragraphs contain a brief description of the programs and their purpose.

User Programs

cu: Connects your 3B2 Computer to a remote machine and allows you to be logged in on both machines at the same time. This allows you to transfer files or execute commands on either machine without dropping the link.

ct: Connects your 3B2 Computer to a remote terminal and allows the user of the remote terminal to log in. The user of a remote terminal may call into the 3B2 Computer and request that the 3B2 Computer call the remote terminal back. Here, the 3B2 Computer drops the initial link so that the modem will be available when it is called back.

uucp: Performs all the preliminary work in queuing file transfers to remote machines. It creates "work" files that contain the instructions for transferring the queued file(s). Depending on the options specified, it may create a copy of the file to be transferred in the spool directory. These files are called "data" files. Once the "work" and "data" files have been created, **uucp** calls the **uucico** daemon that in turn attempts to contact the remote machine.

uuto: This program works similar to the **uucp** program. It calls the **uucp** program to create "work" and "data" files. The main difference between **uuto** and **uucp** is the way the transferred files are placed on the remote machine. With **uucp**, you can specify a path name on the remote machine where you want the files to be placed. With **uuto**, all transferred files are placed in the **uucppublic** directory under **/usr/spool/uucppublic/receive**.

uupick: When files are transferred to a machine using **uuto**, **uupick** can be used to retrieve the files placed under **/usr/spool/uucppublic/receive**.

uux: This program creates "work" files, "data" files, and "execute" files for executing commands on a remote machine. The "work" file contains the same information as those created by **uucp** and **uuto**. The "execute" files contain the command string to be executed on the remote machine and a list of the "data" files. The "data" files are those files required for the command execution.

uustat: This program displays status information for requested transfers (**uucp**, **uuto**, or **uux**). It also provides you with a means of controlling queued transfers.

Administrative Programs

uulog: This program displays the contents of a specified machine's log file. Individual log files are created for each remote machine your 3B2 Computer communicates with using the **uucp**, **uuto**, and **uux** programs.

uucleanup: This program has several functions that are all associated with the cleanup of the spool directory. It is normally executed out of a shell script called **uudemon.cleanu** that is started by **cron**.

Uutry: This program is a shell script that is used to test call processing capabilities with a moderate amount of debugging. It invokes the **uucico** daemon to establish the communication link between your 3B2 Computer and the specified machine.

uucheck: This program checks for the presence of Basic Networking directories, programs, and support files. It is also capable of checking certain parts of the **Permissions** file.

Internal Programs

uugetty: This program is similar to the **getty** program, except it permits a line (port) to be used in both directions. The **uugetty** allows users to log in on your 3B2 Computer and, if the line is not in use, it will allow **uucico**, **cu**, or **ct** to use it for dialing out. If one of these programs attempts to dial out when the line is busy, **uugetty** will deny the requester permission and echo a message indicating that the device is unavailable. **Uugetty** is executed as a function of the **init** program.

The UUCP Daemons and Their Purpose

There are three daemons that are part of the Basic Networking Utilities. These daemons are routines that run as background processes to handle file transfers and command executions.

uucico: This daemon is referred to as the transport program for UUCP requests. It selects the device used for the link, establishes the link to the

remote machine, performs the required log in sequence, performs permission checks, transfers "data" and "execute" files, logs results, and notifies specified users of transfer completions via **mail**. When the local **uucico** daemon calls a remote machine, it "talks" to the **uucico** daemon on the remote machine during the session. The **uucico** daemon is executed by several methods. It is started by the **uucp**, **uuto**, and **uux** programs to contact the remote machine after all the required "data", "work", and/or "execute" files have been created. It is also started by the **uusched** and **Uutry** programs.

uuxqt: This daemon is the execution program for remote execution requests. It searches the spool directory for "execute" files (X.) that have been sent from a remote machine. When an X. file is found, **uuxqt** opens it to get the list of data files that are required for the execution. It then checks to see if the required data files are available and accessible. If the files are present and can be accessed, **uuxqt** checks the **Permissions** file to verify that it has permission to execute the requested command. The **uuxqt** daemon is executed out of the **uudemon.hour** shell script that is started by **cron**.

uusched: This daemon schedules the queued work in the spool directory. Before starting the **uucico** daemon, **uusched** randomizes the order in which remote machines will be called. **Uusched** is executed out of a shell script called **uudemon.hour** that is started by **cron**.

The Supporting Data Base Files and Their Purpose

As mentioned earlier, several of the Basic Networking programs require information contained in support files. These support files are located in the **/usr/lib/uucp** directory. The **cu**, **ct**, **uucico**, and **uuxqt** programs require supporting information from the following files:

Devices This file contains information about the location and line speed of the automatic call unit, direct links, and possibly network devices.

Dialers This file contains character strings required to negotiate with network devices (automatic calling device) in the establishment of connections to remote computers (non 801-type dialers).

Systems This file contains information needed by the **uucico** daemon (and possibly the **cu** program) to establish a link to a remote machine. It contains information such as the name of the remote machine, the name of the connecting device associated with the remote machine, when the machine can be reached, telephone number, login ID, password, etc.

Dialcodes This file contains dial-code abbreviations that may be used in the phone number field of **Systems** file entries.

Permissions This file defines the level of access that is granted to machines when they attempt to transfer files or remotely execute commands on your 3B2 Computer.

There are several other files that may be considered part of the supporting data base, but these files are not directly related to the process of establishing a link and transferring files. So, discussion of these files are reserved for Chapter 3, "ADMINISTRATION."

HOW BASIC NETWORKING OPERATES

This section briefly describes the operation of the Basic Networking Utilities. There are five programs that allow your 3B2 Computer to communicate with remote machines. The following paragraphs briefly describe what happens when you execute these programs.

ct - Connect a Terminal

The **ct** program instructs your 3B2 Computer to begin a call to a remote terminal and issue a **getty** to that remote terminal. The command line of the **ct** command must contain the telephone number of the remote terminal. Of course, the remote terminal must be attached to a modem that will automatically answer the call.

When the **ct** command line is issued, the **ct** program will search for an automatic dialer in the **Devices** file with a transfer rate that matches what was specified in the command line. If no transfer rate was specified, it defaults to 1200 bps. When **ct** finds the dialer to be used, it attempts to dial the phone number specified in the command line. If no dialer is available, **ct** asks if it should wait for an available dialer and if so, how many minutes should it wait. An option is available to override this dialogue.

When the modem at the remote terminal answers the call from your 3B2 Computer, it is issued a **getty** (login) process. At this point, the user at the remote terminal may attempt to log in.

The user at a remote terminal may call your 3B2 Computer, log in, and request that the 3B2 Computer call the remote terminal back using the **ct** command. If this scenario is used, the remote user issues a **ct** command and the link from the remote terminal is dropped. After **ct** finds an available dialer in the **Devices** file, it will call the remote terminal back.

cu - Call a UNIX System

The **cu** command enables you to call another machine and log in as a remote user. The telephone number or node name of the remote machine is required in the command line. If the phone number is specified, it is passed on to the automatic dial modem. If a system name is specified, the phone number is obtained from the associated **Systems** file entry. If an automatic dial modem is not used to establish the connection, the line (port) associated with the direct link to the remote machine can be specified in the command line.

If an automatic dial modem is used, the **cu** program will search for an automatic dialer in the **Devices** file with a transfer rate that matches what was specified in the command line. If no speed is specified, the first dialer listed (if available) is used regardless of its transfer rate. After the link is established and you have successfully completed the log in process, you will be logged in on both computers. This will allow you to execute commands on either computer and/or transfer ASCII coded files from one computer to another. After you end the connection, you will still be logged in on your 3B2 Computer (calling computer). This command can only be executed by an active computer.

uucp - UNIX-to-UNIX System Copy

The **uucp** command will allow you to transfer file(s) to a remote computer without knowing any details of the connection. All that you are required to know is the name of the remote computer and possibly the login ID of the remote user to whom the file(s) is being sent. The details of the connection are kept in the **Systems** file.

When you enter a **uucp** command, the **uucp** program creates a "work" file and possibly a "data" file for the requested transfer. The "work" file contains information required for transferring the file(s). The "data" file is simply a copy of the specified source file. After these files are created in the spool directory, the **uucico** daemon is started.

The **uucico** daemon attempts to establish a connection to the remote machine that is to receive the file(s). It first gathers the information required for establishing a link to the remote machine from the **Systems** file. This is how **uucico** knows what type of device to use in establishing the link. Then, **uucico** searches the **Devices** file looking for the devices that match the requirements listed in the **Systems** file. After **uucico** finds an available device, it attempts to establish the link and log in on the remote machine.

When **uucico** logs in on the remote machine, it starts the **uucico** daemon on the remote machine. The two **uucico** daemons then negotiate the line protocol to be used in the file transfer(s). The local **uucico** daemon then

transfers the file(s) to the remote machine and the remote **uucico** places the file in the specified path name(s) on the remote machine. After your 3B2 Computer completes the transfer(s), the remote machine may send files that are queued for your 3B2 Computer. The remote machine can be denied permission to transfer these files with an entry in the **Permissions** file. If this is done, the remote machine must establish a link to your 3B2 Computer to do the transfers.

If the remote machine or the device selected to make the connection to the remote machine is unavailable, the request remains queued in the spool directory. Each hour (default), **uudemon.hour** is started by **cron** that in turn starts the **uusched** daemon. When the **uusched** daemon starts, it searches the spool directory for the remaining "work" files, generates the random order in which these requests are to be processed, and then starts the transfer process (**uucico**) described in the previous paragraphs.

The transfer process described generally applies to an active machine. An active machine (one with calling hardware and Basic Networking software) can be set up to "poll" a passive machine. A passive machine can queue file transfers (because it has Basic Networking software) but, it cannot call the remote machine because it does not have the required hardware. The **Poll** file (**/usr/lib/uucp/Poll**) contains a list of machines that are to be polled in this manner. For additional information, refer to the discussion on the **Poll** file and **uudemon.poll** in Chapter 3, "ADMINISTRATION."

uuto - Public UNIX-to-UNIX System Copy

The **uuto** program uses the **uucp** program to build "work" files and "data" files in the spool directory for requested transfers. The difference is that the **uuto** command will not allow you to specify a path name as a destination for the file. The **uuto** command automatically puts the file in a directory under **/usr/spool/uucppublic/receive**. Once the transfer is complete, mail is sent to the appropriate user indicating that a file has arrived and was placed in the public area. That user can then use the **uupick** command to retrieve that file. The **uupick** command will search the public area for files destined to the user and allow the user to interactively delete, print, or move the file to a named directory.

uux - UNIX-to-UNIX System Execution

The **uux** command allows commands to be executed on a remote machine. It gathers files from various computers and executes the specified command on these files and sends the standard output to a file on the specified computer. This can be useful when some of the required resources (commands and/or files) are not present on your 3B2 Computer. Remote mail is implemented using the **uux** program, but its execution is embedded in the standard **mail** command. For security reasons, many machines will limit the list of commands that can be executed via **uux** to the default, receipt of mail.

When the **uux** command is issued, the **uux** program creates an "execute" (X.) file that contains the names of the files required for execution, your log in name, the destination of the standard output, and the command to be executed. **Uux** also creates "work" (C.) files that are used to gather the files required for execution. These files are then sent to the remote machine, along with the "execute" file, by the **uucico** daemon and placed in the remote spool directory. Periodically, the **uuxqt** daemon on the remote machine is started to search for X. files in the spool directory. On finding an X. file, the **uuxqt** daemon checks to see if all the required data files are available and accessible. It then checks the **Permissions** file to verify that the command(s) listed can be performed. After execution, **uuxqt** sends the standard output to a file on the specified computer.

Chapter 3

ADMINISTRATION

PAGE

Chapter 3

ADMINISTRATION

This chapter discusses the files and tasks associated with the operation of the Basic Networking Utilities. The amount of effort required to administer the Basic Networking Utilities depends on the amount of "traffic" that enters or leaves your 3B2 Computer. For an average computer, little if any intervention with the automatic cleanup functions is required. A computer with a large amount of traffic may require more attention as problems arise.

By now, you have probably realized that the "UUCP facilities" make up the bulk of the Basic Networking Utilities. The UUCP facilities could generally be defined as all the programs and support files in Basic Networking except the **ct** and **cu** programs.

Refer to the *AT&T 3B2 User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

ADMINISTRATIVE FILES

TM - temporary data file

These data files are created under the spool directory (i.e. /usr/spool/uucp/XXXX) when receiving a file from another machine. The directory "XXXX" has the same name as the remote machine that is sending the file. The temporary data file names have the format:

TM.*pid.ddd*

Where:

pid	is a process-id,
ddd	is a sequential three-digit number starting at zero.

After the entire file is received, the **TM.** file is moved to the path name specified in the command line. If the file was sent via the **uuto** program, the file is automatically moved to the public area. If processing is abnormally ended, the **TM.** file may remain in the "XXXX" directory. These files should be periodically removed.

LCK - lock file

Lock files are created in the **/usr/spool/locks** directory for each device in use. Lock files prevent duplicate conversations and multiple attempts to use the same calling device. The file name has the format

LCK.*str*

where *str* is either a device or computer name. The files may be left in the spool directory if runs abort (usually on computer crashes). The lock files will be ignored (reused) after the parent process is no longer active.

Work (C.) File

Work files are created in a spool directory when work (transfers or remote command executions) has been queued for a remote computer. Their names have the format:

C. *sysnxxxx*

where *sys* is the name of the remote computer, *n* is the ASCII character representing the grade (priority) of the work, and *xxxx* is the four-digit job sequence number assigned by UUCP.

Work files contain the following information:

- Full path name of the file to be sent or requested.
- Full path name of the destination or ~user/file name.
- User log in name.
- List of options.
- Name of associated data file in the spool directory. If the **-c** or **-p** option was specified, a dummy name (D.0) is used.
- Mode bits of the source file.
- Remote user's log in name to be notified on completion of the transfer.

Data (D.) File

Data files are created when it is specified in the command line to copy the source file to the spool directory. Their names have the following format:

D. *sysnxxxx*

where *sys* is the name of the remote computer, *n* is the character representing the grade (priority) of the work, and *xxxx* is the four-digit job sequence number assigned by UUCP. The four-digit job sequence number may be followed by a subjob number that is used when there are several **D.** files created for a work (**C.**) file.

Execute (X.) File

Execute files are created in the spool directory before remote command executions. Their names have the following format:

X. *sysnxxxx*

where *sys* is the name of the remote computer, *n* is the character representing the grade (priority) of the work, and *xxxx* is the four-digit sequence number assigned by UUCP.

Execute files contain the following information:

- Requester's log in and computer name.
- Name of file(s) required for execution.
- Input to be used as the standard input to the command string.
- Computer and file name to receive standard output from the command execution.
- Command string.
- Option lines for return status requests.

Machine Log Files

Log files are created for each remote machine with which your 3B2 Computer communicates. Each machine may have four log files, one for **uucico**, **uuxqt uux**, and/or **uucp** requests depending on the type of communication that has taken place. The log files are kept in the directory **/usr/spool/uucp/.Log**. Each day, these log files are combined and stored in the directory **/usr/spool/uucp/.Old** when **uudemon.cleanu** is executed. The combined files are kept three days before they are removed. If space is a problem, the administrator may consider reducing the number of days the files are kept by changing the **uudemon.cleanu** shell file.

SUPPORTING DATA BASE

The data base that supports the Basic Networking Utilities is composed of several support files. These support files contain information required by the **uucico** and **uuxqt** daemons during file transfers or remote command executions. All the support files are located in the **/usr/lib/uucp** directory.

Devices File

The **Devices** file (**/usr/lib/uucp/Devices**) contains the information for all the devices that may be used to establish a link to a remote machine. It contains information for both automatic call units, direct links, and network connections. Although provisions are made for several types of devices, only the AT&T Automatic Dial Modem and Direct Links are supported by AT&T. The management of this file is supported by the Simple Administration subcommand **devicemgmt**.

This file works closely with the **Dialers**, **Systems**, and **Dialcodes** files. It may be beneficial to become vaguely familiar with these files before attempting to gain an understanding of the **Devices** file.

Each entry in the **Devices** file has the following format:

Type Line Line2 Class Dialer-Token-Pairs

where each field (separated by a space) is defined in the following paragraphs.

Type: This field may contain one of four keywords:

Direct	This keyword shows a Direct Link to another computer (for **cu** connections only).
ACU	This keyword shows that the link to a remote computer is made through an automatic call unit (Automatic Dial Modem). This modem may be connected either directly to the 3B2 Computer or indirectly through a Local Area Network (LAN) switch.
Network	This keyword shows that the link is established through a LAN switch where *Network* is replaced with either **micom** or **develcon**. These two LAN switches are the only ones that contain caller scripts in the **Dialers** file (discussed a little later). Other switches may be used if caller scripts are constructed and placed in the **Dialers** file.
System-Name	This keyword shows a direct link to a particular machine where *System-Name* is replaced by the name of the particular computer. This naming scheme is used to convey that the line associated with this **Devices** entry is for a particular machine.

The keyword used in the Type field is matched against the third field of **Systems** file entries, shown below:

Devices: **ACU** contty - 1200 penril

Systems: eagle Any **ACU** 1200 3-2-5-1 ogin: nuucp ssword: Oakgrass

Line: This field contains the device name of the line (port) associated with the **Devices** entry. For instance, if the Automatic Dial Modem for a particular entry was attached to the /dev/contty line, the device name would be contty.

Line2: If the ACU keyword was used in the Type field and the ACU is an 801-type dialer, this field would contain the device name of the 801 dialer. Noted that 801 type ACUs do not contain a modem. Therefore, a separate modem is required and would be connected to a different line (defined in the Line field). This means that one line would be allocated to the modem and another to the dialer. Since the 3B2 Computer will not normally use this type of configuration, this field is ignored, but must contain a pseudo entry as a placeholder (use a "-" as a placeholder).

Class: If an ACU keyword is used, this may be just the speed of the device. It may contain a letter and speed (e.g. C1200, D1200) to differentiate between classes of dialers (Centrex or Dimension PBX). This is necessary because many larger offices may have more than one type of telephone network. One network may be dedicated to serving only internal office communications while the other handles the external communications. Therefore, it is necessary to distinguish what line(s) should be used for internal communications and what should be used for external communications. The same distinction must be made in the **Systems** file because a match is made against the fourth field of **Systems** file entries, shown below:

Devices: ACU contty - **D1200** penril

Systems: eagle Any ACU **D1200** 3-2-5-1 ogin: nuucp ssword: Oakgrass

Some devices can be used at any speed, so the keyword "Any" may be used in the Class field. If "Any" is used, the line will match any speed requested in a **Systems** entry. If this field is "Any" and the **Systems** Class field is "Any", the speed defaults to 1200 bps.

Dialer-Token-Pairs: This field contains pairs of dialers and tokens. The "dialer" portion may be an automatic dial modem, LAN switch, or "direct" for Direct Link devices. The "token" portion may be supplied immediately following the "dialer" or if not present, it is taken from the **Systems** file. This field has the format:

dialer-token dialer-token

where the last pair may or may not be present, depending on the associated device (dialer). Usually, the last pair will contain only a "dialer" and the "token" and is retrieved from the Phone field of the **Systems** entry.

The Dialer-Token-Pairs (DTP) field may be structured four different ways, depending on the device associated with the entry:

1. If a direct link is established to a particular computer, the DTP field of the associated entry would contain the keyword "direct." This is true for both types of direct link entries, Direct and System-Name (refer to discussion on the Type field).

2. If an automatic dialing modem is connected directly to a 3B2 Computer port, the DTP field of the associated **Devices** entry will only have one pair. This pair would normally be the name of the modem. This name is used to match the particular **Devices** entry with an entry in the **Dialers** file. Therefore, this "dialer" must match the first field of a **Dialers** file entry, shown below:

   Devices: ACU contty - 1200 **ventel**

   Dialers: **ventel** =&-% "" \r\p\r\c $ <K\T%%\r>\c ONLINE!

Notice that only the "dialer" (**ventel**) is present in the DTP field of the **Devices** entry. This means that the "token" to be passed on to the dialer is taken from the Phone field of a **Systems** file entry.

3. If an automatic dialing modem is connected to a LAN, the 3B2 Computer must first access the switch and the switch will make the connection to the automatic dialing modem. This type of entry would have two pairs. The "dialer" portion of each pair (fifth and seventh fields of entry) is used to match entries in the **Dialers** file, shown below:

   Devices: ACU tty14 - 1200 **develcon** vent **ventel**

   Dialers: **ventel** =&-% "" \r\p\r\c $ <K\T%%\r>\c ONLINE!
   Dialers: **develcon** "" "" \pr\ps\c est:\007 \E\D\e \007

   In the first pair, develcon is the "dialer" and vent is the "token" that is passed to the Develcon switch to tell it what device (ventel modem) to connect to the 3B2 Computer. This token would be uncommon for each LAN switch since each switch may be set up differently. Once the ventel modem has been connected, the second pair is accessed where ventel is the "dialer" and the "token" is retrieved from the **Systems** file.

4. If a machine to which you wish to communicate is on the same local network switch as your 3B2 Computer, the 3B2 Computer must first access the switch and the switch can make the connection to the other machine. In this type of entry, there is only one pair. The "dialer" portion is used to match a **Dialers** entry, shown below:

   Devices: develcon tty13 - 1200 **develcon** \D

   Dialers: **develcon** "" "" \pr\ps\c est:\007 \E\D\e \007

   This shows that the "token" is left blank that indicates that it is retrieved from the **Systems** file. The **Systems** file entry for this particular machine will contain the token in the Phone field that is normally reserved for the phone number of the machine (refer to

"Systems File", Phone field). This type of DTP contains an escape character (\D) which ensures that the contents of the Phone field will not be interpreted as a valid entry in the **Dialcodes** file.

There are two escape characters that may appear at the end of a DTP field:

\T This shows that the Phone (token) field should be translated using the **Dialcodes** file. This escape character is normally placed in the **Dialers** file for each caller script associated with an automatic dial modem (penril, ventel, etc.). Therefore, the translation will not take place until the caller script is accessed.

\D This shows that the Phone (token) field should not be translated using the **Dialcodes** file. If no escape character is specified at the end of a **Devices** entry, the \D is assumed (default). A \D is also used in the **Dialers** file with entries associated with network switches (develcon and micom).

Dialers File

The **Dialers** file (**/usr/lib/uucp/Dialers**) is used to specify the initial handshaking that must take place on a line before it can be made available for transferring data. This initial handshaking is usually a sequence of ASCII strings that are transmitted and expected, and is often used to dial a phone number using an ASCII dialer (such as the Automatic Dial Modem). Shown in the above examples, the fifth field in a **Devices** file entry is used as an index into the **Dialers** file. Here an attempt is made to match the **Devices** field with the first field of each **Dialers** entry. In addition, each odd numbered **Devices** field starting with the seventh position is used as an index into the **Dialers** file. The management of this file is not supported by Simple Administration. Therefore, changes must be made using the editors **ed** or **vi**.

If the match succeeds, the **Dialers** entry is interpreted to start the dialer negotiations. The first field matches the fifth and additional odd numbered fields in the **Devices** file. The second field is used as a translate string: the

first of each pair of characters is mapped to the second character in the pair. This is usually used to translate = and - into whatever the dialer requires for "wait for dialtone" and "pause."

The remaining fields are "expect-send" strings. Below are some character strings distributed with the Basic Networking Utilities in the **Dialers** file.

```
penril =W-P "" \d > s\p9\c )-W\p\r\ds\p9\c-) y\c : \E\TP > 9\c OK
ventel =&-% "" \r\p\r\c $ <K\T%%\r>\c ONLINE!
hayes  =,-, "" \dAT\r\c OK\r \EATDT\T\r\c CONNECT
rixon  =&-% "" \d\r\r\c $ s9\c )-W\r\ds9\c-) s\c : \T\r\c $ 9\c LINE
vadiac =K-K "" \005\p *-\005\p-*\005\p-* D\p BER? \E\T\e \r\c LINE
develcon "" "" \pr\ps\c est:\007 \E\D\e \007
micom ""    "" \s\c NAME? \D\r\c GO
direct
```

The meaning of some of the escape characters (those beginning with "\") used in the **Dialers** file are listed below:

\p pause (about ¼ to ½ second)

\d delay (about 2 seconds)

\D phone number or token without **Dialcodes** translation

\T phone number or token with **Dialcodes** translation

\K insert a BREAK

\E enable echo checking (for slow devices)

\e disable echo checking

\r carriage return

\c no new-line

\n send new-line

\nnn send octal number.

Additional escape characters that may be used are listed in the section discussing the **Systems** file.

The penril entry in the **Dialers** file is executed as follows. First, the phone number argument is translated, replacing any "=" with a (pause). The handshake given by the remainder of the line works as follows:

""	Wait for nothing.
\d	Delay for 2 seconds.
>	Wait for a ">."
s\p9\c	Send an "s", pause for ½ second, send a 9, send no terminating new-line
)-W\p\r\ds\p9\c-)	Wait for a ")." If it is not received, process the string between the "-" characters as follows. Send a "W", pause, send a carriage-return, delay, send an "s", send a "9", without a new-line, and then wait for the ")."
y\c	Send a "y."
:	Wait for a ":."
\E\TP	Enable echo checking. (From this point on, whenever a character is transmitted, it will wait for the character to be received before doing anything else.) Then, send the phone number followed by a pause character (P). The \T means take the phone number passed as an argument and apply the **Dialcodes** translation and the modem function translation specified by field number 2 of this entry.

>	Wait for a ">."
9\c	Send a "9" without a new-line.
OK	Waiting for the string "OK."

Systems File

The **Systems** file (**/usr/lib/uucp/Systems**) contains the information needed by the **uucico** daemon to establish a communication link to a remote machine. Each entry in the file represents a machine that can be called by the 3B2 Computer. In addition, Basic Networking can be configured to prevent any machine that does not appear in this file from logging in on your 3B2 Computer (refer to "**remote.unknown**"). More than one entry may be present for a particular machine. The additional entries represent alternate communication paths that will be tried in sequential order. The management of this file is supported by the Simple Administration subcommand **systemmgmt**.

Each entry in the **Systems** file has the following format:

System-Name Time Type Class Phone Login

where each field is defined in the following paragraphs.

System-name: This field contains the node name of the remote machine.

Time: This field is a string that shows the day-of-week and time-of-day when the remote machine can be called. The day portion may be a list containing some of the following:

Su Mo Tu We Th Fr Sa

Wk	for any week-day

Any for any day

Never for a passive arrangement with the remote machine. Here, the 3B2 Computer will never start a call to the remote machine. The call must be started by the remote machine. The 3B2 computer is in a passive mode with the remote machine. (See discussion of **Permissions** file.)

The time should be a range of times such as 0800-1230. If no time portion is specified, any time of day is assumed to be allowed for the call. Note that a time range that spans 0000 is permitted. For example, 0800-0600 means all times are allowed other that times between 6 a.m. and 8 a.m. An optional subfields available to specify the minimum time (in minutes) before a retry, following a failed attempt. The subfield separator is a semicolon (;). For example, Any;9 is interpreted as call any time; but, wait at least 9 minutes before retrying after a failure occurs.

Type: This field contains the device type that should be used to establish the communication link to the remote machine. The **Devices** file is searched for the device type listed and the device found is used to establish the connection (if available). The following keywords may appear in this field:

ACU This keyword shows that the link to a remote computer is made through an automatic call unit (automatic dial modem). This modem may be connected either directly to the 3B2 Computer or indirectly through a Local Area Network (LAN) switch.

Network This keyword shows that the link is established through a LAN switch where *Network* is replaced with either **micom** or **develcon**. These two switches are the only ones that contain caller scripts in the **Dialers** file (discussed a little later). Other switches may be used if caller scripts are constructed and placed in the **Dialers** file.

System-Name This keyword shows a direct link to a particular machine where *System-Name* is replaced by the name

of the particular computer (should be same as field one).

The keyword used in this field is matched against the first field of **Devices** file entries, shown below:

Systems: eagle Any **ACU** D1200 3-2-5-1 ogin: nuucp ssword: Oakgrass

Devices: **ACU** contty - D1200 penril

Class: This field is used to show the transfer speed of the device used in establishing the communication link. It may contain a letter and speed (e.g. C1200, D1200) to differentiate between classes of dialers (refer to the discussion on the "Devices File," Class field). Some devices can be used at any speed, so the keyword "Any" may be used. This field must match the Class field in the associated **Devices** entry, shown below:

Systems: eagle Any ACU **D1200** 3-2-5-1 ogin: nuucp ssword: Oakgrass

Devices: ACU contty - **D1200** penril

Phone: This field is used to provide the phone number (token) of the remote machine for automatic dialers (LAN switches). The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation must be one that is listed in the **Dialcodes** file. In this string, an equal sign (=) tells the ACU to wait for a secondary dial tone before dialing the remaining digits. A dash in the string (-) instructs the ACU to pause four seconds before dialing the next digit.

If your 3B2 Computer is connected to a LAN switch, you may access other machines that are connected to that switch. The **Systems** entries for these machines will not have a phone number in the Phone field. Instead, this field will contain the "token" that must be passed on to the switch so it will know what machine the 3B2 Computer wishes to communicate with. The associated **Devices** entry should have a \D at the end of the entry to ensure that this field is not translated using the **Dialcodes** file.

Login: This field contains the log in information given as a series of fields and subfields of the format:

expect send

where expect is the string that is received and send is the string that is sent when the expect string is received.

The expect field may be made up of subfields of the form:

expect[-send-expect]...

where the send is sent if the prior expect is not successfully read and the expect following the send is the next expected string. For example, with "login--login", UUCP will expect "login." If UUCP gets "login," it will go on to the next field. If it does not get login, it will send nothing followed by a new line, then look for login again. If no characters are initially expected from the remote machine, the characters "" (null string) should be used in the first expect field. Note that all send fields will be sent, followed by a new-line unless the send string is stopped with a \c.

There are several escape character that cause specific actions when they are a part of a string sent during the log in sequence. The following escape characters are useful in UUCP communications:

\N	Send a null character.
\b	Send a backspace character.
\c	If at the end of a string, suppress the new-line that is normally sent. (Ignored otherwise.)
\d	Delay two seconds before sending or reading more characters.

\p	Pause for about ¼ to ½ second.
\n	Send a new-line character.
\r	Send a carriage-return.
\s	Send a space character.
\t	Send a tab character.
\\	Send a \ character.
EOT	Send EOT character (actually EOT new-line is sent twice).
BREAK	Send a break character.
\ddd	Collapse the octal digits (ddd) into a single character and send that character.

Dialcodes File

The **Dialcodes** file (**/usr/lib/uucp/Dialcodes**) contains the dial-code abbreviations used in the Phone field of the **Systems** file. Each entry has the format

abb dial-seq

where abb is the abbreviation used in the **Systems** file (Phone field) and dial-seq is the dial sequence that is passed to the dialer when that particular **Systems** entry is accessed.

The entry

jt 9=847-

would be set up to work with a Phone field in the **Systems** file such as

jt7867. When the entry containing jt7867 is encountered, the sequence 9=847-7867 would be sent to the dialer.

The management of this file is not supported by Simple Administration. Therefore, changes must be made using the editors **ed** or **vi**.

Permissions File

The **Permissions** file (**/usr/lib/uucp/Permissions**) is used to specify the permissions that remote machines have with respect to login, file access, and command execution. Options are provided for restricting the ability to request files and the ability to receive files queued by the local site. In addition, an option is available to specify the commands that a remote site can execute on the local machine. The management of this file is not supported by Simple Administration. Therefore, changes must be made using the editors **ed**, **vi**.

How Entries are Structured

Each entry is a logical line with physical lines ending with a \ to show continuation. Entries are made up of "white space" delimited options. Each option is a name/value pair. These are constructed by an option name followed by a "=" and the value. Note that no white space is allowed within an option assignment.

Comment lines begin with a "#" and they occupy the entire line up to a newline character. Blank lines are ignored (even within multi-line entries).

There are two types of **Permissions** entries:

LOGNAME Specifies permissions that take effect when a remote machine logs in on (calls) your 3B2 Computer.

MACHINE Specifies permissions that take effect when your 3B2 Computer logs in on (calls) a remote machine.

LOGNAME entries will contain a LOGNAME option and MACHINE entries will contain a MACHINE option.

Considerations

The following items should be considered when using the **Permissions** file to restrict the level of access granted to remote machines:

1. All login IDs used by remote machines to log in for UUCP type communications must appear in one and only one LOGNAME entry.

2. Any site that is called whose name does not appear in a MACHINE entry, will have the following default permissions/restrictions:

   - Local send and receive requests will be executed.
   - The remote machine can send files to your 3B2 Computers **/usr/spool/uucppublic** directory.
   - The commands sent by remote machine for execution on your 3B2 Computer must be a default command, usually **rmail**.

Options

This section provides the details of each option, specifying how they are used and their default values.

REQUEST

When a remote machine calls your 3B2 Computer and requests to receive a file, this request can be granted or denied. The REQUEST option specifies whether the remote machine can request to set up file transfers from your 3B2 Computer. The string

REQUEST=yes

specifies that the remote machine can request to transfer files from your 3B2 Computer. The string

REQUEST=no

specifies that the remote machine cannot request to receive files from your 3B2 Computer. The "no" string is the default value. It will be used if the REQUEST option is not specified. The REQUEST option can appear in either a LOGNAME (remote calls you) entry or a MACHINE (you call remote) entry.

SENDFILES

When a remote machine calls your 3B2 Computer and completes its work, it may attempt to take work your 3B2 Computer has queued for it. The SENDFILES option specifies whether your 3B2 Computer can send the work queued for the remote machine. The string

SENDFILES=yes

specifies that the 3B2 Computer may send the work that is queued for the remote machine as long as it logged in as a name in the LOGNAME option. This string is mandatory if the 3B2 Computer is in a 'passive mode' with respect to the remote machine.

The string

SENDFILES=call

specifies that files queued in your 3B2 Computer will only be sent when the 3B2 Computer calls the remote machine. The call value is the default for the SENDFILE option. This option is only significant in LOGNAME entries since MACHINE entries apply when calls are made out to remote machines. If the option is used with a MACHINE entry, it will be ignored.

READ and WRITE

These options specify the various parts of the file system that **uucico** can read from or write to. The READ and WRITE options can be used with either MACHINE or LOGNAME entries.

The default for both the READ and WRITE options is the **uucppublic** directory, shown in the following strings:

READ=/usr/spool/uucppublic WRITE=/usr/spool/uucppublic

The strings

READ=/ WRITE=/

specify permission to access any file that can be accessed by a local user with "other" permissions.

The value of these entries is a colon separated list of path names. The READ option is for requesting files, and the WRITE option for depositing files. A value must be the prefix of any full path name of a file coming in or going out. To grant permission to deposit files in /usr/news as well as the public directory, the following values would be used with the WRITE option:

```
WRITE=/usr/spool/uucppublic:/usr/news
```

It should be pointed out that if the READ and WRITE options are used, all path names must be specified because the path names are not added to the default list. For instance, if the /usr/news path name was the only one specified in a WRITE option, permission to deposit files in the public directory would be denied.

NOREAD and NOWRITE

The NOREAD and NOWRITE options specify exceptions to the READ and WRITE options or defaults. The strings

```
READ=/ NOREAD=/etc WRITE=/usr/spool/uucppublic
```

would permit reading any file except those in the **/etc** directory (and its subdirectories - remember, these are prefixes) and writing only to the default **/usr/spool/uucppublic** directory. NOWRITE works in the same way as the NOREAD option. The NOREAD and NOWRITE can be used in both LOGNAME and MACHINE entries.

CALLBACK

The CALLBACK option is used in LOGNAME entries to specify that no transaction will take place until the calling system is called back. The string

```
CALLBACK=yes
```

specifies that your 3B2 Computer must call the remote machine back before any file transfers will take place.

The default for the COMMAND option is

```
CALLBACK=no
```

The CALLBACK option is rarely used. Note that if two sites have this option set for each other, a conversation will never get started.

COMMANDS

Warning: The COMMANDS option can be hazardous to the security of your system. Use it with extreme care.

The **uux** program will generate remote execution requests and queue them to be transferred to the remote machine. Files and a command are sent to the target machine for remote execution. The COMMANDS option can be used in MACHINE entries to specify the commands that a remote machine can execute on your 3B2 Computer. The string

```
COMMANDS=rmail
```

tells the default commands that a remote machine can execute on your 3B2 Computer. If a command string is used in a MACHINE entry, the default commands are overridden. For instance, the entry

```
MACHINE=owl:raven:hawk:dove \
COMMANDS=rmail:rnews:lp
```

overrides the COMMAND default such that the command list for machines owl, raven, hawk, and dove now consists of **rmail**, **rnews** and **lp**.

In addition to the names as specified above, there can be full path names of commands. For example,

```
COMMANDS=rmail:/usr/lbin/rnews:/usr/local/lp
```

specifies that command **rmail** uses the default path. The default paths for the 3B2 Computer are **/bin**, **/usr/bin**, and **/usr/lbin**. When the remote machine specifies **rnews** or **/usr/lbin/rnews** for the command to be executed, **/usr/lbin/rnews** will be executed regardless of the default path. Likewise, **/usr/local/lp** is the **lp** command that will be executed.

Including the "ALL" value in the list means that any command from the remote machine(s) specified in the entry will be executed. If you use this value, you give the remote machine full access to your 3B2 Computer.

The string

COMMANDS=/usr/lbin/rnews:ALL:/usr/local/lp

illustrates two points. The ALL value can appear anywhere in the string. And, the path names specified for rnews and lp will be used (instead of the default) if the requested command does not contain the full path names for rnews or lp.

The VALIDATE option should be used with the COMMANDS option whenever potentially dangerous commands like **cat** and **uucp** are specified with the COMMANDS option. Any command that reads or writes files is potentially dangerous to local security when executed by the UUCP remote execution daemon (**uuxqt**).

VALIDATE

The VALIDATE option is used with the COMMANDS option when specifying potentially dangerous commands. It is used to provide a certain degree of verification of the caller's identity. The use of the VALIDATE option requires that privileged machines have a different login/password for UUCP transactions. An important aspect of this validation is that the login/password associated with this entry be protected. If an outsider gets that information, that particular VALIDATE option can no longer be considered secure.

Much consideration should be given to providing a remote machine with a privileged login and password for UUCP transactions. Giving a remote machine a special login and password with file access and remote execution capability is like giving anyone on that machine a normal login and password on your 3B2 Computer. Therefore, if you cannot trust someone on the remote machine, do not provide that machine with a privileged login and password.

The LOGNAME entry

LOGNAME=uucpfriend VALIDATE=eagle:owl:hawk

specifies that if a remote machine that claims to be eagle, owl, or hawk logs in on your 3B2 Computer, it must have used the login uucpfriend. As can be seen, if an outsider gets the uucpfriend login/password, masquerading is trivial.

But what does this have to do with the COMMANDS option, which only appears in MACHINE entries? It links the MACHINE entry (and COMMANDS option) with a LOGNAME entry associated with a privileged login. This link is needed because the execution daemon is not running while the remote machine is logged in. It is an asynchronous process with no knowledge of what machine sent the execution request. Therefore, the real question is how does your 3B2 Computer know where the execution files came from?

Each remote machine has its own "spool" directory on your 3B2 Computer. These spool directories have write permission given only to the UUCP programs. The execution files from the remote machine are put in its spool directory after being transferred to your 3B2 Computer. When the **uuxqt** daemon runs, it can use the spool directory name to find the MACHINE entry in the **Permissions** file and get the COMMANDS list, or if the machine name does not appear in the **Permissions** file, the default list will be used.

The following example shows the relationship between the MACHINE and LOGNAME entries:

```
MACHINE=eagle:owl:hawk REQUEST=yes \
COMMANDS=ALL \
READ=/  WRITE=/

LOGNAME=uucpz VALIDATE=eagle:owl:hawk \
REQUEST=yes SENDFILES=yes \
READ=/  WRITE=/
```

These entries provide unlimited read, write, and command execution for the remote machines eagle, owl, and hawk. The ALL value in the COMMANDS option means that any command can be executed by either of these machines. Using the ALL value gives the remote machine unlimited access to your 3B2 Computer. Files that are only readable or writable by user "uucp" (like **Systems** or **Devices**) can be accessed using commands like **ed**. This means a user on one privileged machine can write in the **Systems** file as well as read it!

In the first entry, you must make the assumption that when you want to call a machine listed, you are really calling either eagle, owl, or hawk. Therefore, any files put into a eagle, owl, or hawk spool directory is put there by one of those machines. If a remote machine logs in and says that it is one of these three machines, its execution files will also be put in the privileged spool directory. You therefore have to validate that the machine has the privileged login "uucpz."

MACHINE Entry for "Other" Systems

You may want to specify different option values for the machines your 3B2 Computer calls that are not mentioned in specific MACHINE entries. This may occur when there are many machines calling in, and the command set changes from time to time. The name "OTHER" for the machine name is used for this entry, shown below:

```
MACHINE=OTHER \
COMMANDS=rmail:rnews:/usr/lbin/Photo:/usr/lbin/xp
```

All other options available for the MACHINE entry may also be set for the machines that are not mentioned in other MACHINE entries.

Combining MACHINE and LOGNAME Entries

It is possible to combine MACHINE and LOGNAME entries into a single entry where the common options are the same. For example, the two entries

```
MACHINE=eagle:owl:hawk REQUEST=yes \
 READ=/  WRITE=/

LOGNAME=uucpz REQUEST=yes SENDFILES=yes \
 READ=/  WRITE=/
```

share the same REQUEST, READ, AND WRITE options. These two entries can be merged into one entry as shown.

```
MACHINE=eagle:owl:hawk REQUEST=yes \
LOGNAME=uucpz SENDFILES=yes \
 READ=/  WRITE=/
```

Sample Permissions Files

Example 1

This first example represents the most restrictive access to your computer.

```
LOGNAME=nuucp
```

It states that login "nuucp" has all the default permissions/restrictions:

- The remote machine can only send files to **uucppublic**.
- The remote machine cannot request to receive files (REQUEST option).
- No files that are queued for the remote machine will be transferred during the current session (SENDFILES option).
- The only commands that can be executed are the defaults.

This entry alone is enough to start communications with remote machines, permitting files to be transferred only to the **/usr/spool/uucppublic** directory.

Example 2

The next example is for remote machines that log in, but have fewer restrictions. The login and password corresponding to this entry should not be distributed to the general public; it is usually reserved for closely coupled systems where the **Systems** file information can be tightly controlled.

```
LOGNAME=uucpz REQUEST=yes SENDFILES=yes \
 READ=/  WRITE=/
```

This entry places the following permissions/restrictions on a machine that logs in as "uucpz":

- Files can be requested from your 3B2 Computer (REQUEST option).
- Files can be transferred to any directory or any file that is writable by user "other." That is a file/directory that is writable by a local user with neither owner nor group permissions (WRITE option).
- Any files readable by user "other" can be requested (READ option).
- Any requests queued for the remote machine will be executed during the current session. These are files destined for the machine that has called in (SENDFILES option).
- The commands sent for execution on the local machine must be in the default set.

Example 3

The two previous examples showed entries that referred to remote machines when they log into your 3B2 Computer. This example is an entry used when calling remote machines.

```
MACHINE=eagle:owl:hawk:raven \
  REQUEST=yes READ=/  WRITE=/
```

When calling any of the systems given in the MACHINE list, the following permissions prevail:

- The remote machine can both request and send files (REQUEST option).
- The source or destination of the files on the local machine can be anywhere in the file system.
- The only commands that will be executed for the remote machine are those in the default set.

Any site that is called that does not have its name in a machine entry will have the default permissions as stated in Example 1, with the exception that files queued for that machine will be sent (the SENDFILES option is only interpreted in the LOGNAME entry).

Poll File

The **Poll** file (**/usr/lib/uucp/Poll**) contains information for polling specified Machines. Each entry in the **Poll** file contains the name of the remote machine to call, followed by a TAB character, and finally the hours the machine should be called. The entry

eagle 0 4 8 12 16 20

will provide polling of machine eagle every four hours.

> ***Note:*** It should be understood that **uudemon.poll** does not actually do the poll, it merely sets up a polling work (C.) file in the spool directory that will be seen by the scheduler, started by **uudemon.hour**. Refer to the discussion on **uudemon.poll**.

Maxuuxqts File

The **Maxuuxqts** (**/usr/lib/uucp/Maxuuxqts**) file contains an ASCII number to limit the number of simultaneous **uuxqt** programs running. This file is delivered with a default entry of one. This may be changed to meet local needs. If there is much traffic from **mail**, it may be advisable to increase this number to reduce wait time. However, keep in mind that the load on the system increases with the number of **uuxqt** programs running.

Maxuuscheds File

The **Maxuuscheds** (**/usr/lib/uucp/Maxuuscheds**) file contains an ASCII number to limit the number of simultaneous **uusched** programs running. Each **uusched** running will have one **uucico** associated with it; limiting the number will directly affect the load on the system. The limit should be less than the number of outgoing lines used by UUCP (a smaller number is often desirable). This file is delivered with a default entry of one. Again, this may be changed to meet the needs of the local system. However, keep in mind that the load on the system increases with the number of **uusched** programs running.

remote.unknown

The **remote.unknown** program (**/usr/lib/uucp/remote.unknown**) is a shell file that is executed when a remote site that is not in the **Systems** file calls in to start a conversation. The shell script will append the name and time information to the file **/usr/spool/uucp/.Admin/Foreign**. Since it is a shell, it can be easily modified. For example, it can be set up to send mail to the administrator. The contents of this file, as delivered, is as follows:

```
FOREIGN=/usr/spool/uucp/.Admin/Foreign
echo "`date`: call from system $1" >>$FOREIGN
```

If you want to permit unknown machines to converse, you may change the mode of the **remote.unknown** file to unexecutable (444).

ADMINISTRATIVE TASKS

There is a minimum amount of maintenance that must be applied to your 3B2 Computer to keep the files updated, to ensure that the network is running properly, and to track down line problems. When more than one remote machine is involved, the job becomes more difficult because there are more files to update and because users are much less patient when failures occur between machines that are under local control. The **uustat** program provides you with information about the latest attempts to contact various machines and the age and number of jobs in the queue for remote machines. The following sections describe the routine administrative tasks that must be performed by someone acting as the UUCP administrator or are automatically performed by the UUCP daemons (demons).

The biggest problem in a dialup network like UUCP is dealing with the backlog of jobs that cannot be transmitted to other machines. The following cleanup activities should be routinely performed.

Cleanup of Undeliverable Jobs

The **uustat** program should be invoked regularly to provide information about the status of connections to various machines and the size and age of the queued requests. The **uudemon.admin** shell should be started by **cron** at least once per day. This will send the administrator the current status. Of particular interest are the age (in days) of the oldest request in each queue, the number of times a failure has occurred when attempting to reach that machine, and the reason for failure. In addition, the age of the oldest execution request (X. file) is also given.

The **uudemon.cleanu** shell file is set up to remove any jobs that have been queued for several days and cannot be sent. Leftover data (D.) and work (C.) files are removed after seven days, and execute (X.) files are removed after two days. It also provides feedback to the user indicating when jobs are not being done and when these jobs are being deleted.

Cleanup of the Public Area

To keep the local file system from overflowing when files are sent to the public area, the **uudemon.cleanu** procedure is set up with a **find** command to remove any files that are older than seven days and directories that are empty. This interval may need to be shortened by changing the **uudemon.cleanu** shell file if there is not enough space to devote to the public area.

Since the spool directory is dynamic, it may grow large before transfers take place. Therefore, it is a good idea to reorganize its structure. The best way to do this on your 3B2 Computer is to put some code in the **/etc/rc.d** file to execute whenever the system is booted.

The following is an example of such code:

```
#       Clean up /usr/spool/uucp
#       Most cleanup is now done by uudemon.cleanu
#       so just copy out and back.
#
echo " UUCP SPOOL DIRECTORIES CLEANUP STARTED"
#
cd /usr/spool/uucp
mkdir ../nuucp
chown uucp ../nuucp
chgrp uucp ../nuucp
find . -print|cpio -pdml ../nuucp
cd ..
mv uucp ouucp
mv nuucp uucp
rm -rf ouucp
#
chown uucp /dev/cu[al]*
chgrp uucp /dev/cu[al]*
chmod 0666 /dev/cul*
chmod 0222 /dev/cua*
echo " UUCP SPOOL DIRECTORIES CLEANUP FINISHED"
```

Compaction of Log Files

This version of Basic Networking has individual log files for each machine and each program (e.g. machine eagle has a logfile for **uucico** requests and a logfile for **uuxqt** execution requests). The **uulog** program gives the user access to the information in these files by machine name. These files are combined and stored in directory **/usr/lib/uucp/.Old** whenever **uudemon.cleanu** is executed. This shell script saves files that are two days old. The two days can be easily changed by changing the appropriate line in the **uudemon.cleanu** shell. If space is a problem, the administrator might consider reducing the number of days the files are kept.

Cleanup of sulog and cron log

The **/usr/adm/sulog** and **/usr/lib/cron/log** files are both indirectly related to UUCP transactions. The **sulog** file contains a history of the **su** command usage. Since the uudemon entries in the **/usr/lib/cron/root** file each use the **su** command, the **sulog** could become large over a period of time. The **sulog** should be periodically purged to keep the file at a reasonable size.

Similarly, a history of all processes spawned by **/etc/cron** are recorded in **/usr/lib/cron/log**. The cron **log** file will also become large over a period of time and should be periodically purged to limit its size. The *AT&T 3B2 System Administration Utilities Guide* contains information on the purging of these files.

UUCP AND CRON

The **cron** daemon is a tool that proves to be useful in the administration of UNIX Systems. When the 3B2 Computer is in run state 2 (multi-user), **cron** scans the **/usr/spool/cron/crontab/root** file every minute for entries that contain "work" scheduled to be executed at that time. It is recommended that the UUCP administrator make use of **cron** to aid in the administration of the Basic Networking Utilities.

As delivered, the Basic Networking Utilities contain four entries in the **root** crontab file. Each one of these entries execute shell scripts that are used for various administrative purposes. These shell scripts can be easily modified to meet the needs of your system.

uudemon.admin

The **uudemon.admin** shell script mails status information to the UUCP administrative login (**uucp**) using **uustat** commands with the -p and -q options.

The **uudemon.admin** shell script should be executed daily by an entry in the root crontab file. The default **root** crontab entry for **uudemon.admin** is as follows:

```
48 8,12,16 * * * /bin/su uucp -c " /usr/lib/uucp/uudemon.admin" > /dev/null
```

uudemon.cleanu

The **uudemon.cleanu** shell script cleans up the Basic Networking log files and directories. Archived log files are updated so that no log information over three days old is kept. Log files for individual machines are taken from the **/usr/spool/uucp/.Log** directory, merged, and placed in the **/usr/spool/uucp/.Old** directory along with the older log information. Files and directories that are no longer needed in the spool directories are removed. After clean up is performed, the UUCP administrative login (**uucp**) is mailed a summary of the status information gathered during the current day.

The **uudemon.cleanu** shell script should be executed by an entry in the **root** crontab file. It can be run daily, weekly, or whenever, depending on the amount of UUCP traffic that enters and leaves your 3B2 Computer. The default root crontab entry for **uudemon.cleanu** is as follows:

```
45 23 * * * ulimit 5000; /bin/su uucp -c " /usr/lib/uucp/uudemon.cleanu" > /dev/null 2>&1
```

If log files get large, the ulimit may need to be increased.

uudemon.hour

The **uudemon.hour** shell script is used to call UUCP programs on an hourly basis. The **uusched** program is called to search the spool directory for work files (C.) that have not been processed and schedule these files for transfer to a remote machine. The **uuxqt** daemon is called to search the spool directory for execute files (X.) that have been transferred to your 3B2 Computer and were not processed at the time they were transferred.

The **uudemon.hour** shell script should be executed by an entry in the root crontab file. If the amount of traffic leaving and entering your 3B2 Computer is large, it may be started once or twice an hour. If it is small, it may be started once every four hours or so. The default root crontab entry for **uudemon.hour** is as follows:

```
41,11 * * * * " /usr/lib/uucp/uudemon.hour > /dev/null
```

uudemon.poll

The **uudemon.poll** shell script is used to poll the remote machines listed in the **Poll** file (**/usr/lib/uucp/Poll**). It creates work files (C.) for machines according to the entries listed in the **Poll** file. It should be set up to run once an hour just before **uudemon.hour** so that the work files will be present when **uudemon.hour** is called.

The **uudemon.poll** script should be executed by an entry in the root crontab file. The exact times it runs should be dependent on the scheduling of **uudemon.hour**. The default root crontab entry for **uudemon.poll** is as follows:

```
1,30 * * * * " /usr/lib/uucp/uudemon.poll > /dev/null
```

Note how **uudemon.poll** is scheduled to run eleven minutes before **uudemon.hour** runs.

INITTAB ENTRIES

The **/etc/inittab** file contains information for the processes to be spawned on the 3B2 Computer devices, including the ports. This file should normally be managed by the uucpmgmt Simple Administration subcommand **portmgmt**. Ports that are used by Basic Networking are normally bidirectional ports. Bidirectional ports can be used to receive incoming calls, as well as place outgoing calls. The **uugetty** program is used in place of **getty** for those bidirectional ports associated with Basic Networking. For additional information on the **inittab** entries associated with Basic Networking, refer to the Simple Administration subcommand **portmgmt**.

UUCP LOGINS AND PASSWORDS

There are two login IDs associated with the Basic Networking Utilities: one is the UUCP administrative login **uucp**, and the other is an access login (**nuucp**) used by remote computers to access your 3B2 Computer. These log ins should not be changed from their default settings of **uucp** and **nuucp**.

The **uucp** administrative login is the owner of all the UUCP object and spooled data files. The following is a sample entry in the **/etc/passwd** file for the administrative login:

```
uucp:zAvLCKp:5:1:UUCP.Admin:/usr/lib/uucp:
```

The **nuucp** access log in allows remote machines to log in on your 3B2 Computer. The following is a sample entry in the /etc/passwd file for the access login:

```
nuucp:zaaAA:6:1:UUCP.Admin:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

Note that the standard shell is not given to the **nuucp** login. The shell that **nuucp** receives is the **uucico** daemon that controls the conversation when a remote machine logs into your machine.

The assigning of passwords for the **uucp** and **nuucp** logins is left up to the administrator. The passwords should be at least six to eight characters with a rich alphabet. Only the first eight characters of the passwords are significant. If the password for the access log in is changed for security reasons, make certain that the remote machines that are a part of your network are properly notified of the change.

Chapter 4

SIMPLE ADMINISTRATION

PAGE

Chapter 4

SIMPLE ADMINISTRATION

INTRODUCTION

The Basic Networking Utilities Management (uucpmgmt) package is a set of Simple Administration scripts used to maintain certain support files of the Basic Networking Utilities. The support files provide information that enables your 3B2 Computer to contact remote machines for sending and receiving mail and files, as well as executing commands on remote machines. There are four **uucpmgmt** subcommands that add, delete, list, and change the contents of certain Basic Networking Utilities support files.

The support files managed by **uucpmgmt** are listed below:

- **Systems** (/usr/lib/uucp/Systems): This file contains the names and calling information for the communication program, permitting it to call and converse with remote machines. This file is managed using the **systemmgmt** subcommand.

- **Devices** (/usr/lib/uucp/Devices): This file contains a list of the devices that may be used to call remote machines. Device type, speed, port name (such as /dev/contty) are some of the information contained in the **Devices** file. This file is managed using the **devicemgmt** subcommand.

- **inittab** (etc/inittab): Information in this file controls the direction of traffic on the ports used by the Basic Networking Utilities. Ports can be appointed as incoming, outgoing, or bidirectional. This file is managed using the **portmgmt** subcommand.

- **Poll** (/usr/lib/uucp/Poll): This file contains a list of the machines to be polled by your 3B2 Computer, and the times they are to be polled. This file is managed using the **pollmgmt** subcommand.

FUNCTIONALITY

The **uucpmgmt** menu is located under the **packagemgmt** menu of simple administration (sysadm). In order for **uucpmgmt** to appear in the **packagemgmt** menu, the Basic Networking Utilities must first be loaded from the floppy disk. For each subcommand, you can select the operation to be performed by entering either the associated number or the name of the subcommand.

The subcommands are organized as a series of questions and answers. Answers take the form of either an entry from a list, a y/n response, and a name or a number depending on the question. At any point, the user can type a "q" to quit the operation. When shown as a possible default choice, entering a carriage return (<*CR*>) selects the default entry. If the selection list includes a "?", entering a "?" outputs the associated help message and then returns the user to the question.

Common Functionality

The following paragraphs discuss the common functions of each **uucpmgmt** subcommand.

Required Files Test: Common to the list, delete, and modify operations is a test for the existence of the file(s). If that file does not exist, the user is advised about the appropriate operations to be performed. After the advice message has been displayed, the operation is ended, returning the user to the **uucpmgmt** menu.

List Operation: On entry to the list operation, a summary list of all the names known to the Basic Networking Utilities is presented. For the **systemmgmt** subcommand, the summary information is the list of systems in the **Systems** file. For **devicemgmt**, it is the list of ports; for **pollmgmt**, the systems being polled; and for **portmgmt**, the list of ports listed in the **Devices** file. You are then asked to enter the name you want to see in detail. Only names that appear on the summary list are permissible responses. When a valid name is entered, the line(s) for that name is presented. You are then asked if you would like to see another entry. If an "n" response is given, the operation is exited, returning you to the **uucpmgmt** menu. A "y" response again presents the summary list and prompts you to input the appropriate name for the detailed output.

Delete Operation: For the delete operation, you are presented the current list of available names from which deletions can be made. After you are asked to specify what name should be deleted, the line(s) corresponding to the name is displayed and you are then prompted whether you want to delete the entry(ies). An "y" response removes the line(s), a "n" response does not. This is followed by a question, asking if you would like to delete another entry. A "y" response again displays the current list of names from which deletions can be made. An "n" response returns you to the **uucpmgmt** menu.

Add Operation: The add operation prompts you for data required to make the individual entries. After all the data has been input, the generated line(s) is presented, followed by a request asking if you want to add the entry. A "y" response adds the line(s) to the file. An "n" response takes you back to the **uucpmgmt** menu.

Modify Operation: The only modify operation is in the **portmgmt** subcommand. It is later described in the **portmgmt** section.

SUBCOMMANDS

This section contains a detailed description of each of the four subcommands; **systemmgmt**, **devicemgmt**, **portmgmt**, and **pollmgmt**.

systemmgmt

This subcommand manages the **Systems** file (/usr/lib/uucp/Systems). It permits four operations; add, delete, call, and list.

add: For the add operation, the **Systems** file is checked for read and write permission; failure causes the operation to end with a message. After a brief introduction to the subcommand, you are prompted for the fields needed to make entries in the file. The contents of these fields are listed below:

- Node name of system you want to call (e.g. eagle)
- Type of device used to originate call (e.g. acu)
- Speed at which call is placed (e.g. 1200)
- Phone number of machine or token used to access connection device through a switch (e.g. 9=847-7867)
- Type of equipment you are dialing into (e.g. dialup)

- Login ID at remote machine (e.g. nuucp)
- Password entry at remote machine. (e.g. Oakgrass).

After the above information has been supplied, an entry for the **Systems** file is constructed and the contents of the entry are displayed on the screen for verification. You are then prompted to enter yes or no to add the entry to the **Systems** file. After y/n response is received, you are asked if another entry is to be made to the **Systems** file. Again, an "n" response returns you to the **uucpmgmt** menu and a "y" response takes you back to the beginning of the add option.

delete: As described in the "Common Functionality" section.

list: As described in the "Common Functionality" section.

call: This option is used to call a remote machine that has been properly entered in the **Systems** file. It is a test to ensure that all necessary entries have been entered into the **Systems** file correctly.

devicemgmt

This subcommand manages the **Devices** file (/usr/lib/uucp/Devices). It permits three operations; add, delete, and list.

add: The **Devices** file is checked for read and write permission. Failure returns you to the **uucpmgmt** menu. You are prompted for the name of a device (port) used to make the call (e.g. contty). The name entered is checked against the current **Devices** file and if it exists, you are asked if other entries are to be added. For each entry, you are prompted for the following additional data:

- Device type (e.g. penril, ventel, develcon, micom)
- Speed (e.g. 1200).

Note: If one modem is selected, the procedure will automatically generate two entries, one at 1200 bps and one at 300 bps without prompting for speed.

After all the requested data has been provided, one or more entries are created for the **Devices** file and the contents of the entries are displayed on the screen for verification. You are then prompted to enter yes or no to add the entry to the **Devices** file. After a y/n response is received, you are asked if another entry is to be made to the **Devices** file. Again, an "n" response returns you to the **uucpmgmt** menu and a "y" response takes you back to the beginning of the add operation.

delete: As described in the "Common Functionality" section.

list: As described in the "Common Functionality" section.

pollmgmt

This subcommand manages the **Poll** file (/usr/lib/uucp/Poll). It permits three operations: add, delete, and list.

add: You are prompted to enter the name of the system to be polled. If the system name already exists in the **Poll** file, a message advising that no duplicates are permitted is output. You have the opportunity to add another system or to end the operation. If the system name does not appear in the **Systems** file, you are warned and asked whether to continue. If a "y" response is received, the process continues. If an "n" response is receive, you are asked about making other poll entries.

If a potential system name is entered, you are asked for a space separated list of hours for polling. Each hour entered must be an integer in the range of 0 through 23. Invalid responses will result in a message describing the list of hours. The default value for this question causes the specified system to be polled every hour.

On successful input of the hours list, the **Poll** entry (consisting of the system name followed by the polling hours) is displayed on the screen for verification. You are then prompted to enter yes or no to add the entry to the **Poll** file. After a y/n response is received, you are asked if another entry is to be made to the **Poll** file. Again, an "n" response returns you to the **uucpmgmt** menu and a "y" a response takes you back to the beginning of the add option.

delete: As described in the "Common Functionality" section.

list: As described in the "Common Functionality" section.

portmgmt

This subcommand manages the **/etc/inittab** file. It permits two operations, modify and list. The only ports from inittab that are accessible are those that appear in the **Devices** file.

modify: For a valid port, the direction of the port is printed. You are prompted for the desired direction (incoming, outgoing, or bidirectional) and speed (default is current speed). This data is used to change the **inittab** entry and the modified entry is displayed on the screen for verification. You are prompted to tell whether the modified entry is correct. After a y/n response is received, you are asked if another inittab entry is to be modified. Again, an "n" response returns you to the **uucpmgmt** menu and a "y" response takes you back to the beginning of the modify option.

If the direction is set to bidirectional, this sets up the **inittab** entry up to respawn **uugetty** on the port. If incoming is specified, the entry is set up to respawn **getty** on the port, and for outgoing, respawn is turned off.

list: As described in the "Common Functionality" section.

Chapter 5

DIRECT LINKS

GENERAL

This chapter discusses how to directly link

- Two 3B2 Computers
- A 3B2 Computer to a 3B5 Computer
- A 3B2 Computer to a 3B20 Computer.

Direct links would be beneficial only when:

- It is not possible to link the machines together through a Local Area Network (LAN).
- The two machines transfer large amounts of data on a regular basis.
- The two machines are located no more than several hundred cable feet apart.

The amount of cable used to link two machines is dependent on the environment in which the cable is run. The standard for RS-232 connections is 50 feet or less with transmission rates as high as 19200 bits per second (bps). As the cable length is increased, noise on the lines may become a problem. If noise becomes a problem the transmission rate must be decreased or limited distance modems be placed on each end of the line. Normally, you should not use more than 1000 cable feet to connect the two machines. This link should operate comfortably at 9600 bps in a clean (noise free) environment.

The configuration used to hardwire two 3B Computers together uses two 8-wire cables (available in lengths of 7-, 14-, 25-, and 50-feet) and two RS-232 connectors. The ordering information is provided in Figure 5-1.

DESCRIPTION	COMCODE NUMBER
7-Foot Shielded Cable	403-60-09-68
14-Foot Shielded Cable	403-60-09-76
25-Foot Shielded Cable	403-60-09-84
50-Foot Shielded Cable	403-60-09-92
ACU/Modem Connector	232-21-25-005
Terminal/Printer Connector	232-22-25-006

Figure 5-1. Part Numbers for Hardware Used in Directs Links

If the link is established using the parts listed in Figure 5-1, the machines could not be separated by more than 100 cable-feet (two 50-foot cables connected together) because the longest cable available is 50 feet.

If the two machines are separated by more than 100 cable-feet, a null-modem cable must be constructed as follows:

Pin 1 to 1
Pin 2 to 3
Pin 3 to 2
Strap pin 4 to 5 in the same plug
Pin 6 to 20
Pin 7 to 7
Pin 8 to 20
Pin 20 to 6
Pin 20 to 8.

HOW THE DIRECT LINK IS CONNECTED

3B2 Computer to 3B2 Computer Direct Link

Using the parts listed in Figure 5-1, the establishment of a direct link between two 3B2 Computers is simple. The following steps will guide you in establishing a direct link between two 3B2 Computers (also, refer to Figure 5-2).

1. Connect one end of the first shielded cable to the selected port on your 3B2 Computer. Be sure to attach the ground connector.

2. Connect the other end of the first shielded cable to the ACU/Modem Adapter. (If it is desired to connect the two computers over a distance greater than 100 feet, substitute the ACU/Modem Adapter with a Terminal/Printer Adapter attached to a null modem cable of the appropriate length.)

3. Connect the Terminal/Printer Adapter to the ACU/Modem Adapter.

4. Connect the second shielded cable to the Terminal/Printer Adapter.

5. Connect the other end of the second shielded cable to the appropriate port on the remote 3B2 Computer.

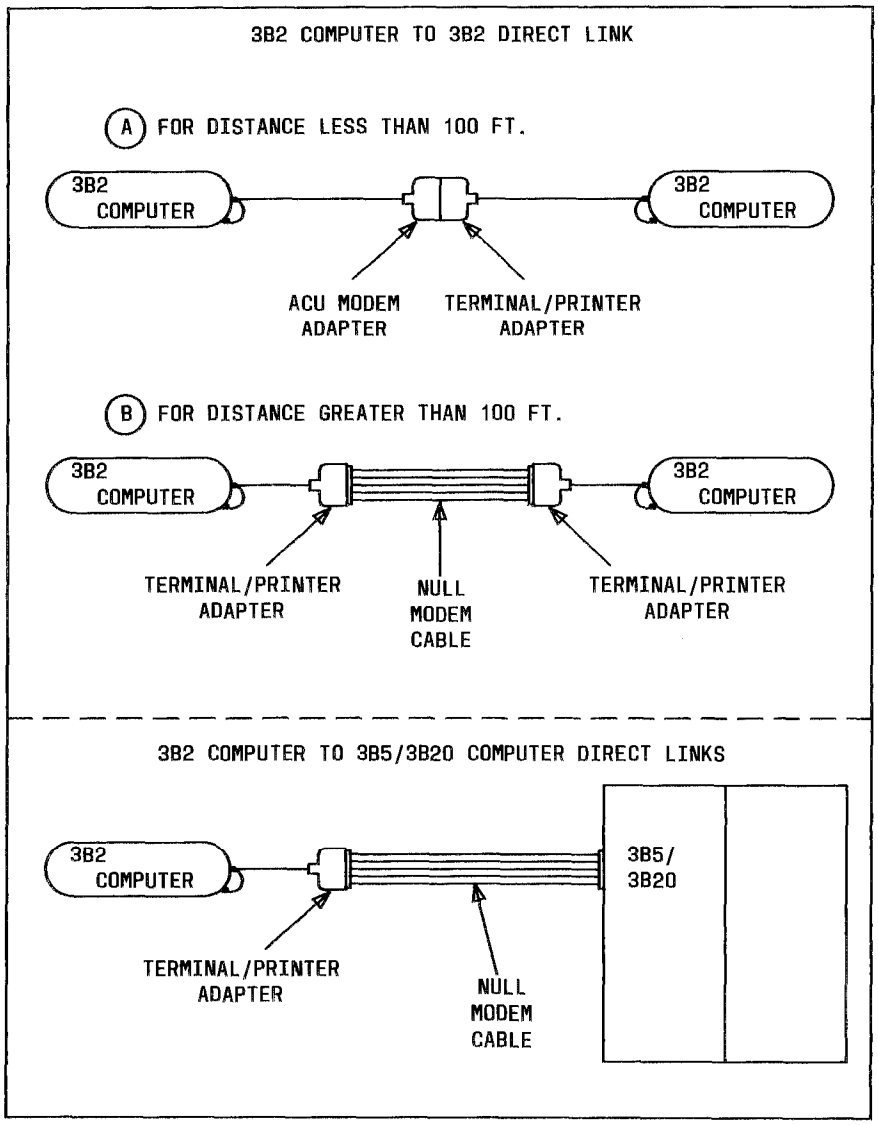


Figure 5-2. Examples of Direct Links

3B2 Computer to 3B5 Computer or 3B2 Computer to 3B20 Computer Direct Link

Establishing a direct link between a 3B2 Computer and a 3B5 Computer or between a 3B2 Computer and a 3B20 Computer is simple. The following steps will guide you in establishing a direct link between a 3B2 Computer and a 3B5 Computer or between a 3B2 and a 3B20 Computer (also, refer to Figure 5-2).

Note: The same parts are used for either link.

1. Connect one end of a shielded cable to the selected port on your 3B2 Computer. Be sure to attach the ground connector.

2. Connect the other end of the shielded cable to a Terminal/Printer Adapter.

3. Connect a null modem cable to the appropriate port on the remote 3B5 Computer or 3B20 Computer.

BASIC NETWORKING SOFTWARE AND DIRECT LINKS

Ideally, systems that have a direct link should run common and current releases of the UNIX System to have the full set of capabilities available. (Bidirectional ports that is supported by the **uugetty** program was introduced with UNIX System V Release 2.0 Version 1.) However, lack of commonality does not prevent use of the Basis Networking feature. This section describes the software files that must be modified on your 3B2 Computer to accommodate a direct link connection. You may want to consult the documentation provided with your machine if you're linking directly to a remote machine other than a 3B2 Computer.

The following support files must be updated to reflect the presence of a Direct Link:

- **/usr/lib/uucp/Devices**
- **/etc/inittab**
- **/usr/lib/uucp/Systems.**

All additions/modifications to the above files can be done by using the **uucpmgmt** Simple Administration subcommands.

Making Entries into the Devices File

The **Devices** file contains the information about the location (line) and transmission rate of the link. Entries can be added to the **Devices** file using the uucpmgmt **devicemgmt** subcommand. The operation to be performed under **devicemgmt** is the add operation. The add operation prompts you for the following information:

- Port name (for /dev/tty21 - **tty21**)
- Device type to call on (**direct**)
- Speed at which you want to call (**9600** or **19200**).

To access the **devicemgmt** subcommand and add entries in the **Devices** file, enter:

```
# sysadm devicemgmt<CR>

Running subcommand 'devicemgmt' from menu 'uucpmgmt',
BASIC NETWORKING UTILITIES MANAGEMENT

Note: After a brief introduction to the devicemgmt
subcommand, the procedure interactively prompts you for
the information listed above.
```

Making Changes to the /etc/inittab File

The differences between the two versions of Basic Networking (UUCP) are reflected in the **/etc/inittab** file. The newest version allows for bidirectional log in capability, as well as communication by respawning **uugetty** instead of **getty**. This means that if two machines (both using **uugetty**) were connected via a direct link, either of these machines could request communication with the other. This would not be true if only one machine was capable of respawning **uugetty**.

If the direct link is connecting your 3B2 Computer with a machine that has the new version of Basic Networking, the **/etc/inittab** files on both machines should be set up to allow "bidirectional" traffic on the associated lines. This means that the lines used must respawn **uugetty** on each end of the link. This would allow either machine to request communication with (call) the other.

If the direct link is connecting your 3B2 Computer with a machine that does not have the new version of Basic Networking, the **/etc/inittab** file would be set up differently on each system. The **inittab** file on each machine would be set up to allow either "incoming" or "outgoing" traffic on its line. If one machine allows incoming traffic, the other must allow only outgoing traffic. A **uugetty** could not be used on either machine in this case.

A machine's **inittab** entry would be respawning **getty** for "incoming" traffic, or have respawn turned off for "outgoing" traffic. In order for this type of link to work, one machine must be set up to "poll" the other. If the remote machine is allowing only incoming traffic, you must set up your 3B2 Computer to poll the remote machine (**uucpmgmt** subcommand **pollmgmt**). If the remote machine is allowing only outgoing traffic on the link, the remote machine must poll your 3B2 Computer.

Entries in the **/etc/inittab** file can be changed using the **uucpmgmt portmgmt** subcommand. The operation to be performed under **portmgmt** is the modify operation. The modify operation prompts you for the following information:

- Port name you want to modify (for /dev/tty21 - **tty21**)
- Direction of traffic on port (**bidirectional**, **incoming**, or **outgoing**)
- Transmission speed of the link (**9600** or **19200**).

The procedure will display the ports that are currently dedicated for use by UUCP (listed in **Devices** file). The port name to be modified must be one that is listed.

To access the **portmgmt** subcommand and modify entries in the **/etc/inittab** file, enter:

```
# sysadm portmgmt<CR>

Running subcommand 'portmgmt' from menu 'uucpmgmt',
BASIC NETWORKING UTILITIES MANAGEMENT

Note: After a brief introduction to the portmgmt
subcommand, the procedure interactively prompts you for
the information listed above.
```

Making Entries into the Systems File

An entry must be made into the **Systems** file for the machine associated with the direct link. This can be done using the **systemmgmt** subcommand and the add operation. The add operation will prompt you for the following information:

- Node name of system
- Type of device to call on (**direct**)
- Transmission speed of link (**9600** or **19200**)
- Device port used with link (for /dev/tty21 - **tty21**)
- Login ID used to log in on system (**nuucp**)
- Password used by above log in.

In order for the direct link to operate properly at high speeds, you must insert pauses (\p) between the characters being sent out for the login ID and the password. For instance, instead of nuucp, you should enter n\pu\pu\pc\p\pp when prompted for the login ID. Do *not* select the default by pressing RETURN (<*CR*>). The same applies for the password assigned.

To access the **systemmgmt** subcommand and add entries in the **Systems** file, enter:

```
# sysadm systemmgmt<CR>

Running subcommand 'systemmgmt' from menu 'uucpmgmt',
BASIC NETWORKING UTILITIES MANAGEMENT

Note: After a brief introduction to the systemmgmt
subcommand, the procedure interactively prompts you for
the information listed above.
```

On completion of the add operation, a new entry is added to the **Systems** file for the remote machine and an additional entry was created for the **Devices** file. When you use **devicemgmt** to create an entry for the link, it creates an entry similar to the one shown below.

```
Direct tty21 - 9600 direct
```

As discussed in Chapter 3, this type of entry is used only with the **cu** command. The UUCP programs require that the first field of a **Devices** entry be *System-Name* when associated with a direct link to another machine. This is why the add operation of **systemmgmt** wants to know the device port that will be used in the link to the remote machine. After it receives the port name, it creates a second entry for that port similar to the following:

```
Direct tty21 - 9600 direct
eagle tty21 - 9600 direct
```

The first entry will be used whenever the **cu** command is used to call "eagle," and the second entry will be used by **uucico** to call "eagle."

Chapter 6

MAINTENANCE

PAGE

Chapter 6

MAINTENANCE

GENERAL

This chapter discusses the maintenance of the Basic Networking Utilities. It discusses some of the common problems associated with the utilities and how to use the **Uutry** program for debugging. Also, included is a list of error messages and their probable causes.

COMMON PROBLEMS

Out of Space

The file system used to spool incoming or outgoing jobs can run out of space and prevent jobs from being sent or received. Not being able to receive jobs is the worse of the two conditions. When file space does become available, the 3B2 Computer will be flooded with the backlog of traffic. The shell script **uudemon.cleanu** should keep the spool directory (**/usr/spool/uucp**) from becoming large. This script should be started by **cron** once a day.

Faulty Automatic Call Units and Modems

The automatic dial modems and/or incoming modems occasionally cause problems that make it difficult to contact other computers or receive files. These problems are usually readily identifiable since the status files accessed by **uustat** give counts and reasons for contact failure. If a bad line is suspected, the **cu** command may be useful in trying to call another computer using the suspected line.

Administrative Problems

Sometimes it can be difficult to keep your **Systems** file up to date. This is because of changing telephone numbers, login IDs, and passwords on remote computers. This can be a costly problem since the automatic dial modem will be tied up calling a computer that cannot be reached. Be sure to contact the administrators of remote machines whenever you change your telephone number, login, or password and request that they show you the same consideration.

DEBUGGING

To verify that a computer on the network can be contacted, the **uucico** daemon can be invoked directly from a terminal. A shell script, **Uutry**, is provided for this purpose in the **/usr/lib/uucp** directory. This directory is normally not listed in most users' command search path. It must therefore be moved to an appropriate place if it is to be used by all users. If the person using the **uucp** login will be the only one using **Uutry**, it can be left where it is, since the HOME directory for the **uucp** login is **/usr/lib/uucp**.

The command line

```
Uutry eagle
```

will start the transfer daemon (**uucico**) with a moderate amount of debugging output. **Uutry** redirects the output into a temporary file

(**/tmp/eagle**) and executes a **tail -f** command on the file. This way, you can hit a " BREAK" to get back to the shell, and come back later to look at the output in **/tmp/eagle**.

If **Uutry** does not isolate the problem, you can attempt to transfer a file while watching the debugging output as follows:

```
uucp -r some-file eagle!~/some-name
```

The **-r** option will queue a job, but will not start the transfer daemon. Now proceed as before using **Uutry**. If any of these steps fail, support personnel may be needed to diagnose the problem. It will be much easier to diagnose the problem if the debugging output saved in the temporary file is available.

ERROR MESSAGES

This section lists the error messages associated with Basic Networking Utilities. There are two types of error messages. ASSERT errors are recorded in the **/usr/spool/uucp/.Admin/errors** file. STATUS errors are recorded in individual machine files found in the **/usr/spool/uucp/.Status** directory.

ASSERT Error Messages

When a process is aborted, ASSERT error messages are recorded in **/usr/spool/uucp/.Admin/errors**. These messages include the file name, sccsid, line number, and the text listed below. Usually, these errors are the result of file system problems. The " errno" (when present) should be used to investigate the problem. If " errno" is present in a message, it is shown as () in the following list.

CAN'T OPEN An open() or fopen() failed.

CAN'T WRITE	A write(), fwrite(), fprint(), etc. failed.
CAN'T READ	A read(), fgets(), etc. failed.
CAN'T CREATE	A create() call failed.
CAN'T ALLOCATE	A dynamic allocation failed.
CAN'T LOCK	An attempt to create a LCK (lock) file failed. Usually, this is a fatal error.
CAN'T STAT	A stat() call failed.
CAN'T CHMOD	A chmod() call failed.
CAN'T LINK	A link() call failed.
CAN'T CHDIR	A chdir() call failed.
CAN'T UNLINK	A unlink() call failed.
WRONG ROLE	This is an internal logic problem.
CAN'T MOVE TO CORRUPTDIR	An attempt to move some bad C. or X. files to the **/usr/spool/uucp/.Corrupt** directory failed. The directory is probably missing or has wrong modes or owner.
CAN'T CLOSE	A close() or fclose() call failed.

FILE EXISTS	The creation of a C. or D. file is attempted, but the file exists. This occurs when there is a problem with the sequence file access. Usually shows a software error.
No uucp server	A tcp/ip call is attempted, but there is no server for UUCP.
BAD UID	The uid cannot be found in the **/etc/passwd** file. The file system is in trouble, or the **/etc/passwd** file is inconsistent.
BAD LOGIN_UID	Same as previous.
ULIMIT TOO SMALL	The ulimit for the current user process is too small. File transfers may fail, so transfer is not attempted.
BAD LINE	There is a bad line in the **Devices** file; there are not enough arguments on one or more lines.
FSTAT FAILED IN EWRDATA	There is something wrong with the ethernet media.
SYSLST OVERFLOW	An internal table in gename.c overflowed. A big/strange request was attempted. Contact your AT&T Service Representative or authorized dealer.

TOO MANY SAVED C FILES	Same as previous.
RETURN FROM fixline ioctl	An ioctl, which should never fail, failed. There is a system driver problem.
BAD SPEED	A bad line speed appears in the **Devices/Systems** files (Class field).
PERMISSIONS file: BAD OPTION	There is a bad line or option in the **Permissions** file. Fix it immediately!
PKCGET READ	The remote machine probably hung up. No action need be taken.
PKXSTART	The remote machine aborted in a non-recoverable way. This can generally be ignored.
SYSTAT OPEN FAIL	There is a problem with the modes of **/usr/lib/uucp/.Status**, or there is a file with bad modes in the directory.
TOO MANY LOCKS	There is an internal problem! Contact your AT&T Service Representative or authorized dealer.
XMV ERROR	There is a problem with some file or directory. It is likely the spool directory, since the modes of the destinations were suppose to be checked before this process was attempted.

CAN'T FORK	An attempt to fork and exec failed. The current job should not be lost, but will be attempted later (**uuxqt**). No action need be taken.

Status Error Messages

Status error messages are messages that are stored in the **/usr/spool/uucp/.Status** directory. This directory contains a separate file for each remote machine that your 3B2 Computer attempts to communicate with. These individual machine files contain status information on the attempted communication, whether it was successful or not. What follows is a list of the most common error messages that may appear in these files.

OK	Things are OK.
NO DEVICES AVAILABLE	There is currently no device available for the call. Check to see that there is a valid device in the **Devices** file for the particular system. Check the **Systems** file for the device to be used to call the system.
WRONG TIME TO CALL	A call was placed to the system at a time other than what is specified in the **Systems** file.
TALKING	Self explanatory.

LOGIN FAILED	The log in for the given machine failed. It could be a wrong login/password, wrong number, a slow machine, or failure in getting through the Dialer-Token-Pairs script.
CONVERSATION FAILED	The conversation failed after successful startup. This usually means that one side went down, the program aborted, or the line (link) was dropped.
DIAL FAILED	The remote machine never answered. It could be a bad dialer or the wrong phone number.
BAD LOGIN/MACHINE COMBINATION	The machine called us with a login/machine name that does not agree with the **Permissions** file. This could be an attempt to masquerade!
DEVICE LOCKED	The calling device to be used is currently locked and in use by another process.
ASSERT ERROR	An ASSERT error occurred. Check the **/usr/spool/uucp/.Admin/errors** file for the error message and refer to the section *ASSERT Error Messages*.
SYSTEM NOT IN Systems	The system is not in the **Systems** file.

CAN'T ACCESS DEVICE	The device tried does not exist or the modes are wrong. Check the appropriate entries in the **Systems** and **Devices** files.
DEVICE FAILED	The open of the device failed.
WRONG MACHINE NAME	The called machine is reporting a different name than expected.
CALLBACK REQUIRED	The called machine requires that it calls your 3B2 Computer.
REMOTE HAS A LCK FILE FOR ME	The remote site has a LCK file for your 3B2 Computer. They could be trying to call your machine. If they have an older version of Basic Networking, the process that was talking to your machine may have failed leaving the LCK file. If they have the new version of Basic Networking, and they are not communicating with your 3B2 Computer, then the process that has a LCK file is hung.
REMOTE DOES NOT KNOW ME	The remote machine does not have the node name of your 3B2 Computer in its **Systems** file.
REMOTE REJECT AFTER LOGIN	The log in used by your 3B2 Computer to log in does not agree with what the remote machine was expecting.

REMOTE REJECT, UNKNOWN MESSAGE	The remote machine rejected the communication with your 3B2 Computer for an unknown reason. The remote machine may not be running a standard version of Basic Networking.
STARTUP FAILED	Login succeeded, but initial handshake failed.
CALLER SCRIPT FAILED	This is usually the same as " DIAL FAILED." However, if it occurs often, suspect the caller script in the **Dailers** file. Use **Uutry** to check.

Chapter 7

COMMAND DESCRIPTIONS

PAGE

Chapter 7

COMMAND DESCRIPTIONS

COMMAND SUMMARY

This chapter describes the commands of the Basic Networking Utilities. The commands are grouped into two categories:

- User Commands
- Administrative Commands.

In addition to the command descriptions, the format and options of the commands are discussed. Then, a sample usage of each command is presented to give you an idea of how the command is used. User commands are invoked during attempts to communicate with remote computers or obtain status information. A summary of these commands is shown in Figure 7-1.

COMMAND	DESCRIPTION
ct	Calls and connects a remote terminal to your 3B2 Computer.
cu	Calls a remote computer and allows you to log in on the called computer without logging off the 3B2 Computer.
uucp	Queues a file transfer to the specified path name on a remote computer.
uuto	Queues a file transfer to the public directory on a remote computer.
uupick	Searches the spool directory for files destined to the person initiating the command.
uux	Sets up a remote command execution to be executed on a specified computer.
uulog	Displays information stored in a specific computers log file.
uustat	Provides you with status information on queued transfers.
uuname	Provides you with the names of the computers that are part of the network.

Figure 7-1. User Commands

Administrative commands are used to do administrative tasks on the UUCP facility. Some are invoked automatically via **cron** while others may be invoked manually if you are logged in as **root** or **uucp**. A summary of the administrative commands is shown in Figure 7-2.

COMMAND	DESCRIPTION
uucheck	Checks for the presence of the required UUCP files and directories. Also checks for errors in the **Permissions** file.
uucleanup	Removes specific files from the spool directory that are more than a specified number of days old.
Uutry	Invokes the transfer daemon (**uucico**) to call a specified machine with a moderate amount of debugging.

Figure 7-2. Administrative Commands

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the Basic Networking commands. The format is as follows:

- **General:** The purpose of the command is defined. This includes any uncommon or special information about the command.
- **Command Format:** The basic command line format (syntax) is defined and the various arguments and options are discussed.
- **Sample Command:** Example command line entries and responses are presented to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax:

- The basic command is shown in bold type.
  For example: **command**
- Arguments that you must supply to the command are shown in an italic type.
  For example: **command** *argument*
- Command options and arguments that do not have to be supplied are enclosed in brackets [].
  For example: **command** [*arguments*]
- The pipe symbol (|) is used to separate arguments when one of several forms of an argument can be used.
  For example: **command** [*argument1* | *argument2*]

In the sample command discussions, user inputs and 3B2 Computer response examples are shown as follows.

```
This style of type is used to show system generated
responses displayed on your screen.
```

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <*CR*> carriage return, <*CTRL d*> control d, <*ESC g*> escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

Also, in the sample command discussion, the dollar sign ($) is used as the system prompt. The system prompt could be the number symbol (#). In either case, you should not enter the prompt.

USER COMMANDS

ct – Generate a getty Process to a Remote Terminal

General

The **ct** command attempts to connect a remote terminal to your 3B2 Computer by dialing the phone number of the modem attached to the terminal. This modem must be able to automatically answer the call when it is received. When **ct** detects that the call has been answered, it issues a **getty** (login) process and allows the user of the remote terminal to log in on the 3B2 Computer. A user at a remote terminal may call your 3B2 Computer, log in, and issue a **ct** command to request that the 3B2 Computer hang up the current line and call the remote terminal back.

If **ct** cannot find an available dialer, it shows that the dialer(s) are busy and asks if it should wait until one becomes available. If you respond yes, it asks how long it should wait for one.

Command Format

The **ct** command has the format

ct [*options*] *telno*

where *telno* is the telephone number of a remote terminal with equal signs (=) for secondary dial tones and dashes (-) for delays.

The **ct** command has the following *options*:

-h Prevents **ct** from disconnecting the current line and allowing the 3B2 Computer to call the remote terminal back.

-v	Requests that a running narrative of the attempt to contact the remote terminal be displayed on the screen.
-w*n*	Overrides the dialogue where **ct** asks if it should wait for a dialer. *n* is the number of minutes that **ct** should wait for a dialer.
-s*speed*	Selects the transmission rate of the modem attached to the remote terminal where *speed* is expressed in baud (default 1200).
-x*n*	Causes debugging output to be displayed on the screen where *n* is a single-digit (0 through 9) indicating the level of debugging; 9 being the highest level of debugging.

Sample Command Usage

If you are logged in on the 3B2 Computer through a local terminal, and you want to connect a remote terminal to your 3B2 Computer, you may enter:

```
$ ct -h -w5 -s1200 9=9323497<CR>
```

that calls the modem connected to 932-3497 using a 1200 baud dialer. If a dialer is not available, the **-w5** option causes **ct** to wait 5 minutes for a dialer to become available before it quits. The **-h** option tells **ct** not to disconnect the local terminal (terminal used to input the command) from the 3B2 Computer.

If you are at a remote terminal some distance away from your 3B2 Computer and you do not want to get billed for long distance charges, you can call the 3B2 Computer initially, log in, and enter the following **ct** command line:

```
$ ct -s1200 9=9323497<CR>
```

If no device is available at the time, the following dialogue is received:

```
1 busy dialer at 1200 baud
Wait for dialer?
```

If you reply no (n), the **ct** command exits. If you reply yes (y), you are prompted to show how long to wait:

```
Time, in minutes?
```

If a dialer is available, **ct** responds with:

```
Allocated dialer at 1200 baud
```

this indicates that it found a dialer. In any case, **ct** asks if you want the line connecting your remote terminal to the 3B2 Computer to be dropped:

```
Confirm hangup?
```

If you enter yes (y), you are logged off and **ct** calls the remote terminal back if a dialer was, or becomes, available. If you enter no (n) the **ct** command exits and you are still logged onto the 3B2 Computer.

cu - Call Another UNIX System

General

The **cu** command calls a remote computer and links your 3B2 Computer to the called computer. The **cu** command can use either direct links, the telephone network, or Local Area Networks (LAN) to establish a connection to a remote computer. Once the connection is made, the called computer will prompt you to log in on the computer. After you are logged in on the called computer, you can transfer ASCII coded files from one computer to another and execute commands on either computer.

> ***Note:*** The **cu** command does not have error detection/correction capability. Therefore, it is possible that some loss or corruption of data may occur during file transfers.

Command Format

The **cu** command has the format:

cu [*options*] *telno* | *systemname*

Where:

telno is the telephone number of a remote computer with equal signs (=) for secondary dial tones and dashes (-) for four second delays.

systemname is a UUCP system name that appears in the **Systems** file. Here, **cu** will obtain the telephone number and baud rate from the **Systems** file and search for a dialer to use for the connection. So, the **-s**, **-n**, and **-l** options should *not* be used with *systemname*. The **uuname** command will display the machines listed in the **Systems** file.

The **cu** command has the following *options*:

-s*speed*	Specifies the transmission rate of the device to be used in the connection where *speed* is the transmission rate (usually 300 or 1200); 300 is the default value.
-l*line*	Used if a specific device is to be used in the connection where *line* is the name used to reference a specific line (port) to which the device is connected. This is useful when calling a computer that is hardwired to your 3B2 Computer.
-h	Used to emulate local echo when the called computer expects terminals to be in the half-duplex mode.
-t	Used when dialing an ASCII terminal set up with automatic answer.
-d	Causes diagnostic traces to be printed.
-e (**-o**)	Designates even (odd) parity to be generated and sent to the remote computer.
-n	Shows that **cu** will prompt you for the telephone number instead of taking it from the command line.

Once the connection is made and you are logged in on the remote computer, any standard (keyboard) input is sent to the remote computer. Figure 7-3 shows the strings that you can execute while you are connected to a remote machine through **cu**.

STRING	INTERPRETATION
~.	Terminates the conversation.
~!	Escape to the local system (3B2 Computer) without dropping the link. To get back to the remote machine, CTRL-d.
~!*cmd*	Execute *cmd* on the local system (3B2 Computer).
~$*cmd*	Execute *cmd* on the 3B2 Computer and send its output to the remote computer.
~%**cd** *path*	Change the directory on the local system where *path* is the path name or directory name.
~%**take** *from* [*to*]	Copy file named *from* (on the remote computer) to file named *to* on the 3B2 Computer. If *to* is omitted, the *from* argument is used in both places.
~%**put** *from* [*to*]	Copy file named *from* (on 3B2 Computer) to file named *to* on the remote computer. If *to* is omitted, the *from* argument is used in both places.
~~...	Send the line ~... to the remote computer.
~%**break**	Transmits a BREAK to the remote machine (can also be specified as ~%**b**).

Figure 7-3. cu Command Strings (Sheet 1 of 2)

STRING	INTERPRETATION
~%nostop	Turn off the handshaking protocol for the remainder of the session. This is useful when the remote computer does not respond properly to the protocol characters.
~%debug	Toggles the **-d** debugging option on or off (can also be specified as **~%d**).
~t	Displays the values of the terminal I/O (termio) structure variables for your terminal (useful for debugging).
~l	Displays the values of the termio structure variables for the remote communication line (useful for debugging).

Figure 7-3. cu Command Strings (Sheet 2 of 2)

Note 1: The use of **~%put** requires **stty** and **cat** on the remote machine. It also requires that the current erase and kill characters on the remote machine be identical to the current ones on the 3B2 Computer.

Note 2: The use of **~%take** requires the existence of **echo** and **cat** on the remote machine. Also, **stty tabs** mode should be set on the remote machine if tabs are to be copied without expansion.

Sample Command Usage

To communicate with the remote computer **eagle**, enter:

```
$ cu -s1200 9=847-7867<CR>
```

where **eagle**s phone number is 847-7867. The **-s1200** option forces **cu** to use a 1200 baud dialer to call **eagle**. If the **-s** option is not specified, **cu** uses the default speed, 300 baud. When **eagle** answers the call, messages similar to the following appear on the screen:

```
connected

login:
```

At this point, you should enter your login ID and password.

Suppose you want a copy of a file named **proposal** from the remote computer. If **proposal** is located in the current directory (on remote computer), the following string copies the file **proposal** from the remote computer to the 3B2 Computer and places it in the current directory (on local computer) under a file named **proposal**:

```
$~%take proposal<CR>
```

To send a file named **minutes** (in the current directory) on your 3B2 Computer to the remote computer, enter:

```
$ ~%put minutes minutes.9-18<CR>
```

This copies the file **minutes** over to the remote computer and places it in a file named **minutes.9-18**.

uucp - UNIX-to-UNIX System Copy

General

The **uucp** command will queue a file transfer as described in Chapter 2. The **uucp** transfer is one that is transparent to the user. Once the command is entered, you may continue with other processes while the UUCP programs attempt to do the transfer.

Command Format

The **uucp** command has the format

uucp [*options*] *source-file system!destination-file*

where the *source-file* and *destination-file* may contain the prefix *system-name!*, which shows the computer where the file resides or where it will be copied. If *system-name* is omitted, the local machine is assumed. The *system-name* argument must be one that UUCP knows about. In other words, it must be listed in the **Systems** file (see **uuname**).

Usually, the *source-file* and *destination-file* will be a specified path name on a given computer. If a shell metacharacter is used in the path name, it will be expanded on the remote computer. Path names may be as follows:

- A full path name.
- A path name preceded by ~*user* where *user* is a login on the specified computer and is replaced by that users HOME directory.
- A path name preceded by ~/*destination* where *destination* is appended to **/usr/spool/uucppublic**. This *destination* will be treated as a file name unless more than one file is being transferred, or if the destination is already a directory. To ensure that *destination* is a directory, follow it with a "/" (~/**destination**/).

- Anything else is prefixed by the current directory.

If *destination-file* is a directory, the last part of the *source-file* name is used. The various ways to specify path names are shown in "Sample Commands."

The **uucp** command has the following *options*:

-c	Use the source file when copying out to the remote computer rather than copying the file to the spool directory (default).
-C	Copy the source file to the spool directory.
-d	Make all necessary directories for the file copy (default).
-f	Do not make intermediate directories for the file copy.
-g*grade*	The *grade* is a single letter/number. Lower ASCII sequence characters will cause the job to be transmitted earlier during a particular conversation.
-j	Output the job identification number on the screen. This is an ASCII sequence that UUCP assigns to each job. It can be used with the **uustat** command to obtain status information or stop a job.
-m	Send mail to the requester when the copy is completed.
-s*file*	Report status of the transfer in *file*. Note that *file* must be a full path name.
-n*user*	Notify *user* on the remote computer that a file was sent.
-r	Queue the job, but do not start the transfer daemon (**uucico**) and attempt to transfer the job.

-x*debug_level* Produces debugging output on the screen where *debug_level* is a number between 0 and 9; higher numbers give more detailed information.

Sample Command Usage

To send the file named **minutes** to the remote computer **eagle**, you could enter:

```
$ uucp -s -C -j -ngws minutes eagle!/usr/gws/minutes<CR>
eagleN3f45
$
```

This sends the file **minutes** (located in current directory on 3B2 Computer) to the computer **eagle** and places it under the path name **/usr/gws** in a file named **minutes**. When the transfer is complete, the user **gws** on the remote computer receives mail indicating that a file has been received. You also receive mail indicating a successful or failed transfer as requested by the **-s** option. The **-j** option requested that the job ID (`eagleN3f45`) be displayed.

The previous example uses a full path name to provide the *destination-file*. There are two other ways the *destination-file* can be specified:

1. The log in directory of **gws** can be specified through use of the ~ character shown below:

   eagle!~gws/minutes

   would be interpreted as

   eagle!/usr/gws/minutes

2. The **uucppublic** area is referenced by a similar use of the prefix ~ preceding the path name. For example:

   eagle!~/gws/minutes

   is interpreted as

   /usr/spool/uucppublic/gws/minutes

uuto - Public UNIX-to-UNIX System Copy

General

The **uuto** command uses the same processes as the **uucp** command when transferring a file to a remote computer. The only difference is that the **uuto** command will always send the file to the public area on the remote computer. The person to whom the file was sent will be notified by mail when the transfer is complete.

Command Format

The **uuto** command has the format

uuto [*options*] *source-file destination*

where the *source-file* can be any of the following:

- A full path name
- A file located in the current directory.

Destination has the format

system!user

where *system* is the name of the remote computer and *user* is the login name of the user to which the file is being sent. When the transfer is complete, the transferred file(s) will be placed under

/usr/spool/uucppublic/receive/*user/my3b2/files*

where *user* is the remote user's login name as specified in the **uuto** command and *my3b2* is the name of your 3B2 Computer.

The **uuto** command has the following *options*:

-m Causes mail to be sent to you (sender) when the transfer is complete.

-p Causes the *source-file* to be copied to the spool directory on your 3B2 Computer before transmission to the remote computer.

Sample Command Usage

To send the file **minutes** to user **gws** on the remote computer **eagle**, you can enter:

```
$ uuto -m -p minutes eagle!gws<CR>
$
```

This makes a copy of the file **minutes** (from the current directory) in the spool directory before the transfer. After the transfer is complete, you will receive mail indicating the completion of the transfer and the remote user **gws**, receives mail indicating that a file has been received from the remote computer **my3b2**. The file is placed on the remote computer under

/usr/spool/uucppublic/receive/gws/my3b2/minutes

uupick - uuto File Retrieval

General

The **uupick** command is used with the **uuto** command. After a file is sent to you via the **uuto** command, that file resides in the public area under a directory that has the same name as your login ID. The **uupick** command will search the public area to see if a directory exists that matches your login ID. If this check is true, **uupick** waits for you to tell it what to do with the file(s) located in that directory.

Command Format

The **uupick** command has the format

uupick [*-s system*]

where the **-s** option tells **uupick** to search for files sent only from the remote computer *system*.

If **uupick** finds a directory that matches your log in name, it will respond with:

from *sys*: **[file** *file-name*] **[dir** *directory-name*]

at which point you should tell **uupick** what you want to do with the file(s). The permitted options are shown in Figure 7-4.

COMMAND	INTERPRETATION
<CR>	Go to next entry.
d	Delete the entry.
m [*dir*]	Move the entry to the directory *dir*. If *dir* is not a full path name, a destination relative to the current directory is assumed. If *dir* is not specified, the current directory is assumed.
a [*dir*]	Same as **m** except **a** moves all files sent from *sys*.
p	Print the contents of the file.
q	Stop.
!*command*	Escape to the shell and execute *command*.
*	Print a command summary.

Figure 7-4. uupick Options

Sample Command Usage

You are user **gws** on the computer **eagle**. You log in on **eagle** and receive mail indicating that you have received a file from the **my3b2** computer. You can then enter the following command line.

```
$ uupick -s my3b2<CR>
```

and wait for **uupick** to prompt you with

```
from my3b2: file minutes?
```

At this point, you can look at the file **minutes** by entering:

```
p<CR>
```

This will display the file **minutes** on the screen. After the file has been displayed, the question mark reappears and **uupick** waits for further instructions. If you want to save the file, move it to your log in directory by entering:

```
m $HOME<CR>
4 blocks
$
```

After the file is moved, the number of blocks taken up by the file is displayed, and you are returned to the shell.

uux - UNIX-to-UNIX System Command Execution

General

The **uux** command is used to execute UNIX System command strings on remote computers. The **uux** command will gather files from various computers, execute a command on a specified computer, and send the standard output to a file on a specified computer. The execution of certain commands may be restricted on the remote machine (**Permissions** file). **Uux** will notify you by mail if the requested command was disallowed.

Command Format

The **uux** command has the format

uux [*options*] *command-string*

where *command-string* is made up of one or more arguments. All special shell characters such as " <>|^" must be quoted either by quoting the entire *command-string* or quoting the character as a separate argument. Within the *command-string* the command and file names may contain a *system-name!* prefix. All arguments that do not contain a *systemname* is interpreted as command arguments. File names may be as follows:

- A full path name
- Anything prefixed by the current directory (local machine).

The **uux** command has the following *options*:

–	Shows that the standard input for *command-string* is taken from the standard input of the **uux** command.
-a*name*	Use *name* as the user ID, replacing the user ID.

-b	Return standard input to the command if the exit status is non-zero.
-c	Do not copy local files to the spool directory for transfer to the remote machine (default).
-C	Force the copy of local files to the spool directory for transfer.
-g*grade*	The grade is a single letter/number. Lower ASCII sequence characters will cause the job to be transmitted earlier during a particular conversation.
-j	Output the job identification number on the screen. This is an ASCII sequence that UUCP assigns to each job. It can be used with the **uustat** command to obtain status information or to end a job.
-n	Shows that no notification is to be sent to the remote user.
-p	Same as "-".
-r	Do not start the file transfer daemon (**uucico**), just queue the job.
-s*file*	Report the status of the transfer in *file*.
-x*debug_level*	Produces debugging output on the screen where *debug_level* is a number between 0 and 9; higher numbers give more detailed information.
-z	Send success notification to the local user.

Sample Command Usage

If your 3B2 Computer is hardwired to a larger host computer you can use **uux** to get printouts of files that reside on your 3B2 Computer by entering:

```
$ pr minutes|uux -p host!lp<CR>
$
```

This command line queues the file **minutes** to be printed on the area printer of the computer **host**.

uulog - UUCP Log Inquiry

General

The **uulog** command is used to display the contents of machine log files associated with a specific remote machine. Each machine will have log files for **uucico** and **uuxqt** processes that have been attempted. These log files reside in the **/usr/spool/uucp/.Log** directory.

Command Format

The **uulog** command has the format

uulog [*options*]

where the following *options* are available:

-s*sys*	Indicates to print the contents of the Log files for the remote machine *sys*.
-f*sys*	Performs a **tail -f** on the file transfer log for machine *sys*.

The two options below can be used with the two previous options that can only be used one at a time:

-x	Displays the **uuxqt** log file for the specified machine.
-number	Shows that the **tail** command should be performed with *number* lines.

Sample Command Usage

If you want to print the current log information associated with the remote computer **eagle**, you can enter:

```
$ uulog -seagle<CR>
uucp eagle  (07/19-1:41:47,1798,0) SUCCEEDED (call to eagle )
uucp eagle  (07/19-1:41:54,1798,0) OK (startup)
uucp eagle  (07/19-1:41:56,1798,0) OK (conversation complete contty 51)
uucp eagle  (07/19-2:11:47,1824,0) SUCCEEDED (call to eagle )
uucp eagle  (07/19-2:11:56,1824,0) OK (startup)
uucp eagle  (07/19-2:11:57,1824,0) OK (conversation complete contty 52)
uucp eagle  (07/19-2:41:48,1846,0) SUCCEEDED (call to eagle )
uucp eagle  (07/19-2:41:58,1846,0) OK (startup)
uucp eagle  (07/19-2:42:00,1846,0) OK (conversation complete contty 55)
uucp eagle  (07/19-3:11:47,1872,0) SUCCEEDED (call to eagle )
uucp eagle  (07/19-3:11:58,1872,0) OK (startup)
uucp eagle  (07/19-3:12:00,1872,0) OK (conversation complete contty 55)
$
```

uustat - UUCP Status Inquiry and Job Control

General

The **uustat** command is used to display general status information of queued jobs (transfers or executions). It also gives you a means of controlling jobs that have been queued, monitoring the size of job queues, and the status of the last attempt to contact all machines.

Command Format

The **uustat** command has the format

uustat [*options*]

where only *one* of the following *options* can be requested at a time:

-a List the jobs that are queued.

-m Report the success/failure status of the last attempt to communicate with all machines listed in the **Systems** file.

-p Executes a **ps -flp** for each process ID that has an associated lock (LCK.) file. This displays detailed information on the UUCP jobs that are currently being processed.

-q List the jobs queued for each machine. If a status file exists for a machine, its date, time, and status information are displayed. If a number appears in () next to the number of C. or X. files, it is the age in days of the oldest C. or X. file for that machine. The "Retry" field represents the number of hours until the next possible call. The "Count" field is the number of failure attempts. For machines with a moderate amount of outstanding jobs, this could take 30 seconds or more (real-time).

-r*jobid*	Rejuvenate the job whose ID is *jobid*. This prevents the cleanup daemon from deleting the job until the jobs modification time reaches the limit imposed by the daemon.

There are two options that can be requested by themselves or together:

-s*sys*	Display the status of all UUCP requests for the remote computer *sys*.
-u*user*	Display the status of all UUCP requests issued by *user*.

Sample Command Usage

One useful form of the **uustat** command is used to display the status of queued transfers. Suppose that you requested a file transfer to the remote computer **eagle**, you can display the status of your requested transfer(s) by entering:

```
$ uustat<CR>
eagleN3f67   07/19-4:58   S   unix  chp 374  /usr/chp/minutes
$
```

If you decide not to send **minutes** to **eagle**, you can kill the request using the **-k** option by entering:

```
$ uustat -keagleN3f67<CR>
Job: eagleN3f67 successfully killed
$
```

where **eagleN3f67** is the job ID of the requested transfer.

uuname - UUCP Network Node Names

General

The **uuname** command is useful in determining the names of those computers that are part of your network.

Command Format

The **uuname** command has the format

uuname [*-l*]

where *-l* will display only the name of the local computer.

Sample Command Usage

To display the names of all the remote computers that are part of your network, enter:

```
$ uuname<CR>
```

This displays the names of those remote computers you can communicate with using the UUCP facilities.

To display the node name of your 3B2 Computer, enter:

```
$ uuname -l<CR>
```

ADMINISTRATIVE COMMANDS

uucleanup - UUCP Spool Directory Cleanup

General

The **uucleanup** command is used to clean-up the UUCP spool directory (**/usr/spool/uucp**). This is usually executed automatically by **uudemon.cleanu**, but the command can be invoked manually to clean-up certain files in the spool directory. The **uucleanup** command will inform owners of send/receive requests of machines that cannot be reached, return mail that cannot be delivered, and delete or execute **rnews** for rnews type files. Also, can warn users of requests that have been waiting for a given amount of days. You must log in as **uucp** or **root** to invoke the **uucleanup** command.

Command Format

The **uucleanup** command has the format

/usr/lib/uucp/uucleanup [*options*]

where the following *options* are available:

-C*time*	Any C. (work) files greater or equal to *time* days old will be removed with appropriate information sent to the owner (default 7 days).
-D*time*	Any D. (data) files greater or equal to *time* days old will be removed. An attempt will be made to deliver mail messages and execute rnews when appropriate (default 7 days).
-W*time*	Any C. (work) files equal to *time* days old (default 1 day) will cause a mail message to be sent to the owner warning about the delay in contacting the remote. The message

includes the JOBID, and the mail message. The administrator may include a message line telling who to call to check the problem (-m option).

-X*time* Any X. (execute) files greater or equal to *time* days old will be removed. The D. files are probably not present (if they were, the X. would have been executed). But if there are D. files, they will be taken care of by D. processing (default 2 days).

-m*string* *String* will be included in the warning message generated by the -W option. The default line is

See your local administrator to locate the problem.

-o*time* Other files whose age is more than *time* days will be deleted (default 2 days).

-s*system* Execute for the directory *system* in the spool directory. The directory *system* is the spool directory for the specific computer.

Sample Command Usage

The following **uucleanup** command line is the one that executes out of uudemon.cleanup:

```
/usr/lib/uucp/uucleanup  -D7 -C7 -X2 -o2 -W1
```

This deletes all " work" and " data" files equal or greater than 7 days old. Any other files (including " execute") are removed if they are equal or greater than two days old. The -W1 option sends a message to the user when the file remains in the spool for one day.

Uutry - Try to Contact a Machine With Debugging On

General

The **Uutry** program is used to invoke **uucico -s***systemname* (with a moderate amount of debugging output) to try contacting the specified machine. The debugging output is placed in **/tmp/***systemname*, as well as displayed on the screen. A **RUBOUT** or **BREAK** will return the terminal back to the shell while **uucico** continues to run, putting its output in **/tmp/***systemname*. The minimum retry time for a machine that is busy or does not answer is 5 minutes. The **uustat -m** output will show if a machine is busy or does not answer. You cannot attempt a **Uutry** as long as a " retry" exists for the particular system. Notice that **Uutry** is initial capped.

Command Format

The **Uutry** command has the following format:

/usr/lib/uucp/Uutry [*options*] *systemname*

where *systemname* is the name of the remote machine to be called.

The **Uutry** command has the following *options*:

-x*debug_level*	Overrides the default debugging level (5). The *debug_level* is a single-digit (0 through 9) number with higher numbers providing more debugging output.
-r	Allows you to input a **Uutry** command before the retry time has expired.

Sample Command Usage

If you have experienced problems in communicating with a remote machine, use the **Uutry** command to aid in the resolution of the problem, shown below:

```
$ /usr/lib/uucp/Uutry eagle<CR>
conn(eagle)
Device Type ACU wanted
expect: ("")
got it
sendthem (DELAY
^M)
expect: (>)
^M^JAT&T ACU/Modem ^M^J1200 BPS^M^J>got it
sendthem (PAUSE
<NO CR>)
expect: (E)
9^M^JSUREgot it
sendthem (<NO CR>)
expect: (:)
? (Y/N)y^M^JNO.:got it
sendthem (ECHO CHECK ON
 3P2P5P1P^M)
expect: (>)
^M^M^J>got it
sendthem (9<NO CR>)
expect: (OK)
^M^JDIALING: 3251^M^JOKgot it
getto ret 5
expect: (in:)
^M^J^J^M^JEAGLE login:got it
sendthem (^M)
expect: (word:)
 ^M^J>nuucp^M^JPassword:got it
sendthem (^M)
Login Successful: System=eagle
wmesg 'U'g
Proto started g
wmesg 'H'
wmesg 'H'Y
send OO 0,Conversation Complete: Status SUCCEEDED
$
```

Additional debugging output can be obtained by using the **-x** option (i.e. **-x9**).

uucheck - Check UUCP Directories and Permissions File

General

The **uucheck** command is used to check for the presence of files and directories required by UUCP. It also checks for obvious errors in the **Permissions** file.

Note: A **root** or **uucp** log in is required to execute **uucheck**.

Command Format

The **uucheck** command has the format

/usr/lib/uucp/uucheck [*options*]

where the following **options** are available:

-v	Provides a detailed explanation of how the UUCP programs will interpret the **Permissions** file.
-x*debug_level*	Produces debugging output where *debug_level* is a single-digit (0 through 9) number with higher numbers providing more debugging output.

Sample Command Usage

To see a detailed explanation of how the UUCP programs would interpret your **Permissions** file, enter:

```
# /usr/lib/uucp/uucheck -v<CR>
*** uucheck:  Check Required Files and Directories
*** uucheck:  Directories Check Complete

*** uucheck:  Check /usr/lib/uucp/Permissions file
** LOGNAME PHASE (when they call us)

When a system logs in as: (nuucp)
                        We DO allow them to request files.
                        We WILL send files queued for them on this call.
                        They can send files to
                           /
                        They can request files from
                           /
                        Myname for the conversation will be my3b2.
                        PUBDIR for the conversation will be /usr/spool/uucppublic.


** MACHINE PHASE (when we call or execute their uux requests)

When we call system(s): (eagle) (raven) (hawk)
                        We DO NOT allow them to request files.
                        They can send files to
                           /usr/spool/uucppublic (DEFAULT)
                        Myname for the conversation will be my3b2.
                        PUBDIR for the conversation will be /usr/spool/uucppublic.

Machine(s): (eagle) (raven) (hawk)
CAN execute the following commands:
command (rmail), fullname (rmail)
command (lp), fullname (lp)
command (uuxqt), fullname (uuxqt)

*** uucheck:  /usr/lib/uucp/Permissions Check Complete
#
```

Replace this

page with the

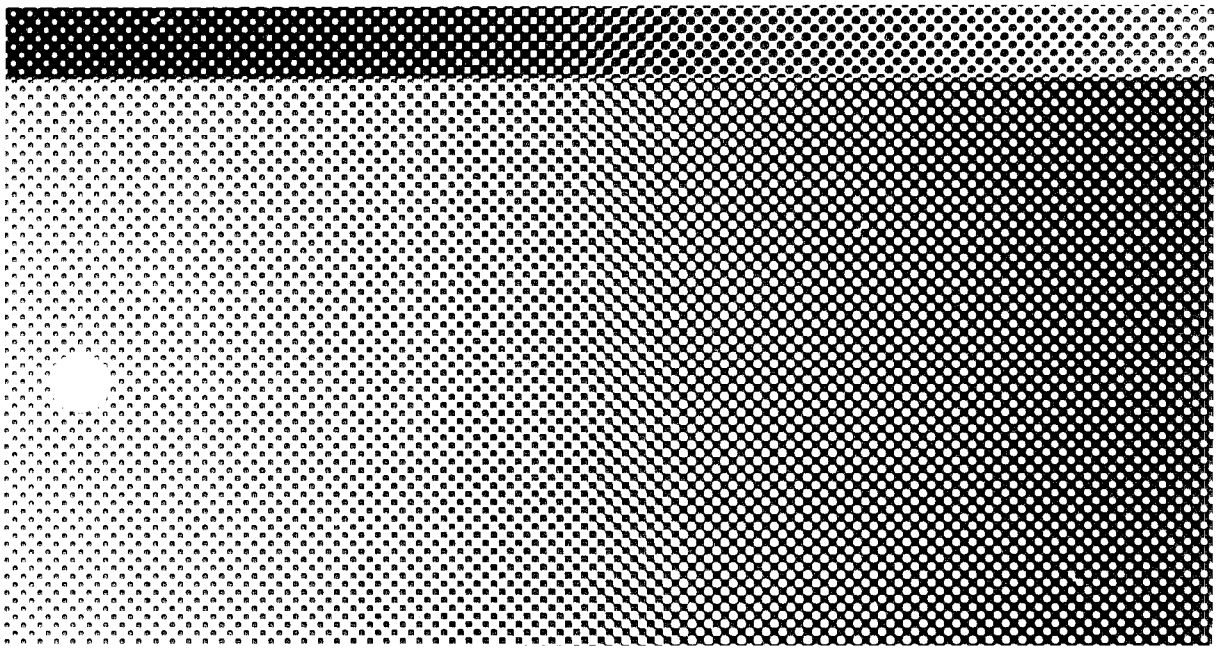
CARTRIDGE TAPE

tab separator.

AT&T

AT&T 3B2 Computer

UNIX™ System V Release 2.0 Cartridge Tape Utilities Guide



CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

GENERAL

This guide describes command syntax and use of the Cartridge Tape Utilities available with your AT&T 3B2 Computer.

The commands and procedures described in this guide are for use by sophisticated users who have a cartridge tape drive, and would like to do the following:

- Create a backup cartridge tape.
- Load files from a backup cartridge tape.
- Obtain information about a cartridge tape.
- Format a cartridge tape.
- Use the cartridge tape as an intermediate device to reorganize a file system.

- Copy file archives to the cartridge tape.
- Copy file archives from the cartridge tape.

To use these commands the following utilities must be installed:

- *Cartridge Tape Utilities*
- *System Administration Utilities.*

In addition, if your 3B2 Computer is configured with a AT&T/XM, the following prerequisites must also be met:

1. *3B2/XM Administration Utilities* must be installed on the 3B2 Computer
2. An AT&T/XM containing a cartridge tape drive must be connected to the 3B2 Computer.

Before proceeding with this guide, be sure you have read the *AT&T/XM Manual.*

Some of the commands described in this guide appear to accomplish the same result. They copy data from the disk to the cartridge tape. Each command, however, has advantages and disadvantages. When you become familiar with each command, you will know what command to use for different circumstances.

GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this guide is organized as follows:

Chapter 2, "COMMAND DESCRIPTION," describes the command formats (syntax) for each command in the *Cartridge Tape Utilities*. The descriptions include the purpose of the command, a discussion of the command syntax and options, and examples of using each command.

Appendix, "ERROR MESSAGES," contains the **UNIX*** System Error Messages pertaining to the Cartridge Tape Utilities.

* Trademark of AT&T

Chapter 2

COMMAND DESCRIPTIONS

PAGE

Chapter 2

COMMAND DESCRIPTIONS

COMMAND SUMMARY

The *Cartridge Tape Utilities* provide seven UNIX System commands. These commands allow you to do various operations on the cartridge tape drive. A summary of these commands is provided in Figure 2-1.

COMMAND	DESCRIPTION
cmpress	Improves disk performance by cleaning up fragmentation of a disk file system.
ctccpio	Copies file archives to and from a cartridge tape using streaming mode.
ctcfmt	Formats or reformats a cartridge tape.
ctcinfo	Displays information about a cartridge tape drive or a cartridge tape.
finc	Does an incremental backup on a cartridge tape.
frec	Recovers files from a backup cartridge tape.
tar	Saves and restores files on a cartridge tape.

Figure 2-1. Cartridge Tape Utilities - Command Summary

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the commands. The format is as follows:

- **General:** The purpose of the command is defined. Any uncommon or special information about the command is also provided.
- **Command Format:** The basic command format (syntax) is defined and the various arguments and options are discussed.
- **Sample Command Use:** Example command line entries and system responses are provided to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example: **command** is in bold type
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument*
- Command options and arguments that do not have to be supplied are enclosed in brackets ([]). For example: **command** *[optional arguments]*
- The pipe symbol (¦) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example:
  command *[argument1 ¦ argument2]*

In the sample command discussions, the lines that you input are ended with a carriage return. This is shown by using <*CR*> at the end of the lines.

Refer to the *AT&T 3B2 User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

Refer to the Appendix for the UNIX System Error Messages that pertain to the *Cartridge Tape Utilities*.

The following conventions are used to show your terminal input and the system output:

```
This style of type is used to show system generated
responses displayed on your screen.
```

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <*CR*> carriage return, <*CTRL d*> control d, <*ESC g*> escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

cmpress — Reorganizes a File System to Improve Access Time

General

The **cmpress** command is a shell script that internally reorganizes the free space to improve access time. The **cmpress** command works by first writing the named file system onto a formatted cartridge tape. Then, the original copy of the file system is removed from the hard disk, and the free block list is sorted into sequential order. Finally, the file system is copied from the tape back onto the hard disk in such a way that free space previously scattered throughout the file system are collected together.

Since the file system is destroyed during the compression process, it is recommended that an up-to-date backup of the file system be available before compressing.

All file systems can be compressed except **root** and **/usr** through the **sysadm tapemgmt** commands. Refer to the *AT&T/XM Manual*, under Chapter " Using Your AT&T/XM." The **root** file system cannot be compressed using the **cmpress** command. Compressing the **/usr** file system is more complex because the **sysadm** facilities reside in the **/usr** file system. A scheme for compressing **/usr** is given in the following *Sample Commands*.

The **cmpress** command requires the user to be logged in on the console terminal as **root**. This command should only be executed when adequate time is available, such as during off hours. Execution time depends on the size of the file system being compressed.

Command Format

The **cmpress** command has the following format:

cmpress ctape? (where **?** is the ctape drive number)

Sample Commands

The following steps shows the various commands it will take to compress the file system */usr*, using the **cmpress** command. At the end of the step procedure is a screen display of the commands as they were entered.

Step 1. See how **/usr** is mounted, by entering the **mount** command. Make note of the **/dev/dsk/c?d?s?** information for **/usr**. This will be used several times in this example. The **?** will vary according to the partitions and mount point.

Step 2. Take the system down to a single user mode by entering the **shutdown** command.

Step 3. Mount the **/usr** file system by using the **/dev/rdsk/c?d?s?** information from the **mount** command in Step 1.

Step 4. Compress the file system by entering the **cmpress** command and the cartridge tape drive to be used.

Step 5. When compression is complete, unmount **/dev/rdsk/c?d?s?** by entering the **umount** command.

Step 6. Now bring the system back to the multi-user mode by entering the **init 2** command.

COMPRESSING THE **USR** FILE SYSTEM IS COMPLETE.

```
# mount<CR>
/ on /dev/dsk/c?d?s? read/write on Fri July 19 06:30:22 1985
/ usr on /dev/dsk/c?d?s? read/write on Fri July 19 06:32:22 1985
/ usr2 on /dev/dsk/c?d?s? read/write on Thu July 18 08:47:22 1985
# shutdown -i1<CR>

Several messages will appear, ending with:

INIT: SINGLE USER MODE

#mount /dev/rdsk/c?d?s? /usr<CR>
# cmpress ctape1<CR>
Mounted file systems available for compression:
 /usr /usr2
Enter the file system you want to compress [q]: /usr<CR>
Insert tape into ctape1 drive, wait for re-tension pass to complete,
and press the <RETURN> key when ready [q]:<CR>
Copying file system to tape

Verify pass begins.
Verify pass complete.

(number) blocks
Copy to tape complete.
 Removing file system.
Consolidating the freelist to reorganize the space

 /dev/dsk/c?d?s?
 File System: usr Volume: 1.1

 ** Phase 1 - Check Blocks and Sizes
 ** Phase 2 - Check Pathnames
 ** Phase 3 - Check Connectivity
 ** Phase 4 - Check Reference Counts
 ** Phase 5 - Check Free List (Ignored)
 ** Phase 6 - Salvage Free List
 2 files 2 blocks 9918 free
 *** FILE SYSTEM WAS MODIFIED ***
Copying file system back from tape

(number) blocks
File system copy from tape completed
# umount /dev/rdsk/c?d?s?<CR>
# init 2<CR>

Several messages will appear, ending with:

Console Login:
```

ctccpio — Copies File Archives to and from Cartridge Tape

General

The **ctccpio** command is used to copy file archives to and from a cartridge tape. The **ctccpio** command uses the streaming feature of the cartridge tape controller. Any cartridge tape that has data placed on it by the command **ctccpio** can not be appended to.

Command Format

The **ctccpio** command has the following format:

[file(s)] **| ctccpio -o** *[avVK]* **-T cartridge_tape_device**

ctccpio -i *[dmrtuvVf]* **-T cartridge_tape_device** *[pattern]*

The *file(s)* is the standard input; such as file, files, directories, or path names, that are piped (|) to the **ctccpio** command. The *file(s)* can use commands like **find** or **ls** (see *FIND* (1) and *LS* (1)). If the *file(s)* starts with (/), the directories will be placed under the root directory when extracted from the tape.

The **-o** argument (copy to) reads the standard input, to obtain the path names and status information, then copies this data onto the tape device specified by the argument **-T cartridge_tape_device**. Any data stored on the **cartridge_tape_device** will be overwritten.

The **-i** argument (copy from) extracts the data from the cartridge tape device, created by the **ctccpio -o** command, then copies the data to the current directory.

The *pattern* option is used to select a file or files you want extracted from the tape. The default for *pattern* is * (i.e., select all files).

The options for **ctccpio -o** are:

a Reset access times of input files after they have been copied.

v *Verbose* causes a list of file names to be displayed as they are copied out to tape.

V Prints a dot (.) for each file copied instead of the file name.

K Performs a verify pass on output to tape. It is recommended that the **K** option always be used with the **-o** option.

The options for **ctccpio -i** are:

d *Directories* are to be created as needed.

m Retains previous file *modification* time.

r Interactively *renames* files. The system will ask you to rename the files as they are written to hard disk. If you do not want the file written in, enter a *<CR>* and the system will skip that file.

t Prints a *table of contents* of the input. No files are created.

u Copies *unconditionally.*

v *Verbose* causes a list of file names to be printed. When used with the **t** option, the table of contents looks like the output of an **ls -l** command.

V Prints a dot (.) for each file copied instead of the file name.

f Copies all files from the cartridge tape except those in *pattern.*

Sample Commands

The following example shows how to archive all files and subdirectories, of the directory *usr/old*, out to the tape device **/dev/rSA/ctape1** with the file names displayed and verified:

```
# cd /<CR>
# find usr/old -print | ctccpio -ovKT /dev/rSA/ctape1<CR>
usr/old/junk/chapter1
usr/old/junk/chapter2
usr/old/junk/chapter3
usr/old/junk/chapter4
usr/old/bin/vikeys
usr/old/sample

Verify pass begins.
Verify pass completed.

12 blocks
#
```

The next example shows how to retrieve the data stored on the tape from the previous example, place it in another directory, rename the files, then see what was copied to the new directory with the **ls** command.

```
# mkdir lastexp<CR>
# cd lastexp<CR>
# ctccpio -irT /dev/rSA/ctape1<CR>
Rename <chapter2>
newchapter1<CR>
Rename <chapter4>
newchapter2<CR>
Rename <chapter6>
newchapter3<CR>
Rename <chapter8>
newchapter4<CR>

10 blocks
#ls<CR>
newchapter1
newchapter2
newchapter3
newchapter4
#
```

ctcfmt — Format Cartridge Tape

General

The **ctcfmt** command is used to format or reformat a cartridge tape. Formatting typically takes about ten minutes to complete. The cartridge tape can be formatted by using the device specified by the *rawdevice* parameter. Formatting a tape can be verified to make sure it is formatted properly.

Note: Once formatting has started, it can not be stopped.

Command Format

The **ctcfmt** command has the following format:

ctcfmt *[-v] [-p passct] -t rawdevice*

The *-v* option specifies that you want to verify the format. Verifying a tape takes an additional ten minutes, but it could save you time and loss of data if a problem were to occur. The decision of whether to use this option should be based on the importance of the data to be placed on the tape and whether you have had recent problems with cartridge tape defects. It is recommended that verification be made to make sure the tape is formatted properly and all blocks are readable.

The *-p passct* option specifies the maximum amount of tape passes allowed for the tape. This number is used to warn the user that the tape is nearing the end of its expected life and should be replaced to avoid the loss of data. If this option is not specified, a default value of 4000 is used. If the cartridge tape is going to be used for a file system, the pass count should be lowered to 3000.

The *-t rawdevice* argument specifies what device you are using to format a cartridge tape: **/dev/rSA/ctape1** for the first cartridge tape drive or **/dev/rSA/ctape2** for a second cartridge tape drive that may be added.

Sample Commands

The following example shows how to format a cartridge tape, with verification, and set the pass count to default (4000):

```
# ctcfmt -v -t /dev/rSA/ctape1<CR>
Insert tape, wait for re-tension pass to complete,
and press the <RETURN> key when ready [q]:<CR>
Format completed successfully.
#
```

The following example shows how to format a cartridge tape and set the pass count to 3500:

```
# ctcfmt -v -p 3500 -t /dev/rSA/ctape1<CR>
Insert tape, wait for re-tension pass to complete,
and press the <RETURN> key when ready [q]:<CR>
Format completed successfully.
#
```

ctcinfo — Display Information About a Cartridge Tape or Tape Drive

General

The **ctcinfo** command is used to display information about a cartridge tape drive and any cartridge tape that is inserted in the drive. The drive you want information about must be specified in the command line.

Command Format

The **ctcinfo** command has the following format:

ctcinfo *[options] rawdevice*

The *options* for **ctcinfo** are:

The *-v* option prints the volume table of contents for the tape in the specified tape drive.

The *-d* option prints the device type.

The *-t* option prints the current tape pass count.

The *-m* option prints the maximum tape count.

The *-u* option prints the current tape drive usage count.

The *-c* option prints the number of cylinders.

The *-x* option prints the number of tracks per cylinder.

The *-s* option prints the number of sectors per track.

The *-b* option prints the number of bytes per sector.

The *-a* option prints the total bytes on the device.

The *-B* option prints the total blocks on the device.

The **-r** option resets the tape drive usage count back to 0 and will stop issuing warning messages about using a dirty tape drive.

> ***Warning:*** ***The*** *-r* **option should only be used to inform the system that the tape drive has just been cleaned, when in fact it has been cleaned.**

Using the *-r* option when the tape drive has not been cleaned creates the possibility of losing data from any tapes that are run through the tape drive when it needs cleaning. If data is lost, it cannot be recovered unless you have retained another copy of it.

The *rawdevice* argument is the name of the tape drive you want information about. For example: */dev/rSA/ctape1* identifies cartridge tape drive 1.

Sample Commands

The following examples show how to print all the information about tape drive 1.

```
# ctcinfo -vdtmucxsbaB /dev/rSA/ctape1<CR>

Tape Drive Usage:
        1 hours 4 minutes since last cleaning
        18 hours 55 minutes until next cleaning

For device named /dev/rSA/ctape1:
VOLUME TABLE OF CONTENTS
VOLUME NAME: ctctape VERSION: 1 POSSIBLE NUMBER OF PARTITIONS: 16

PARTITION   TAG   FLAG   SECTOR START   SIZE IN SECTORS
    0         2     0          5272            8928
    1         3     1           126            5146
    2         4     0         14200           31341
    3         0     1             2           45539
    6         5     1             0           45541
    7         1     1             0             126
Device type is 6 Stream Floppy Tape
Tape Pass Count: 148
Maximum Allowed Tape Pass Count: 4000
Cylinders: 6
Tracks per cylinder: 245
Sectors per track: 31
Bytes per sector: 512
Total available bytes: 23315968
Total available blocks: 45539
#
```

finc — Incremental Backup

General

The **finc** command selectively copies the input file system to the output tape. This is a slow process. The tape must already be formatted by the command *ctcfmt* or the Simple Administration command *format*. Also, the tape must be labeled by the *labelit* command. The selection is controlled by the *selection-criteria* described below. Only one option should be used at a time.

Before executing **finc**, you should execute the **ff** command and save the output in a file as an index of the tape contents. Files on the tape can be recovered with the **frec** command.

> ***Note:*** You should mount the input file-system as read-only to insure an accurate backup, although acceptable results can be obtained in the read-write mode. The **finc** command copies the file system one block at a time.

The *labelit* command requires that you specify the desired cartridge tape drive in the raw mode and use the *-n* option to skip label checking.

Command Format

The **finc** command has the following format:

> **finc** *[selection-criteria] file-system raw-tape*

The *selection-criteria* for **finc** are:

The *-a n* option specifies to only copy the file, if the file has been accessed in *n* days.

The *-m n* option specifies to only copy the file, if the file has been modified in *n* days.

The *-c n* option specifies to only copy the file, if the i-node has been changed in *n* days.

The *-n file* option specifies to copy any file that has been modified more recently than the argument file.

The *file-system* argument specifies what file system you want to backup.

The *raw-tape* argument specifies where you want the *file-system* copied.

> ***Note:*** The *n* argument for options *a*, *m*, and *c* must be a whole decimal number, where *+n* means more than *n*, *-n* means less than *n*, and *n* means exactly *n*. A day is defined as a 24-hour period.

Sample Commands

To copy a file system to the output tape, the first command would be to enter **mount** to see what is available. Next run the **ff** command to list the i-nodes and directories of the special file you want copied. Then you can run the **finc** command.

The following examples show how to copy to a tape all files for file system **/mnt** that were modified in the last 48 hours. The **/mnt** file system is actually a file system on the integral floppy diskette that has been mounted as **/mnt**.

```
# mount<CR>
/ on /dev/dsk/c1d0s0 read/write on Fri July 19 07:43:55 1985
/usr on /dev/dsk/c1d0s2 read/write on Fri July 19 07:45:51 1985
/usr4 on /dev/dsk/c1d1s2 read/write on Fri July 19 08:36:35 1985
/mnt on /dev/SA/diskette1 read/write on Fri July 19 08:45:56 1985
# ff -m -1 /dev/SA/diskette1<CR>
ff: /dev/SA/diskette1: ff: 5 files selected
 .                    2
 ./awa                3
 ./awa/junk1          4
 ./awa/junk2          5
 .awa/bin             9
# labelit /dev/rmt/ctape1 tape14 tape14 -n<CR>
Skipping label check!
NEW fsname =tape14, NEW volname =tape14 - - DEL if wrong!!
# finc -m -1 /dev/rSA/diskette1 /dev/rmt/ctape1<CR>
finc: /dev/rSA/diskette1->/dev/rmt/ctape1: 5 files (7 blocks) selected
finc: /dev/rSA/diskette1->/dev/rmt/ctape1: datacopy done

#
```

frec — Recover Files from a Backup Tape

General

The **frec** command recovers files from the specified backup tape. The tape must have been written by **finc**. To recover a file, you need to give the i-numbers for that file. The output for each recovery request will be written into the file specified by *name*.

Command Format

The **frec** command has the following format:

frec *[-p path] [-f reqfile] raw-tape i-number:name...*

The *-p* option allows you to specify a default prefixing *path* different from your current working directory (./). This path will be prefixed to any *names* that are incomplete. For example: any *names* that do not begin with / or ./.

The *-f* option specifies a file that contains recovery requests. The format is:

i-number:newname

The file can only contain one request per line.

The *raw-tape* argument identifies the tape you want to recover data from.

The *i-number:name...* argument identifies the file you want to recover and where you want the file saved.

Sample Commands

The following example shows how to recover a file named *chap1* and place it in the current working directory. *Chap1* has an i-number of **9** and is located on a tape labeled csave.

```
# frec /dev/rSA/ctape1 9:chap1<CR>
frec: /dev/rSA/ctape1: Finc of csave (reel 1/1) made on Fri July 19 09:08:55 1985
#
```

The following example shows how to recover files *setup* and *Rlist.ctc* that have i-numbers of **7** and **6**, respectively. The files are recovered from a tape that was labeled *tape14* and placed in the *usr3* directory instead of the current working directory.

```
# frec -p /usr3 /dev/rSA/ctape1 7:setup 6:Rlistic.ctc<CR>
frec: /dev/rSA/ctape1: Finc of tape14 (reel 1/1) made on Fri July 19 14:33:00 1985
```

tar — Tape File Archiver

General

The **tar** command saves and restores individual files on a cartridge tape. Its actions are controlled by the *key* arguments.

Command Format

The **tar** command has the following format:

tar *[key] [device] [files]*

The *key* argument tells what actions you want the **tar** command to take. The *key* argument must have one, but no more than one, function letter. It must have at least one function modifier, but may have several, if desired. The **f** modifier must be used to specify the cartridge tape drive (**/dev/rSA/ctape1**).

The function letters are listed below:

r The named *files* are written at the end of the data stored on the tape. Data is not overwritten using the **r** function letter.

x The named *files* are extracted from the tape. If a directory name is given instead of a file name, the entire directory will be extracted. If no *files* argument is given, the entire contents of the tape is extracted. If several files with the same name are on the tape, the last one overwrites all earlier ones.

t The names of the specified files are listed each time they occur on the tape. If no *files* are specified, all the names on the tape are listed.

u The named *files* are added to the tape if they are not already there or if they have been modified since last written on the tape.

c Creates a new tape. Writing begins at the beginning of the tape instead of after the last file. The **c** function letter overwrites all data stored on the tape.

The function modifiers are listed below:

0,.,7 This modifier selects the drive on which the tape is mounted. The default is 1.

v Causes **tar** to type the name of each file it treats, preceded by the function letter. When used with the **t** function, **v** will give more information about the tape entries than just the name.

w Causes **tar** to print the action to be taken, followed by the name of the file, and then waits for the users confirmation. If a word beginning with " y" is given, the action is performed. Any other input means " no."

f Causes **tar** to use the next argument as the name of the device to use instead of **/dev/mt?**. If the name of the file is -, **tar** writes to the standard output, or reads from standard input, whichever is appropriate. Thus, **tar** can be used as the head or tail of a pipeline. The **tar** command can also be used to move hierarchies with the command:

 # cd fromdir; tar cf - . | (cd todir; tar xf -)<*CR*>

b Causes **tar** to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. The block size is determined automatically when reading tapes (function letters **x** and **t**).

l Tells **tar** to complain if it cannot resolve all the links to the files being dumped. If **l** is not specified, no error messages are printed.

m Tells **tar** not to restore the modification times. The modification time of the file will be the time of extraction.

The *files* argument specifies what files are to be dumped or restored. If a directory name is used instead of file names, all the files and subdirectories under that directory are specified.

Sample Commands

The following example will write out all the files and any subdirectories in directory **/mnt** that end with *.c*, then list each file as it is transferred, and start saving the files at the start of the tape:

```
# cd /mnt<CR>
# tar cvf /dev/rSA/ctape1 *.c<CR>
a chapter.c 8 blocks
a moreinfo.c 7 blocks
#
```

The next example shows how to write the file **addinfo.h** at the end of the data stored on the tape:

```
# tar rvf /dev/rSA/ctape1 addinfo.h<CR>
a addinfo.h 2 blocks
#
```

To see what has been stored on the tape, enter:

```
# tar tvf /dev/rSA/ctape1<CR>
Tar: blocksize = 20
rw------- 0/1  3590 July 19 11:18 1985 chapter.c
rw------- 0/1  3328 July 19 11:16 1985 moreinfo.c
rw------- 0/1   955 July 19 12:15 1985 addinfo.h

Note: The file addinfo.h was stored at the end
of the data on the tape.
#
```

The next example shows how to retrieve the files from the previous **tar** command and place them in another directory:

```
# cd /usr2/awa<CR>
# tar xvf /dev/rSA/ctape1<CR>
Tar: blocksize = 20
x chapter.c 8 blocks
x moreinfo.c 7 blocks
x addinfo.h 2 blocks
# ls -l<CR>
rw--------        0/1          955   July 19 12:15 1985 addinfo.h
rw-rw----        0/1         1093   July 19 10:14 1985 chapter1
rw-rw----        0/1         1006   July 19 01:28 1985 chapter2
rw-rw----        0/1         5554   July 19 03:06 1985 chapter3
rw--------        0/1         3590   July 19 11:18 1985 chapter.c
rw--------        0/1         3328   July 19 11:16 1985 moreinfo.c
Note: The three chapters (1, 2, and 3) already existed
in the directory awa.
```

Appendix

ERROR MESSAGES

This appendix contains the error codes that are created and displayed by the Cartridge Tape Utilities. The error codes are stored in the file */usr/include/sys/ct.h*. They are in addition to those found in */usr/include/sys/errno.h*. Information on the *errno.h* error codes can be found in the **intro(2)** manual page of the *AT&T 3B2 Computer Programmer Reference Manual.*

The following is an example of an error messages that will appear on the console terminal.

```
NOTICE: CTC Access Error: Consult the Error Message Section of the
3B2 Computer Cartridge Tape Utilities Guide (error num=215)
```

The following tables show the error **CODE** number and **NAME**, a **DESCRIPTION** of the error, and what **ACTION** must be taken to correct the error. The **ACTION** statement " See CTC RECOVERY PROCEDURE" is at the end of the tables.

CODE	NAME	DESCRIPTION — ACTION
200	**EUSRSPL**	Access to device is blocked because a special control function (ioctl - open) has exclusive access. This condition will occur if the tape unit is being used to do a backup/restore or format operation. — Wait for either of these operations to complete, then retry.
201	**ENOSGEN**	This condition occurs when the CTC board fails to complete its initialization and is left in an insane state. — See CTC RECOVERY PROCEDURE.
202	**EBRDDWN**	This condition occurs when it is detected that the CTC board is not operating properly and is then marked unavailable. — See CTC RECOVERY PROCEDURE.
203	**ENOCONF**	This condition occurs when an attempt is made to do an operation on the CTC sub-device (such as, cartridge tape drive or floppy disk drive) that is not connected to the CTC board. — Check hardware configuration for proper sub-devices.
204	**EFWCBAD**	This condition shows that a software routine failed to execute properly. — See CTC RECOVERY PROCEDURE.

CODE	NAME	DESCRIPTION — ACTION
205	**ENOTOPN**	This condition shows that read/write access from the CTC board to the sub-device is blocked. This condition will not occur under normal operating conditions. — See CTC RECOVERY PROCEDURE.
206	**EROPART**	Cartridge tape in sub-device is write protected or mounted read-only. — Remove write protection from media or mount in read/write mode.
207	**EPRTOVR**	This condition occurs when attempts are made to write to a cartridge tape that has run out of available space. — Retry on cartridge tape with adequate space. (See *ctcinfo.*)
208	**EBDVTOC**	The volume table of contents (vtoc) on the cartridge tape is not detected as sane. This may be a result of the cartridge tape needing to be re-tensioned. — Remove tape from drive and reinsert tape into drive and wait for re-tensioning pass to complete. Retry operation. If failure condition reoccurs, reformat cartridge tape. ***Warning: Reformatting tape will destroy data stored on the cartridge tape.***

CODE	NAME	DESCRIPTION — ACTION
209	EBDPSEC	The physical descriptor sector on the cartridge tape is not detected as sane. This may be a result of the cartridge tape needing to be re-tensioned. — Remove tape from drive, reinsert tape into drive, and wait for re-tensioning pass to complete. Retry operation. If failure condition reoccurs, reformat cartridge tape. ***Warning: Reformatting tape will destroy data stored on the cartridge tape.***
210	ENSLOPN	This condition occurs when a software routine fails to function properly. This condition will not occur under normal operating conditions. — See CTC RECOVERY PROCEDURE.
211	ENTSUSR	This condition occurs when an attempt is made to access the CTC board while a cartridge tape is being formatted. — Wait for format operation to complete then retry.
212	ETIMOUT	This condition occurs when the CTC board failed to complete a task in the time allotted. — See CTC RECOVERY PROCEDURE.
213	EHDWERR	This condition shows a CTC board hardware failure. — See CTC RECOVERY PROCEDURE.

CODE	NAME	DESCRIPTION — ACTION
214	ENOTRDY	The device was not ready for access. — Try again.
215	ERWERR	Attempt to read or write to cartridge tape has failed. — Try again. If repeated failures occur, re-tension cartridge tape by removing and reinserting cartridge tape into the tape drive. If condition persist, it may be because of a bad cartridge tape.
216	EWRTPRT	This condition occurs when an attempt is made to write to a write protected cartridge tape. — Remove write protection from cartridge tape.
217	EBDJSIZ	This condition occurs when a stream request exceeds the 15.5 Kilobyte limit. This condition will not occur under normal conditions. — See CTC RECOVERY PROCEDURE.
218	EBDOFLG	This condition occurs when the software detects a bad open flag and can not determine the read/write direction. This condition will not occur under normal conditions. — See CTC RECOVERY PROCEDURE.
219	ENOMED	This condition occurs when an attempt is made to do an operation on media that is not present in the sub-device. — Put cartridge tape in tape drive.

CTC RECOVERY PROCEDURE

In the normal course of using the Cartridge Tape Utilities you may encounter problems interfacing with the CTC or its sub-devices. These problems may be characterized by various error messages or simple no response at all. What follows is a general recovery procedure that may clear the source of your problem. This procedure requires the user to be logged in as **root** on the console terminal.

Repump the CTC firmware using the following command:

```
# /etc/pump /dev/rSA/ctape? /lib/pump/ctc<CR>
#
```

Note: The (**?**) *is the number of the CTC (such as, ctape1 for the first CTC).*

The system should respond with the root prompt. Check to see if error condition still exists. If the error condition continues to exist, remove and reinstall the Cartridge Tape Utilities software. If the error condition still exists, it may require the attention of your AT&T Service Representative or authorized dealer.

Replace this

page with the

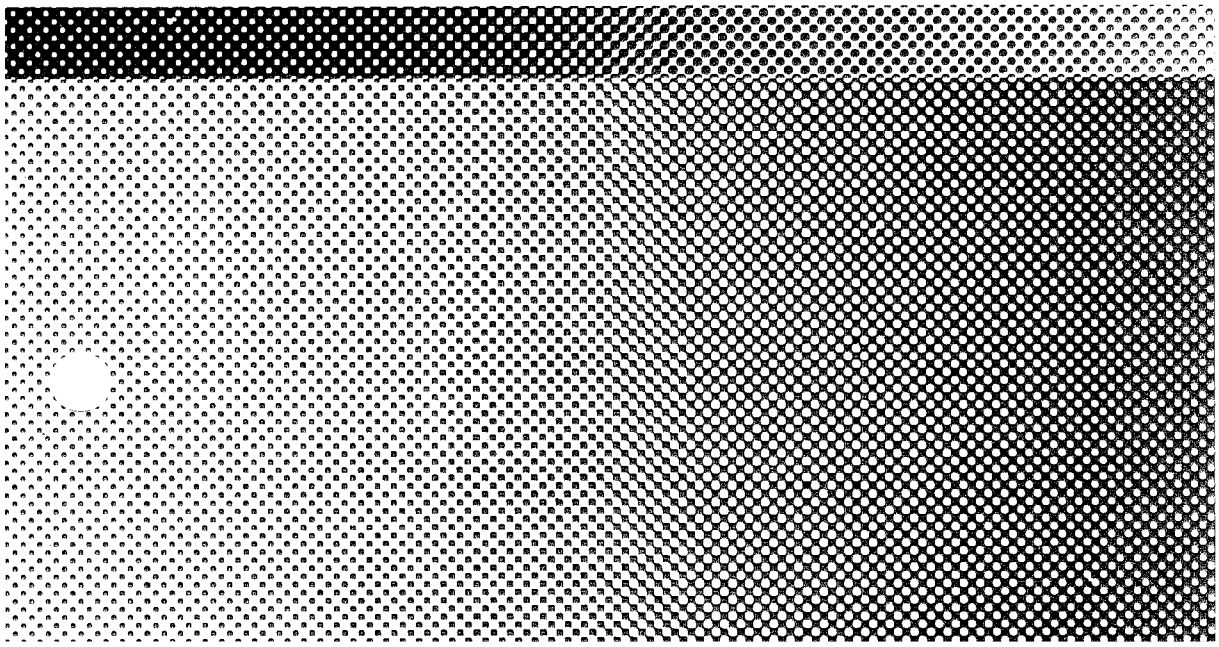
DIRECTORY AND FILE MANAGEMENT

tab separator.

AT&T

AT&T 3B2 Computer

UNIX™ System V Release 2.0 Directory and File Management Utilities Guide



CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

GENERAL

This guide describes command formats (syntax) and use of the Directory and File Management Utilities provided with your AT&T 3B2 Computer. The commands and procedures described in this guide can be used by anyone who has a need for enhanced file and directory manipulation on the 3B2 Computer.

Directory and File Management Utilities are software tools to aid in managing your directories and files. With one-step commands, you can skillfully do any of the following:

- Search directories and files
- Compare their contents
- Manipulate file data.

GUIDE ORGANIZATION

The remainder of this guide, Chapter 2 -- "COMMAND DESCRIPTIONS," describes the command formats (syntax) for each command in the Directory and File Management Utilities. The descriptions include the purpose of the command, a discussion of the command syntax and options, and examples of using each command.

Chapter 2

COMMAND DESCRIPTIONS

PAGE

Chapter 2

COMMAND DESCRIPTIONS

COMMAND SUMMARY

The Directory and File Management Utilities Package provided with the 3B2 Computer includes twenty-seven **UNIX*** System commands. These commands with a brief description are listed in Figure 2-1.

---

* Trademark of AT&T

Commands	Description
ar	Maintains groups of files that are part of a single archive file.
awk	Searches input lines for a matching pattern and performs specific actions.
bdiff	Finds what lines should be changed for two files to agree.
bfs	A read-only editor, similar to the **ed** editor, that is used to scan big files.
comm	Selects or rejects lines common to two sorted files.
csplit	Splits a file into parts as specified.
cut	Cuts out selected fields of data on each line of a file.
diff3	Compares three versions of a file.
dircmp	Compares two directories.
egrep	Searches a file for an *egrep* pattern, that is, a full regular expression.
fgrep	Searches a file for an *fgrep* pattern, that is, a fixed string.

Figure 2-1. Directory and File Management Commands (Sheet 1 of 3)

Commands	Description
file	Determines information about a file.
join	Joins two sorted files.
newform	Reads lines from a file or the standard input and reproduces those lines in a reformatted form on the standard output.
nl	Numbers lines in a file.
od	Outputs data in octal, decimal, ASCII, or hexadecimal formats.
pack	Stores file data in a compressed form.
paste	Merges the lines of two or more files in a side-by-side fashion.
pcat	Unpacks a compressed file for viewing only.
pg	Allows you to view a file, one page at a time on a video display terminal.
sdiff	Compares two files to produce a side-by-side listing of different lines.
split	Splits a file into parts of equal length.

Figure 2-1. Directory and File Management Commands (Sheet 2 of 3)

Commands	Description
sum	Calculates the checksum and blocks of a file.
tail	Copies a file or portion of a file to standard output.
tr	Filters a file by translating specified characters to other characters.
uniq	Reports file lines that are repeated.
unpack	Stores a compressed file in uncompressed form.

Figure 2-1. Directory and File Management Commands (Sheet 3 of 3)

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the commands. This format is as follows:

- **General:** The purpose of the command is defined. Any special or uncommon information about the command is also provided.
- **Command Format:** The basic command line format (syntax) is defined and the various arguments and options discussed.
- **Sample Command Use:** Example command line entries and system responses are provided to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example, **command** is in bold type.
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument.*
- Command options and fields that do not have to be supplied are enclosed in brackets ([]). For example: **command** [*optional arguments*].
- The pipe symbol (|) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example: **command** [*argument1* | *argument2*]

In the sample command discussions, user inputs and 3B2 Computer response examples are shown as follows:

```
This style of type is used to show system generated
responses displayed on your screen.
```

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: *<CR>* carriage return, *<CTRL d>* control d, *<ESC g>* escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

COMMANDS

"ar" — Archive and Library Maintainer for Portable Archives

General

The **ar** command is used to maintain groups of files that are part of a single archive file. The **ar** command is mainly used to create and update library files that are used by the link editor (**ld** command), but it can be used for any similar purpose. Refer to the *AT&T 3B2 Computer User Reference Manual* for information on the **ld** command. When the **ar** command creates an archive, the archive file is put into a format with headers that are portable across all computers that are compatible with your AT&T 3B2 Computer. These headers are placed at the beginning of each archive and have the following format:

```
#define ARMAG   "<ar>"
#define SARMAG  4


struct ar_hdr {/* archive header */
char  ar_magic[SARMAG]; /* magic number */
char  ar_name[16];/* archive name */
char  ar_date[4];/* date of last archive modification */
char  ar_syms[4];/* number of ar_sym entries */
};
```

The header is followed by an archive symbol table which is included in each archive that has common object files. This symbol table is automatically created by the **ar** command. The archive symbol table is used by the link editor to determine what archive members must be loaded during the link edit process.

There may be more than one archive symbol table. The number of symbol table entries is shown in the header under the *ar_syms* variable. Each archive symbol table has the following format:

```
struct  ar_sym {/* archive symbol table entry */
char  sym_name[8];/* symbol name, recognized by ld */
char  sym_ptr[4];/* archive position of symbol */
};
```

The archive symbol table is rebuilt each time the **ar** command is used to create or update the contents of an archive.

The archive symbol table is followed by the archive file members. A file member header precedes each file member. The file member header has the following format:

```
struct  arf_hdr {/* archive file member header */
char  arf_name[16];/* file member name */
char  arf_date[4];/* file member date */
char  arf_uid[4];/* file member user identification */
char  arf_gid[4];/* file member group identification */
char  arf_mode[4];/* file member mode */
char  arf_size[4];/* file member size */
};
```

All the information in the archive header, the archive symbol table, and the archive file member headers is stored in a machine (computer) independent fashion. Because of this, an archive file may be used on a computer that is compatible with the 3B2 Computer.

Command Format

The general format of the **ar** command is as follows:

ar *key* [*posname*] *afile name(s)*

The *key* argument uses the following options:

Note: Options **v**, **u**, **a**, **i**, **b**, **c**, **l**, or **s** must be used in combination with at least one of options **d**, **r**, **q**, **t**, **p**, **m**, or **x**.

d Delete *name(s)* from the archive file.

r Replace *name(s)* in the archive file using the following options:

- **u** Replace only those files that have modified dates later than the archive files.
- **a** Place new files after *posname*.
- **b** or **i** Place new files before *posname*.

  The *posname* argument must be specified. If you do not specify where to place new files, they will be placed at the end.

q Quickly append the named files to the end of the archive file. The positioning options under the **r** option will not work if used with this option. The **ar** command does not check to see if the added members are already in the archive.

t Print a table of contents of the archive file. If *name(s)* is specified, only that file(s) will be placed in the table of contents. If *name(s)* is not specified, all files in the archive will be placed in the table of contents.

p Print the contents of *name(s)* in the archive file.

m Move *name(s)* to the end of the archive. The positioning options under the **r** option can be used to place the file in a specific place.

x Extract *name(s). If name(s) is not specified, extract all files in the archive. This option will not alter the archive file.*

v Verbose. When making a new archive from an old archive and the constituent files, a file–by–file description of the process is given. When this option is used with the **t** option, a long listing of all information about the files is given. When this option is used with the **x** option, each file is preceded by its name.

c Create *afile*. Normally, the **ar** command will create *afile* when it needs to. The normal message that is produced when *afile* is created will not appear when this option is specified.

l Place temporary files in the local directory. If this option is not specified, temporary files will be placed in **/tmp**.

s Regenerate the archive symbol table even if the **ar** command is invoked with an option that will not change the archive contents. This option is useful to restore the archive symbol table after the **strip** command has been used on the archive.

The *posname* argument is used to determine the position of a file that is being moved: either before or after *posname*. *posname* is the name of a file in the archive. The *posname* argument must be used when using the positioning options listed under the **r** option of the *key* argument.

The *afile* argument is the name of the archive file.

The *name(s)* argument is the name of the constituent files in the archive file. Take caution not to list *name(s)* twice. If *name(s)* is mentioned twice, it may be put in the archive twice.

Sample Commands

The following command line entries and system responses show you how to create an archive. The **ls** command is used to show you the files that will be placed in the archive. The **ar** command used with the **q** option shows you how to create an archive named **archive1**.

```
$ ls<CR>
cars
cities
people
states
streets
$ ar q archive1 cars cities people states streets<CR>
ar: creating archive1
$
```

The following command line entry and system response show you how to print a table of contents of the archive that was created in the previous example:

```
$ ar t archive1<CR>
cars
cities
people
states
streets
$
```

The following command line entry and system response show you how to print the contents of a file in an archive:

```
$ ar p archive1 cars<CR>
Camero
Ferrari
Jaguar
Mustang
Porsche
$
```

The following command line entry and system response show you how to print a long listing of the table contents in an archive:

```
$ ar tv archive1<CR>
rw------- 5516/ 5500    38 July 19 13:29 1985 cars
rw------- 5516/ 5500    51 July 19 13:33 1985 cities
rw------- 5516/ 5500    29 July 19 13:27 1985 people
rw------- 5516/ 5500    51 July 19 13:48 1985 states
rw------- 5516/ 5500    60 July 19 14:16 1985 streets
$
```

"awk" — Pattern Scanning and Processing Language

General

The **awk** command is used to search lines of input data for defined patterns and to do specified actions on the lines or fields when a match is found. Lines that do not contain a matching pattern are ignored. Conversely, a line that contains more than one matching pattern can be operated on and output several times. One primary use of **awk** is for the generation of reports. Input data is processed to extract counts, sums, and other pertinent information. The processed information is then output in a specified format.

The **awk** command has its own programming language for defining patterns and their corresponding actions. The language is designed to simplify the task of information retrieval and text manipulation. Initially, the novice user will find **awk** difficult to use and understand. Your understanding of **awk** will increase as you spend more time using (and experimenting with) the capabilities provided by the command. Remember that the use of this command is task oriented; you must establish a purpose for using the command. For example, the **awk** command can be used to output tabular material in a different sequence of columns. Certain basic arithmetic functions can also be performed on designated fields.

Input Data Characteristics

Input data is normally taken from files of data. The variable called **FILENAME** contains the name of the current input data file. Input data is divided into records with each record ended by a record separator. The record separator is stored in a variable named **RS**. The default record separator is a new-line character. This means that by default, **awk** reads and processes data on a line–by–line basis. The record separator can be redefined by setting the variable **RS** equal to the desired character. When the **RS** is empty (undefined), a blank line is used as the record separator. In addition, the field separators are defined as blanks, tabs, and new-lines. The novice user should not arbitrarily redefine the **RS** variable. The number of the current record (current line) is stored in a variable named **NR**.

Each input record (line) is divided into fields. Each field, except the last field, is ended by a field separator. The field separator is stored in a variable named **FS**. The default field separator is white space: blanks or tabs. The field separator can be redefined by setting the variable **FS** equal to the desired character. The novice user should not arbitrarily redefine the **FS** variable. Each field is identified by an uncommon variable. The variables are **$1**, **$2**, **$3**, and etc. The first field is named **$1**. The entire record (line) is named **$0**. The number of fields in the current record is maintained in a variable named **NF**.

Command Language Format

The instructions that tell the **awk** command what to do to the input data can be specified directly as an argument to the command, or the instructions can be read from a file. In either case, these instructions constitute an **awk** program. An **awk** program is a sequence of statements that are in the following form for each statement. Note that an action must be enclosed in braces to distinguish it from a pattern. Additional command format information is provided later in this description. The general form of each statement is as follows:

pattern { action }

The **awk** command operates on one record (line) of input data at a time. Each line of input data is tested against each of the command lines defined in an **awk** program. When a pattern match is found, the associated action is executed. When a command line defines a pattern without an associated action, then the input data record is output when a match is found. When a command line defines an action without an associated pattern, then the action is performed for all input lines (records). After all program command lines have been tested against the current record (line), the next record (line) is read and the process repeated until all records are read.

Patterns

A pattern is an expression that determines whether the associated action is to be performed. When a command line defines a pattern without an associated action, then the input data record is output when a match is found. A variety of expressions can be used as patterns. Patterns can be regular expressions as used with **ed** or **grep**, relational expressions, or special expressions. Combinations of these types of expressions can be used to define a pattern by using Boolean operators to connect each expression. The Boolean operators are OR (**||**), AND (**&&**), and NOT (**!**). Conventional arithmetic operators are also provided. The arithmetic operators are add (+), subtract (-), multiply (*), divide (/), and modulus (**%**). Also, included are the increment (++) and decrement (--) operators.

Regular Expressions: Regular expressions, in their simplest form, are context search patterns in the form used by the **ed** or **grep** commands. The **awk** language adds operators to the regular expression to specify whether the corresponding action is to be executed if the pattern matches (~) or does not match (!~). For example, the following pattern outputs all records in which the first field does not contain the word "operates".

```
$1 !~ /operates/
```

Relational Expressions: Relationships are statements that express conditions such as greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (!=). For example, the following relational expression selects lines that begin with any letter that is equal to or greater than the letter **s**: All lines beginning with the letters **s** through **z** are matched by this pattern.

```
$1 >= "s"
```

Special Expressions: Two special expressions named **BEGIN** and **END** are provided. The **BEGIN** expression defines a special pattern that matches the beginning of the input data before the first record is read. The **END** expression defines a special pattern that matches the end of the input data after the last record has been processed. These special expressions

provide the means to establish initial and post-processing conditions. When **BEGIN** is used, it must be the first pattern in the **awk** sequence of commands (program). The **END** must be the last pattern in the program. The following example shows a typical structure. In this example, the field separator (**FS**) is set to a colon at the beginning of the program; the number of each record (**NR**) is output at the end of the program.

```
BEGIN { FS = ":" }

...body of program...

END { print NR }
```

Two patterns, separated by a comma, can be used to control the execution of an action over a range of records. The action is executed for each record, starting with the match of the first pattern and continuing until the match of the second pattern, inclusive of the record containing the second pattern. The following example shows the general construction for a pattern range that controls an action:

```
/pattern1/, /pattern2/ { action }
```

Actions

An action specifies a function that is to be executed. When an action is associated with a pattern, then the action is executed only when the current record (line) matches the associated pattern. An action that does not have a corresponding pattern is executed for each input record (line).

The various action terms recognized by the **awk** command are as follows:

```
exp
index(s1,s2)
int
length
log
print
printf(" f" ,e1,e2,...)
split(s,array," sep" )
sprintf(" f" ,e1,e2,...)
sqrt
substr(s,m,n)
```

Each of these action terms are described in the following paragraphs.

exp: The **exp** function computes *e* (2.7182818) raised to the *x* power, where *x* is a field argument. For example, when combined with the **print** function, the following statement outputs the value of e^x for each record, where *x* is the value of third field:

```
{ print exp($3) }
```

index(s1,s2): The index function is used to obtain the starting position of a string (**s2**) within another string (**s1**). A zero is returned when **s2** does not exist within **s1**. When combined with the print function, the following statement outputs the starting character position of a string **Smith** within the first field of each record:

```
{ print index($1," Smith" ) }
```

int: The integer function converts irrational numbers to rational numbers for a specified field; numbers expressed to some fractional quantity are converted to whole numbers. The function DOES NOT round off numbers. Fractional quantities are deleted. For example, the number 3.984 would be converted to the number 3. When combined with the **print** function, the following statement outputs the fifth field of each record expressed as whole numbers (integers):

```
{ print int($5) }
```

length: The **length** function computes the length of a string of characters. When combined with the **print** function, the following statement outputs the length of each record (line):

```
{ print length($0) }
```

The following statement outputs the length of each record, followed by the record:

```
{ print length($0), $0 }
```

The **length** function can also be used to output records (lines) that are within a specified length range. For example, the following statement outputs all records that are less than 20 and greater than 10 characters in length. The || is the Boolean OR function.

```
{ length > 10 || length < 20 }
```

The following statement outputs all records that are outside the 10 to 20 character range:

```
{ length < 10 || length > 20 }
```

log: The **log** function computes logarithms to the base *e*. When combined with the **print** function, the following statement outputs the logarithm of fifth field for each record:

```
{ print log($5) }
```

print: The simplest action provided is the **print** function. For example, the following statement outputs the first two fields of each record in reverse order (field 2 followed by field 1):

```
{ print $2, $1 }
```

The output of a **print** statement can be directed to a file. For example, the following statement outputs the first field to **file1** and the second field to **file3** for each record:

```
{ print $1 >> " file1" ; print $2 >> " file3" }
```

The output of a **print** statement can be directed to another program. For example, the following statement outputs the fifth field of each record to the **sort** command: The output of the **sort** command is directed to a file named **file5**.

```
{ print $5 | " sort -o file5" }
```

printf(" f" ,e1,e2,...): The **printf** converts, formats, and prints its arguments on the standard output. This function is exactly like the C Language **printf** function. The **f** argument specifies the format. The expressions to be formatted are specified by the **e** arguments. For example, the following statement prints the third field of each record as a floating point number that is ten digits wide with two decimal places. The fifth field is printed as a ten-digit long decimal number, followed by a new-line (\n).

```
{ printf(" %8.2f %10ld\n" , $3, $5) }
```

Remember that with this print function, you must specify the output field separators; no field separators are automatically output.

split(s,array," sep"): The split function is used to automatically divide a string into fields. The **sep** argument, if provided, defines the field separator. The **FS** variable is used as the field separator if the **sep** argument is omitted from the statement. The string **s** is divided into fields defined by the **array** argument. For example, the following statement divides the second field of a record into elements of an array named **z** based on the dash as the field separator. Each element of the array (field) is individually identified. The first element is named **z[1]**; the fifth element is **z[5]**. In the following example, the **print** function is used to output the fifth field of the array **z**:

```
{split($2,z," -" );print z[5]}
```

sprintf(" f" ,e1,e2,...): The string print function is used to place formatted output in a character array pointed to by a single character name. The **sprintf** converts, formats, and outputs its arguments to a string name. For example, the following statement sets **x** equal to the formatted result of the third and fifth field. The third field of each record is formatted as a floating point number that is ten digits wide with two decimal places.

The fifth field is formatted as a ten-digit long decimal number followed by a new-line (\n). Thus, the variable **x** is set to the string produced by formatting the values of fields $3 and $5. The variable **x** can be used in other statements to express these values as a formatted expression.

```
{ x = sprintf(" %8.2f %10ld\n" , $3, $5) }
```

sqrt: The square root function computes the second root of a specified item. When combined with the **print** function, the following statement outputs the square root of the first field for all records:

```
{print sqrt($1)}
```

substr(s,m,n): The substring function is used to obtain a specified part of a string. The **m** argument defines the starting character position of the substring. The beginning of the string is character position number 1. The **n** argument defines the number of characters to be included in the substring. If the **n** argument is omitted, the substring is defined from the beginning position **m** to the end of the string **s**. When combined with the **print** function, the following statement outputs part of the third field for all records. The portion of the field that is selected is the fifth character position to the end of the field.

```
{ print substr($3,5) }
```

Assigning Variables

Variables are assigned as either floating point numbers or as string values. Unlike C Language, variables DO NOT have to be declared at the beginning of a program. For example, the following sets **x** equal to the string **word**:

```
x = " word"
```

The following sets **x** equal to the number **100**:

```
x = " 100"
```

Arrays

Arrays are used to hold fields of data that are called elements. Each element or field in the array is identified by its sequential position. Array elements can also be named by nonnumeric values, which provides an associative type of memory. For example, the following **awk** program counts the number of times the patterns **apple** and **orange** occur. The results are stored in an array named **z**. The accumulated counts for each of these patterns is output at the end of the program. Note that the ++ operator increments the count by one each time it is called.

```
/apple/ {z[" apple" ]++}
/orange/ {z[" orange" ]++}
END {print z[" apple" ], z[" orange" ]}
```

The following example does the same function as the previous program. The only differences are that numeric designators are used for the elements of the array as opposed to associative names, and that the names are output to identify the counts.

```
/apple/ {z[1]++}
/orange/ {z[2]++}
END {print " apple = " z[1] "  orange = " z[2]}
```

Control Flow Statements

The **awk** programming language provides the following basic control flow statements: **if-else**, **while**, and **for**. Also provided are the following control statements: **break**, **continue**, and **next**. The **break** statement causes an immediate exit from an enclosing **while** or **for** construction. The **continue** statement causes the next cycle of a loop to begin. The **next** statement causes **awk** to immediately skip to the next record and begin processing.

Control flow constructions are exactly like that of the C Programming Language. For example, the following construction outputs all fields on a separate line using a **while** statement:

```
i=1
while (i<=NF) {
    print $i
    ++i
}
```

The following example construction outputs all fields on a separate line using a **for** control statement:

```
for (i=1;i<=NF;++i)
print $i
```

Commenting Programs

In general, the **awk** programs that you write are done so in files. Only the simplest of **awk** functions are done by specifying patterns and actions directly to **awk** as arguments. When writing a program, the importance of adequately providing comments that show what you are doing at various stages in the program cannot be over emphasized.

Comments are entered by preceding the comment with a pound symbol (#). The comment ends with the end of the line. When more than one line is used for a comment, each comment line must begin with the pound symbol. Remember that if your erase character is the pound symbol, you must precede the pound with a backslash (\) to enter the symbol. This is referred to as escaping the special meaning of the character. The following shows how you enter comments into a program:

```
print x, y # Print results
# This is a continued or new comment line.
```

Command Format

The general format of the **awk** command is as follows:

awk [**-f** *source* ¦ *'cmds'*] [*parameters*] [*file*]

The instructions that tell the **awk** command what to do can be directly expressed to the command as arguments or they can be entered into a file that is then read by the command. When instructions are expressed as arguments, they are in the form *'cmds'*. Note that instructions that are expressed as arguments must be enclosed in single quotes. When instructions are placed in a file, the file is specified to the **awk** command in the form **-f** *file*. Note that the file name can be expressed as a full path name.

The *parameters* argument is used to identify the value of variables. The argument is: **x**=... **y**=..., and so on. Note that a space is used to separate each variable statement.

The *files* argument identifies the input data file. The file name can be expressed as a complete path name.

Sample Command Use

The following examples are based on a file named **list**. This file contains a list of names, addresses, and phone numbers as follows. Note that the format of each line of this file is name(tab)address(tab)phone.

```
$ cat list<CR>
Nancy  1080 Route 3, Farmington, NC  27015    919-736-2437
John   4589 Breckenridge, Clemmons, NC  27012  919-828-7512
Sam    2700 Route 67, Winston-Salem, NC  27106 919-234-1940
doctor 4100 First St, Winston-Salem, NC  27102 919-727-1111
$
```

The first example uses the **awk** command to output selected names and addresses from the **list** file in a format suitable for mailing labels. The program is in a file called **labelsprgm** and follows:

```
$ cat labelsprgm<CR>
BEGIN{FS=" \t" }
$1~/Nancy/||$1~/Sam/{split($2,x," ," )
printf(" %s\n%s\n%s\n%s\n\n" ,$1,x[1],x[2],x[3])}
$
```

The second example uses the **awk** command to output selected names and phone numbers from the **list** file. The program is in a file called **numbers** and follows:

```
$ cat numbers<CR>
BEGIN{FS=" \t" }
$1~/Nancy/||$1~/Sam/{printf(" %s\t%s\n" ,$1,$3)}
$
```

The following command line entry and system responses show the use of the **labelsprgm**:

```
$ awk -f labelsprgm list<CR>
Nancy
1080 Route 3
Farmington
NC  27015

Sam
2700 Route 67
Winston-Salem
NC  27106

$
```

The following command line entry and system responses show the use of the **numbers** program:

```
$ awk -f numbers list<CR>
Nancy   919-736-2437
Sam     919-234-1940
$
```

The following example is based on a file named **gaintbl**. This file is a table containing columns of measured data, input and output voltage (**Vi** and **Vo**), and blank columns for new data.

```
$ cat gaintbl<CR>
Vi Vo Vo/Vi Log+Av Av(dB)
-- -- ----- ------ ------
 2  5
 8 15
10 18
$
```

In this example, the **awk** command performs several arithmetic operations on the data in **gaintbl**. The measured data and the resulting new data are output in a table format. Since the field separators of **gaintbl** are spaces, a **BEGIN** statement is not required. The program is in a file called **calcprgm** and follows:

```
$ cat calcprgm<CR>
#
#       PRINT TABLE HEADING
$1~/Vi/#$1~/--/{
printf " %s\t%s\t%s\t%s\t%s\n" ,$1,$2,$3,$4,$5}
#
#     CALCULATE & PRINT DATA
$1!~/Vi/&&$1!~/--/{$3=($2/$1);$4=log($3);$5=20*$4;
printf " %2ld\t%2ld\t%3.3f\t%3.4f\t%3.3f\n" ,\
$1,$2,$3,$4,$5}
$
```

The following command line entry and system responses show the use of the **calcprgm**:

```
$ awk -f calcprgm gaintbl<CR>
Vi      Vo      Vo/Vi   Log+Av  Av(dB)
--      --      -----   ------  ------
 2       5      2.500   0.9163  18.326
 8      15      1.875   0.6286  12.572
10      18      1.800   0.5878  11.756
$
```

"bdiff" — Big Differential File Comparator

General

The **bdiff** command operates much like the **diff** command covered later in this chapter. It compares two files and outputs instructions that tell what must be changed to bring the two files into agreement. The purpose of **bdiff** is to compare files that are too large for **diff** to process. It splits the files being compared into segments and performs **diff** on each segment. The output is identical to that of **diff**, except the line numbers are adjusted to account for the previous segments.

Command Format

The general format for the **bdiff** command is as follows:

bdiff *file1 file2* [**n**] [**-s**]

If no options are specified, **bdiff** ignores the lines that are common to the beginning of both files and splits the remainder of each file into 3500-line segments. The **diff** command is then performed automatically on the segments. The output will be the lines of the first named file followed by the lines of the second named file that are different. The less-than symbol (<) precedes the lines of the first named file. The greater-than symbol (>) precedes the lines of the second named file.

The optional third argument, **n**, is used to specify the number of lines to be contained in the file segments. If **n** is given in numeric form, the files are split into **n**-line segments instead of the 3500-line default count. These **n**–line segments are useful where the 3500-line segments are still too large for **diff** to handle.

The **-s** option will suppress any diagnostics that would be displayed by **bdiff**. However, any diagnostics output by **diff** will still be displayed.

If both the **n** and **-s** options are specified, they must be specified in the order shown in the command format, that is, the numeric value for **n** is entered before the **-s** option.

If a dash (—) is entered instead of *file1* or *file2*, the file that is named will be compared to what is input from the terminal. The input is entered exactly as it is to be compared to the named file. A "control d" is used to show the end of the input.

Sample Command Use

The following command line and system response shows how to output the differences between **chapter1.1** and **chapter1.2**:

```
$ bdiff chapter1.1 chapter1.2<CR>
23c23
< designed for Release 1.1 of the software.
---
> designed for Release 1.2 of the software.
104c104
< with update considerations for Release 1.2 compatibility.
---
> with update considerations for Release 1.3 compatibility.
$
```

The following command line and response show how to split the files into 1000-line segments:

```
$ bdiff file1 file2 1000<CR>
2124c2124
< is a sample of the command.
---
> is an example of how to use the command.
$
```

Note: The difference between the two files was found in the second segment, but **bdiff** adjusts the line count to specify the correct line number for the original file.

The following example shows how to format **chap1** and compare the formatted file to **OLDchap1**. The format program for this example is called **form**.

```
$ form chap1 ! bdiff OLDchap1—<CR>
72c72
< will not be displayed on the screen.
---
> will be displayed on the screen.
$
```

Note: In this example, **OLDchap1** lines will be displayed first. The order may be reversed if the filename and — are reversed.

"bfs" — Big File Scanner Editor

General

> ***Note:*** This command does not follow the same format as the other commands in this Utilities Guide.

This part of the chapter describes the **bfs** (big file scanner) editor used on the 3B2 Computer. The bfs editor is similar to the *ed* editor, except that it is read-only. Since bfs cannot be used to change a file, commands such as: insert, append, substitute, delete, and move will not execute.

Bfs works with the file instead of a copy placed in a buffer (temporary memory). It is normally used for processing files that are too large for conventional editing. Bfs can access files up to 1024 kilobytes (maximum size) and 32,000 lines--with up to 255 characters per line.

The bfs editor is useful for identifying sections of a large file where the commands *csplit* or *split* can be used to divide it into more manageable pieces for editing. The *csplit* and *split* commands are included in this Utilities Guide.

This editor description assumes that you know how to log in to the 3B2 Computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

Current Line Definition

Throughout this chapter, the term "current line" is used to identify what line in the file you are currently on. To display the current line, enter:

p<*CR*>

Any commands you execute will use this line as a reference point.

Getting Started

The **bfs** editor can only be used on existing files. To create a new file by inputting data directly, you must use another editor. However, **bfs** can be used to create a new file if you need to copy part of an existing file into another file. Commands to do this are discussed later in this chapter.

To execute the **bfs** editor, you must first be logged in to the 3B2 Computer. Once you are logged in, the UNIX System prompt (**$** or #) should be displayed. You are now ready to begin working with the **bfs** editor.

Accessing a File

To scan a file using the **bfs** editor, you will need to type **bfs** followed by a space, and then the name of the file you wish to scan. Execute the command by entering a carriage return <*CR*>. For example:

$ **bfs filename**<*CR*>

will execute the **bfs** editor against the file "filename". If you entered the command correctly, the response will be a number that represents the number of characters in the edited file.

If you do not enter the command correctly, you will receive a usage message indicating an incorrect syntax was used. When this occurs, verify the name of the file; make sure you are in the right directory; and reenter the command correctly.

If you do not want the editor to display the size of the file, enter:

$ **bfs - filename**<*CR*>

where **filename** is the name of the file you want to access. The 3B2 Computer will not display a response.

Once you are in the **bfs** editor, you may begin scanning the file. To begin displaying lines in the file, you must enter a line number (for example: **1**) followed by a carriage return. The editor will use the line as a reference point. After you display a line, any of the commands described in this chapter can be used.

Displaying a Prompt

The **bfs** editor does not display a prompt unless you request one. At times, absence of a prompt can be confusing. Most users find it easier to use the editor with the prompt (*) displayed. To display the prompt, enter:

P<*CR*>

Receiving Error Messages

When the prompt is not requested, any editor error message displayed will simply be "**?**". To receive self-explanatory error messages, the prompt must be turned on. See the previous discussion on "Displaying a Prompt".

Getting File Information

There are two editor commands that can be used to obtain information about the file you are editing. To display how many lines are in the file, enter:

=<*CR*>

To display the name of the file, enter:

f<*CR*>

Quitting the Editor

Because the **bfs** editor is read-only, it will allow you to quit without warning you to write the file. To quit the editor, enter:

q<*CR*>

The 3B2 Computer will return you to the UNIX System.

Displaying Lines in the File

As previously discussed in "Current Line Definition", the current line is always displayed whenever you move through the file. However, you can display more than one line by using the print command (**p**). An example of the print command would be:

1,10p<*CR*>

that would display lines 1 through 10. This form of the print command can be used to display as much of the file as you wish. The end of file symbol (**$**) can also be used with the print command to display lines. For example:

250,$p<*CR*>

will display lines 250 through the end of the file. As you become familiar with the editor, you will find that the lines will be displayed even if you leave the **p** off the end of the command. For example:

250,$<*CR*>

will display from line 250 to the end of the file.

The print command can also be used with other commands, such as searches and marks. These uses of the print command are discussed in the explanation of the individual commands.

Basic Movement Commands

As previously discussed, one way to move through a file is to use the carriage return. You can also use the + and - commands with the carriage return to move you forward or backward through the file. With these commands, you can move to an adjacent line in the file.

To move in larger steps, you can use numbers with the + and - commands. For example:

+15<*CR*>

will move you forward in the file 15 lines, and display the current line. Likewise,

-15<*CR*>

will move you backward 15 lines and then display the current line.

Each line in the file has a line number associated with it, although it is not displayed. The **bfs** editor allows you to move across large areas of the file by just entering a line number followed by a carriage return. For example:

375<*CR*>

will make 375 the current line and display the line.

Another movement command that is useful on large files is the **$**. If you enter:

$<*CR*>

bfs will move you to the last line of the file and display the line.

Forward and Backward Searches

If you do not know a specific line number, but you do know an exact pattern of characters on a line in the file, the quickest way to locate that line is with a search. **The pattern must be on one line.** There are several types of searches. The type you should use depends on your specific application.

Searches With Wrap-Around

When entering a command, the **bfs** editor interprets the character "/" as meaning "search for this pattern". The search command "/" searches from the current line forward through the file for the first occurrence of the pattern. When the end of the file is reached, the search will wrap-around to the beginning of the file and continue searching until the pattern is found or it reaches the line where the search started. If the pattern is found, the line will be displayed and will become the current line. An example of a forward search command would be:

/learning the bfs editor/<*CR*>

that will search for the first occurrence of a line containing the pattern "learning the bfs editor", make it the current line, and display the line. If the pattern is not found, the message:

" learning the bfs editor not found"

will be displayed. This means the pattern you searched for is not on one line in the file, and the current line does not change. Check to see if you entered the command correctly, or if it included any characters with a special meaning (see "Special Search Characters").

The character "**?**" also executes a search when used in a command. It works the same as the "/" search character, except that it searches backward through the file from the current line. This search will wrap-around to the end of the file and continue searching until the pattern is found or it reaches the line where the search started. An example of a backward search command would be:

?learning the bfs editor?<*CR*>

Searches Without Wrap-Around

Another set of search commands can be used that do not wrap-around the end of the file. These commands are similar to the wrap-around searches, except that they stop at the beginning or end of the file. The forward search command ">" searches for the first occurrence of the specified pattern until it reaches the end of the file. An example of this type of forward search command would be:

>**learning the bfs editor**><*CR*>

If the pattern is found, the line will be displayed and will become the current line. If the pattern is not found, the message:

" learning the bfs editor not found"

will be displayed and the current line will not change.

The character "<" also executes a search. It works the same as the ">" search, except that it searches from your current position backwards until it reaches the beginning of the file. An example of this type of backward search command would be:

<**learning the bfs editor**<<*CR*>

Repeating a Search

Often when searching for a pattern, the first occurrence is not the one you were actually looking for. You could repeat the entire search command, but there is a much easier way. The editor remembers the last search pattern entered. If you enter the command:

//<*CR*>

a forward search will look for the remembered pattern. The commands **??**, >>, and << will also repeat searches. The type of search repeated depends on the command used. The repeated search does not have to be the same type as the original search.

Global Searches

The **bfs** editor also allows you to do global searches on the file. A global search is used to find all the occurrences of a specified pattern in a file. This type of search is useful when scanning for a pattern that occurs in several places. The two types of global searches that can be executed use the **g** and **v** commands.

The global search that uses the **g** command locates all the lines that contain a specified pattern. An example would be:

g/sample pattern/p<*CR*>

that will search for and display all lines containing the words "sample pattern". The current line will be the last line displayed.

The global search that uses the **v** command locates all lines that **do not** contain a specified pattern. An example would be:

v/sample pattern/p<*CR*>

that will search for and display all lines that **do not** contain the words "sample pattern." The current line will be the last line displayed.

Special Search Characters

Several characters have special meaning when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters are: ".", "*", "\", "[]", "$", and "^".

- The period matches any single character except the newline (carriage return) character. For example:

  /bfs edit.r/<*CR*>

  will search for a pattern such as "bfs editor", "bfs editxr", or with any other character on a line between "bfs edit", and "r".

* The asterisk matches any string of characters except the first ., \, [, or ~ in that group. For example:

/the x* editor/<*CR*>

will search for a pattern such as "the xxx editor", "the xxxxxx editor", or a pattern with any amount of "x" characters on a line between "the" and "editor".

\ The backslash is used to nullify the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example:

/This is a \$/<*CR*>

will search for the pattern "This is a $", instead of interpreting the "$" as meaning "at end of line".

[] Brackets are used to enclose a variable string. For example:

/Search for file[23]/<*CR*>

will search for the patterns "Search for file2"or "Search for file3" and stop at the first occurrence of either pattern.

$ The dollar sign is interpreted by the editor to mean "end of the line". It is used to identify patterns that occur at the end of a line. For example:

/last character$/<*CR*>

will search for the next occurrence of a line ending in "last character", and make it the current line.

^ The circumflex (caret) works like "**$**" except it looks for the pattern at the beginning of the line. For example:

/^First character/<*CR*>

will search for the next occurrence of the pattern "First character" at the beginning of a line and makes it the current line.

To search for the characters ., *, \, [,], $, or ^, you must precede the characters with a backslash. This will nullify the characters special meaning.

Marking Lines

The **bfs** editor gives you the ability to set marks in the file. Marks are useful when you are planning on moving around in the file and you want to set some reference points. They can save you from having to search for the same address several times.

Marks are set by moving to the line where you want the mark set and using the **k** command. The mark must be a single, lower case letter. For example, if you wanted to identify a line with the mark "a", you would move to that line and enter the command:

ka<*CR*>

To move to that marked line from anywhere in the file, enter the command:

'a<*CR*>

The marked line will become the current line. To set another mark, repeat the **k** command using a different letter.

To change an existing mark, move to the line where you want the mark and use the **k** command with the existing mark. The new position will replace the previous one.

The **n** command will display a list of the active marks. For example, if the active marks in a file were a, b, and c, you could display them by entering:

n<*CR*>

The system response would be:

a
b
c

Notice that only the marks are displayed and not the lines.

Note: All marks are removed if you quit the editor using the **q** command. However, if you leave the editor by using the **e** command and then return to the file with the **e** command, all marks are saved. See " Changing Files While Using the " bfs " Editor."

Writing to Another File

The **bfs** editor allows you to copy all or part of the file you are editing to another file. To copy the whole file to another file, use the **w** command and the name of the file you want to create. For example:

w newfile<*CR*>

will create a copy of the file you are editing and name it "newfile". The number of characters in the new file will be displayed to show that the new file was created.

Caution: Be careful when naming the new file. If you use an existing filename, the text in that file will be overwritten by the new text.

If you only want to write part of the file, you must specify the beginning and ending lines you want to write. For example:

50,220w newfile<*CR*>

will create a file named "newfile" which will contain lines 50 through 220 of the file you are editing.

Changing Files While Using the "bfs" Editor

When using the **bfs** editor, only one file can be scanned at a time. However, the "**e**" command allows you to change files without quitting the editor. For example, if you are scanning *file1* with the **bfs** editor and want to change to *file2*, you would use the command:

e file2<*CR*>

This will cause the editor to leave *file1* and enter *file2*. To reenter *file1*, you would need to use the **e** command again. Using the quit (**q**) command will cause you to leave the file you are now in and return you to the UNIX System.

Issuing UNIX System Commands

The **bfs** editor allows you to execute a single UNIX System command by entering a command of the form:

!cmd<*CR*>

where "cmd" represents the command you want to execute. The system will then execute the command. When finished, **bfs** will display an **!** and then return you to the current line in the file. You can then continue editing or issue another **!** command.

If you need to execute more than one UNIX System command, enter the command:

!sh<*CR*>

When you are finished executing UNIX System commands, enter a **control-d** (depress and hold the CONTROL "CTRL" key and simultaneously depress the "d" key). The editor will display an **!** and return to the current line in the file.

High-Level "bfs" Commands

A "command file" is an executable file that contains editor commands. Command files may be set up and run against other files with the **bfs** editor. When executing command files, the output is directed to another file.

"comm" — Select or Reject Common Lines

General

The **comm** command compares two files and produces an output showing the differences and similarities between them. The contents of the two files should be in alphabetical order, that is, in order according to the ASCII collating sequence. The output is formatted into three columns. The first column lists those lines found only in the first named file, the second column lists those lines found only in the second named file, and the third column lists those lines that are common to both files.

Command Format

The general format for the **comm** command is as follows:

comm [— [**123**]] *file1 file2*

The — [**123**] option suppresses the column corresponding to the number specified. For example, if —**1** is specified, the first column of the output is not displayed. Thus, only those lines uncommon to the second named file and those lines common to both named files are displayed.

If a dash (—) is entered in place of a file name, the standard input from the terminal is read. The **comm** command compares the input with the file and produces the three-column output as before.

Sample Command Use

The examples provided are based on the contents of two files named **listA** and **listB**. The contents of these two files are as follows:

listA:	**birds**	**listB:**	**birds**
	cats		**cats**
	dogs		**horses**
	horses		**mice**
	mice		**mules**
	snakes		**pigs**

The following command line and system response shows how to compare the two lists and receive all columns of the output:

```
$ comm listA listB<CR>
                birds
                cats
dogs
                horses
                mice
        mules
        pigs
snakes
$
```

The following command line and system response shows how to compare the two lists and output only those lines that are common to both files:

```
$ comm -12 listA listB<CR>
birds
cats
horses
mice
$
```

The following example shows how to alphabetize a file named **list** and compare it to **listB** with the same command line. The file **list** is:

birds
mice
dogs
cats
pigs

The command line uses **sort** to alphabetize the file **list**.

```
$ sort list | comm - listB<CR>
                birds
                cats
dogs
        horses
                mice
        mules
                pigs
$
```

"csplit" — Context Split

General

The **csplit** command splits a file into sections using input arguments as the boundaries of the sections. The sections are suffixed with a number starting with 00 and may go up to 99. The first section (00) will contain from the beginning of the file up to, but not including, the line defined by the first argument. The second section (01) will contain the line defined by the first argument up to, but not including, the line defined by the second argument. The last section will contain the line defined by the last argument through the end of the original file. The original file is not affected.

Command Format

The general format for the **csplit** command is as follows:

csplit [**-s**] [**-k**] [**-f** *prefix*] **filename arg1** [**arg2 ... arg***n*]

The **csplit** command normally outputs the character count of each section as the section is created. The **-s** option will suppress the printing of these character counts. The process is complete when the system response is returned.

If an error occurs during the **csplit** operation, the sections that have been created are removed. The **-k** option overrides the removal of previously created files. However, the process will halt at the point the error occurred. The current section and the remainder of the original file will not be processed.

The created files are normally named **xx00** thru **xxnn**. If the **-f** *prefix* option is used, the files are named *prefix***00** through *prefix***nn**.

The **filename** is the name of the original file that is to be split. The command will start at the beginning of the file and search for the first

argument. That section is then written into a file, and the argument is used as the beginning for the next section. The arguments for the **csplit** command can be any combination of the following:

/string/ A file will be created from the current line up to, but not including, the line containing the character string *string*. This string may be followed by a *+n* or *-n* where *n* is a line number. For instance, if your file should contain **Page 5** and the three lines that follow it in the original file, the expression would be **/Page 5/+3**. If the character string has blanks or other significant characters to the command, the string must be enclosed in quotes.

%string% This argument acts exactly like */string/* except that no file will be created for the section from the current line to the line containing *%string%*.

zzz A file will be created from the current line up to, but not including, line number *zzz*. The line numbered *zzz* would then become the current line, that is, the first line in the next section.

{num} The argument that appears before *{num}* will be repeated *num* times. If the argument is a *string* type argument, that argument is searched for *num* more times. By using *{num}* after the *zzz* argument, you can split a file *num* times every *zzz* lines. It is a good idea to use the **-k** option with this argument because if the *{num}* number is too high, you will receive an error message and lose the files that have already been created.

Sample Command Use

The following example shows how to split the file **basic** into three pieces, **bas00**, **bas01**, and **bas02**. **The first line of bas01** will contain the string *test procedures*. The first line of **bas02** will contain the string *2.05*.

```
$ csplit -f bas basic " /test procedures/" /2.05/<CR>
2345
1068
297
$
```

The following example shows how to split the file **doc** into pieces of 100 lines each. To be sure that the entire file is split, an arbitrary number, 99 has been used for the number of times to split the file. Any lines over 10,000 will not be split. The **-s** option is used to suppress the character counts of each 100-line file.

```
$ csplit -s -k doc 100 {99}<CR>
$
```

The 100-line files would be named **xx00** through **xx***nn*.

The following example shows how to save the last piece of the file **mail**. The saved file, **xx00**, contains the text from the line **MISC** to the end of the file.

```
$ csplit mail %MISC%<CR>
$
```

"cut" — Output Selected Fields of a File

General

The **cut** command is used to output selected columns or fields from a line of data. The lines of data operated on by the **cut** command can be from one or more files, the output of another command, or from the terminal (standard input).

Command Format

The general format of the **cut** command is as follows:

cut -c*list* [*file(s)*]

cut -f*list* [**-d***char*] [**-s**] [*file(s)*]

The **-c***list* argument identifies the character positions in each line that are to be output. Individual character positions are identified by integers. A comma (,) is used to separate each position identifier. Ranges are specified by using a dash between the starting and ending number in the range. For example, character positions 1, 5, and 7 through 10 are identified as follows:

-c1,5,7-10

The **-f***list* argument identifies the field positions of each line that are to be output. A comma (,) is used to separate each field identifier. Ranges are specified by using a dash between the starting and ending number in the range. For example fields 1, 5, and 7 through 10 are input as follows:

-f1,5,7-10

The **-s** option is used with the **-f***list* argument to prevent lines that do not contain field delimiters from being output.

The **-d***char* argument identifies the field delimiter. The default field delimiter is the tab character. For example, the argument **-d:** defines a colon as the field delimiter. Delimiter characters that have a special meaning to the shell must be either placed in single quotes or escaped by preceding the character with a backslash (\). For example, the space can be defined as a field delimiter by the following:

-d' '

The *file(s)* argument identifies the name or names of the files that are to be operated on by the command.

Sample Command Use

The following sample command line entries and system responses show you how to output the character positions 5 through 10 and 15 from each line of a file named **list**. In this example, the **cat** command is first used to display the contents of the **list** file.

```
$ cat list<CR>
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      1111111111222222
12345678901234567890123456
$ cut -c5-10,15 list<CR>
EFGHIJO
   11
5678905
$
```

The following sample command line entries and system responses show you how to output the second and fifth fields from a file named **table**. The field separator (delimiter) is a colon (:). Note what happens when the delimiter is defined as a space by the **-d' '** argument. Also, note that the sequence in which you define the fields (**-f5,2** verse **-f2,5**) does not change the sequence in which they are output. The selected fields are output in the order that they appear in the data, from left to right. In this example, the **cat** command is used to display the contents of the **table** file.

```
$ cat table<CR>
field 1:field 2:field 3:field 4:field 5:field 6
field 1:field 2:field 3:field 4:field 5:field 6
field 1:field 2:field 3:field 4:field 5:field 6
$ cut -f2,5 -d: table<CR>
field 2:field 5
field 2:field 5
field 2:field 5
$ cut -f5,2 -d':' table<CR>
field 2:field 5
field 2:field 5
field 2:field 5
$ cut -f2,5 -d' ' table<CR>
1:field 4:field
1:field 4:field
1:field 4:field
$
```

"diff3" — 3-Way Differential File Comparator

General

The **diff3** command compares three files and outputs information showing the range of lines that differ between the files. The information is separated by a string of equal signs (====) to signify that the files are different. If the string of equal signs is alone, this shows that all files differ. If the string of equal signs is followed by a number, the number signifies what file is different. For example, ====**2** would show that the second named file is different and the information following the ====**2** would show the differences.

The range of lines that are different are shown in the format "**f:ln ed**" where;

f = the number of the file as it was entered in the command line.

ln = the line number that is different. This could be a range of lines.

ed = the editor command that needs to be performed to bring the files into agreement with each other. If the **c** (change) operation is shown, the original contents of the file will be shown immediately after the range of lines information.

Command Format

The general format for the **diff3** command is as follows:

diff3 [-ex3] *file1 file2 file3*

The **-e** or **-x** options publish a script file with the editor commands needed to make the first named file agree with the third named file. The script file contains all the commands necessary to make the proper changes and may be applied directly to the file.

Sample Command Use

The sample commands used in this section are based on the usage of three files named **a**, **b**, and **c**. The contents of the three files are shown below:

a:	b:	c:
A	B	C
B	C	D
C	D	E
D	E	A
E	A	B

The following command and system response show how to compare the three files and have the standard output displayed:

```
$ diff3 a b c<CR>
====
1:1,2c
  A
  B
2:1c
  B
3:0a
====
1:5a
2:5c
  A
3:4,5c
  A
  B
$
```

The following command line and system response show how to compare the three files and receive an editor script file that will make **a** agree with **c**:

```
$ diff3 -e a b c<CR>
5a
A
B
1,2c
w
q
$
```

The script file that is produced can be applied directly to the file being changed. This can be done on the same command line as the **diff3** command. The following command line shows how to compare the three files and apply the script file to **a**. There will be no output from this command, completion is shown with the system prompt.

```
$ diff3 -e a b c ¦ ed - a<CR>
$
```

"dircmp" — Directory Comparison

General

The **dircmp** command compares two directories and outputs information about the names and contents of the files in each directory. The output is paginated to list those files that are uncommon to each directory and then those files that have common file names. The files common to both directories are compared, and the output includes whether the contents are the "same" or "different".

Command Format

The general format for the **dircmp** command is as follows:

dircmp [**-d**] [**-s**] *dir1 dir2*

The **-d** option makes a comparison of the files common to both directories and gives information on what must be done to bring the two files into agreement. The format for the output is identical to the format of the **diff** command covered previously in this chapter.

The **-s** option will suppress any messages about identical files. That is, the output will only contain information on the files that are different from each other.

Sample Command Use

The examples provided in this section are based on the contents of the two directories **dir1** and **dir2**. The contents of the two directories follow:

dir1:	dir2:
appendix	**appendixA**
chap1	**appendixB**
chap2	**chap1**
chap3	**chap2**
chap4	**chap3**
index	**chap4**
table	**index**
toc	**toc**
trademarks	**trademarks**

The following command line and system response show how to compare **dir1** and **dir2**:

```
$ dircmp dir1 dir2<CR>

July 19 09:02 1985  ../dir1 only and ../dir2 only Page 1


./appendix          ./appendixA
./table             ./appendixB




July 19 09:02 1985  Comparison of ../dir1 ../dir2 Page 1


directory       .
same            ./chap1
different       ./chap2
different       ./chap3
different       ./chap4
same            ./index
different       ./toc
same            ./trademarks



$
```

Note: The output used in this example contains only the text of the output. The output is paginated with "white space" separating the uncommon files in each directory from the section displaying the common file names.

The following command line shows how to compare the files in **dir1** and **dir2** and output information that tells what must be done to bring the files into agreement:

```
$ dircmp -d dir1 dir2<CR>

Note: The first part of the output would appear
  the same as in the previous example. The
  last part of the output would be in the
  format identical to that of the diff
  command.

$
```

"egrep," "fgrep" — Search a File for a Pattern

General

The **egrep** and **fgrep** commands search files or input lines for matching character patterns. These commands are similar to the **grep** command explained in the *AT&T 3B2 Computer User Reference Manual.*

The input data to be searched can be the output of another command, one or more specified files, or the input from the terminal. When more than one file is searched, the file name is printed along with the matching input lines. The character patterns are regular expressions or fixed strings of characters in the style of the text editor (**ed**). Be careful when using the characters that have special meaning to the editor shell. In general, the pattern should be enclosed in single quotes ('pattern') to remove any special character meaning.

The *expression* **grep** (**egrep**) searches for full regular expressions. The **egrep** command accepts the following conventions for defining expressions:

- A pattern followed by a plus sign (+) matches one or more occurrences of the pattern.
- A pattern followed by a question mark (?) matches 0 or 1 occurrences of the pattern.
- Multiple patterns can be defined by separating each pattern by a pipe symbol (|) or by a new-line (carriage return). When a new-line is used, the secondary system prompt (>) is displayed. Each pattern is entered on a separate line following the prompt. The last pattern is entered on the same line as the remainder of the command. The command outputs matches for any or all patterns.

- Patterns can be grouped by enclosing the pattern in parentheses.

The *fast* **grep** (**fgrep**) command searches for fixed patterns. This command is fast and compact.

Command Format

The general format for each of these commands is as follows:

egrep [*options*] [*expression*] [*file(s)*]

fgrep [*options*] [*string(s)*] [*file(s)*]

The *options* recognized by these commands are explained as follows:

-b	Outputs the block number of the matching line. Each line is preceded by the number of the data block containing the line.
-c	Outputs only the number of lines that match the pattern.
-e *expression*	Same as a simple *expression* argument, but is useful when the *expression* contains a —.
-f *file*	The *pattern* (expressions or strings) is read (taken) from the specified *file*.
-l	Outputs only the names of the files that contain matching lines.
-n	Each line is preceded by its relative line number in the file.
-v	Outputs the lines that DO NOT contain the defined pattern.

-x	Outputs the lines that match the pattern exactly and entirely. This option is used with the **fgrep** command only.

The *expression* and *string* arguments define the search pattern or patterns. The *file(s)* argument is used to identify the file or files that are to be searched. Note that the file names are separated by a space.

Sample Command Use

The following command line and system response show how you can search two files (**list1** and **list2**) for lines containing one of several patterns. The patterns to be searched for are *eggs* and *bacon*. The **-n** option is used to display the line number of the matching line.

```
$ egrep -n 'eggs|bacon' list1 list2<CR>
list1:2:eggs
list2:1:bacon
list2:3:eggs
$
```

Note: The semicolon (:) is used to separate each field of the output of the **egrep** command. The first field is the file name. The second field is the line number of the matched pattern in the named file. The last field is the line containing the matching pattern.

The following command line and system response show you how to enter the previous example using a new-line (carriage return) to separate the patterns instead of a pipe symbol (|):

```
$ egrep -n 'eggs<CR>
> bacon' list1 list2<CR>
list1:2:eggs
list2:1:bacon
list2:3:eggs
$
```

The following command line and system response show how you can search multiple files for lines that DO NOT contain a specified pattern. The **-v** option causes all lines that DO NOT match the specified patterns to be output. The **-n** option is used again to output the line number of the matching line.

```
$ egrep -nv 'eggs|bacon' list1 list2<CR>
list1:1:milk
list1:3:toast
list1:4:ham
list2:2:milk
list2:4:juice
list2:5:bread
$
```

The following example shows how you can use a file containing a list of patterns to search a group of files. The file to search from is named **words** and contains the following patterns: **570ab[3-7], 448hj2, 747bg32**. The first line of the **words** file defines a pattern beginning with 570ab and ending with 3, 4, 5, 6, or 7. The **egrep** command will search all files in the current directory that begin with the characters **serials**.

```
$ egrep -f words serials*<CR>
serialsnet:570ab4
serialsnet:570ab5
serialsold:747bg32
serialsnew:570ab3
serialsnew:570ab7
serialsnew:448hj2
$
```

"file" — Determine File Type

General

The **file** command is used to determine the contents of one or more specified files. The command examines the contents of the first block of data (1024 bytes) of each file and attempts to classify the data. A file called **/etc/magic** is used by the **file** command to classify files containing certain special numeric or string constants. If you enter **cat /etc/magic**, an explanation of the **magic** file format is displayed.

Some of the file-types that can be classified are:

3b2/3b5 executable
3b2/3b5 executable not stripped
ASCII text
c program text
commands text
data
directory
empty
English text
[nt]roff, tbl, or eqn input text

Note: The file must have read permission before a classification can be made.

Command Format

The general format of the **file** command is as follows:

file [**-c**] [**-f** *ffile*] [**-m** *mfile*] *name(s)*

The **-c** argument causes the command to check the magic file for format errors. No file classification is done with this function.

The **-f** *ffile* option is used to specify the name of a file that contains a list of file names that are to be examined. The *ffile* argument identifies the name of the file containing the list of file names to be examined.

The **-m** *mfile* option is used to specify an alternate magic file. The *mfile* argument identifies the name of an alternate magic file.

Sample Command Use

The following command line entry and system responses show how you can determine the classification of a given file:

```
$ file /f1/house/bills/electric<CR>
/f1/house/bills/electric:      ascii text
$
```

The following command line entries and system responses show how you can determine the classification of several files. A file named **list** is first created that contains a listing of the files to be examined. The **file** command is then executed with the **-f** option to classify the files identified in the **list** file.

```
$ ed list<CR>
?list
a<CR>
/f1/house/bills/electric<CR>
/f1/house/bills/water<CR>
/f1/house/bills/gas<CR>
/f1/house/bills/telephone<CR>
.<CR>
w<CR>
94
q<CR>
$ file -f list<CR>
/f1/house/bills/electric:  ascii text
/f1/house/bills/water:  ascii text
/f1/house/bills/gas:  ascii text
/f1/house/bills/telephone:  ascii text
$
```

"join" — Relational Data Base Operator

General

The **join** command is used to join a common field of two files. The results are printed on your terminal screen. The fields that are to be joined must be sorted in an increasing ASCII collating sequence. Normally, the first field in each line is the field to be joined. A blank, tab, or new-line usually separates the fields. Multiple separators will be counted as one, and leading separators are discarded.

One line of output is generated for each pair of lines in the files that have identical join fields. The output line normally consists of the common field followed by the rest of the line from the first file, followed by the rest of the line from the second file.

Command Format

The general format of the **join** command is as follows:

join [*options*] *file1 file2*

The following options exist:

-a*n*	In addition to the normal output, a line is produced for each unpairable line in file *n* (where n is 1 or 2).
-e *s*	Replace empty output fields by string *s*.
-j*n m*	Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.
-o *list*	Each output line includes the fields specified in *list* and each element of *list* has the form *n.m* (where *n* is a file number and *m* is a field number).

-t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

The *file1* and *file2* arguments are the names of the files that are to be joined.

Sample Command Use

The following command line entries and system responses show the basic operation of the **join** command. The **cat** command is used to display the contents of **file1** and **file2**. The **join** command is used to join an inventory of red and blue items.

```
$ cat file1<CR>
1. red balls (7)
2. red bicycles (3)
3. red cars (5)
4. red ink pens (10)
5. red shoes (2 pair)
$ cat file2<CR>
1. blue balls (4)
2. blue bicycles (2)
3. blue cars (3)
4. blue ink pens (5)
5. blue shoes (3 pair)
6. blue pants (2 pair)
$ join -a2 file1 file2<CR>
1. red balls (7) blue balls (4)
2. red bicycles (3) blue bicycles (2)
3. red cars (5) blue cars (3)
4. red ink pens (10) blue ink pens (5)
5. red shoes (2 pair) blue shoes (3 pair)
6. blue pants (2 pair)
$
```

"newform" — Change the Format of a Text File

General

The **newform** command is used to read lines from a file or the standard input, reformat those lines, and reproduce the lines on the standard output. The format is selected through the command line options listed under "Command Format".

Command Format

The general format of the **newform** command is as follows:

newform [**-s**] [**-i***tabspec*]
[**-o***tabspec*] [**-b***n*]
[**-e***n*] [**-p***n*]
[**-a***n*][**-f**] [**-c***char*]
[**-l***n*] [*file(s)*]

where:

-i*tabspec* This option expands tabs into spaces. The *tabspec* part of this option uses the same tab specifications used with the **tab** command. Tab specifications may be found on the first line of the standard input. Here, use a double minus sign (--)as the *tabspec*. If *tabspec* is not specified, **-8** is used. If **-0** is given as the *tabspec*, there should not be any tabs in the text. If tabs are found in the text, they are treated as **-1**.

-o*tabspec* This option will replace spaces with tabs. The *tabspec* part of this option uses the same tab specifications as the *tabspec* part of the **-i***tabspec* option. If *tabspec* is not specified, **-8** is used. If a *tabspec* of **-0** is specified, spaces will not be converted to tabs.

-l*n* The effective line length is set to *n* characters. If this option is not specified, the effective line length is 80 characters. If **-l** is specified without *n*, the effective line length is set to **72**. Tabs and backspaces are considered to be one character. Remember, tabs may be expanded to spaces by the **i***tabspec* option.

-b*n* Shorten the beginning of the line by *n* characters when the line length is greater than the effective line length set by the **-l***n* option. If *n* is not specified, the line will be shortened by the amount of characters necessary to obtain the effective line length set by the **-l***n* option. It is a good idea to specify **-l***n* as **-l1** when using this option. That way, you will be sure that this option will be started, because the effective line length will be shorter than any line in the file.

-e*n* This option works the same as the **-b***n* option except that characters are removed from the end of the line.

-c*k* Change the prefix character (see **-p***n* option) and/or the append character (see **-a***n* option) to *k*.

-p*n* Prefix *n* characters to the beginning of a line when the line length is less than the effective line length set by the **-l***n* option. Spaces will be the prefix if the **-c***k* option is not specified. If *n* is not specified, the number of characters necessary to obtain the effective line length will be the prefix number.

-a*n* This option is the same as the **-p***n* option except that characters are appended to the end of a line.

-f Write the tab specification format line on the standard output before printing the output. The **-o***tabspec* option determines what tab specification format line is printed. If the *tabspec* part of the **-o***tabspec* option is not specified, the line printed will be the default specification of **-8**.

-s Shears the leading characters off each line up to the first tab. Up to eight of the sheared characters are placed at the end of the line. If more than eight characters are sheared, the eighth character is replaced by an * and the rest are discarded. The first tab is always discarded.

There must be a tab on each line of the file. If there is not a tab on each line, an error message and a program exit will occur. The characters sheared off are saved internally until all other options specified are applied to that line. These characters are then added at the end of the processed line.

files(s) The name of the file(s) that is to be read.

The command line options may appear in any order, may be repeated, and may be mingled with *file(s)*. However, if you use the **-s** option, it must be the first option specified.

Sample Command Use

The following command line entries and system responses show you a typical **newform** command output. The **cat** command is used to display the contents of **testfile**. The **newform** command is used to display the contents of **testfile** while removing the first three characters of each line and keeping the same column definition.

```
$ cat testfile<CR>
  RENTAL ITEM  DATE RENTED    DATE RETURNED

1. ladder      7/15/85       7/16/85
2. lawn mower  7/16/85       7/17/85
3. spray gun   7/17/85       7/18/85
4. tiller      7/18/85       7/19/85
5. weed eater  7/19/85       7/22/85
$ newform -i -l1 -b3 testfile<CR>
RENTAL ITEM     DATE RENTED    DATE RETURNED

ladder         7/15/85       7/16/85
lawn mower     7/16/85       7/17/85
spray gun      7/17/85       7/18/85
tiller         7/18/85       7/19/85
weed eater     7/19/85       7/22/85
$
```

The following command line entry and system response show how to display the contents of **testfile** without the last column:

```
$ newform -i -l1 -e13 testfile<CR>
  RENTAL ITEM  DATE RENTED

1. ladder      7/15/85
2. lawn mower  7/16/85
3. spray gun   7/17/85
4. tiller      7/18/85
5. weed eater  7/19/85
$
```

"nl" — Line Numbering Filter

General

The **nl** command is used to read lines from a file or the standard input. The lines that are read are numbered and printed on your terminal screen. The way the lines are numbered depends on the options you select.

The **nl** command views the text it reads in terms of logical pages. A logical page contains three sections: a header, a body, and a footer section. You can have empty sections. The options for numbering lines can be different for each of the three sections. In order for the three sections to be recognized, the following delimiter character(s) must be included in the input lines:

LINE CONTENTS	START OF
\:\:\:	header
\:\:	body
\:	footer

Note: There must not be any other input on the lines containing the delimiter character(s).

The **nl** command assumes the text being read is a single, logical page body unless you select other options.

Command Format

The general format of the **nl** command is as follows:

nl [-**h***type*] [-**b***type*] [-**f***type*] [-**v***start#*] [-**i***incr*] [-**p**] [-**l***num*] [-**s***sep*] [-**w***width*] [-**n***format*] [-**d***delim*] *file*

where:

-**h***type* Used to specify what logical page header lines are to be numbered. The recognized types are:

a	Number all lines
t	Number lines with printable text only
n	No line numbering
p*string*	Number only the lines that contain the regular expression specified in *string*.

The default *type* for the logical page header is **n**.

-**b***type* Used to specify what logical page body lines are to be numbered. The recognized types are the same as -**h***type*. The default *type* for the logical page body is **t**.

-**f***type* Used to specify what logical page footer lines are to be numbered. The recognized types are the same as -**h***type*. The default *type* for the logical page footer is **n**.

-**v***start#* The initial value used to number the logical page lines. Default is **1**.

-**i***incr* The increment value used to number the logical page lines Default is **1**.

-p Do not restart numbering at the logical page delimiters.

-l*num* The number of blank lines to be considered as one. The appropriate **-ha, -ba**, and **-fa** option must be set. A **-l2** results in only the second adjacent blank line being numbered. Default is **1**.

-s*sep* The character(s) used in separating the line number and the corresponding text line. Default is a tab.

-w*width* The number of characters to be used for the line number. Default is **6**.

-n*format* The line numbering format. The recognized values are:

- **ln** Left justified with no leading zeros
- **rn** Right justified with no leading zeros
- **rz** Right justified with leading zeros.

Default is **rn**.

-d*delim* Used to change the delimiter characters. If only one character is entered, the second character remains the default character (:). If you wish to use a backslash (\) as a delimiter character, you need to enter two backslashes (\\).

file The name of the file that is read by the **nl** command.

The options can be specified in any order and can be mingled with an optional file name. However, when intermingling options with a file, only one file can be moved.

Sample Command Use

The following command line entries and system responses show the basic operation of the **nl** command. The **cat** command is used to show the contents of file1. The **nl** command is used with several options to show how the command is input and the system response.

```
$ cat file1<CR>
\:\:\:
THIS IS THE HEADER SECTION
...
\:\:
This is the body section.
...
...
...
...
\:
THIS IS THE FOOTER SECTION
...
...
$ nl -ha -fa -v5 -i5 -nrz file1<CR>

000005  THIS IS THE HEADER SECTION
000010  ...

000015  This is the body section.
000020  ...
000025  ...
000030  ...
000035  ...

000040  THIS IS THE FOOTER SECTION
000045  ...
000050  ...
$
```

"od" — Octal Dump

General

The **od** command is used to output data in octal, decimal, ASCII, or hexadecimal formats. The name of the command, octal dump, is derived from the default output. Input can be from a named file, the output of another command, or from the standard input.

Command Format

The general format of the **od** command is as follows:

od [**-bcdoscx**] [*file*] [[+]*offset*[.][**b**]]

The meaning of the various format options are as follows:

-b Interpret bytes in octal (base 8).

-c Interpret bytes in ASCII.

-d Interpret words in unsigned decimal (absolute value).

-o Interpret words in octal. This is the default when no option argument is supplied.

-s Interpret 16-bit words in signed decimal.

-x Interpret words in hexadecimal (base 16).

The *file* argument identifies the name of the file to be output. If the *file* argument is omitted, input is taken from the standard input.

The *offset* argument identifies where the output is to start. This argument is normally expressed as the number of octal bytes to be skipped before data is output. If a period (.) is appended to the *offset*

argument, the argument is interpreted as a decimal number of bytes. If a letter **b** is appended to the argument, the argument is interpreted as the number of blocks to be skipped before data is output. If the *file* argument is omitted, the *offset* argument must be preceded by a plus sign (+) to identify what follows as being the *offset* argument.

Sample Command Use

The following command line entries and system responses show how you can output the contents of a file named **list1** using the default form of the **od** command. This form of the command outputs octal words. The **cat** command is first used to display the normal ASCII contents of the file.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ od list1<CR>
0000000 062547 063563 005142 071145 060544 005155 064554 065412
0000020 061165 072164 062562 005155 062541 072012
0000034
$
```

The following command line entries and system responses show how you can output the contents of a file named **list1** using the **-b** option of the command. This form of the command outputs the octal value for each character (byte). An *offset* of 27 bytes is used in this example. This *offset* causes the output to start at the last line of the file in this example. The **cat** command is first used to display the normal ASCII contents of the file. You need to refer to an octal map of the ASCII character set to make sense out of the output. For example, the letter "m" is octal 155; the letter "e" is octal 145; a new-line character is octal 012.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ od -b list1 27<CR>
0000027 155 145 141 164 012 000
0000034
$
```

"pack" — Compress Files

General

The **pack** command is used to compress and store files. Text files can be reduced between 60% and 75% of their original size. Load modules that use a larger character set and have a more uniform distribution of characters can be reduced to about 90% of their original size. The original file is removed and the compressed data is stored in a file with the same file name, except that a **.z** is added to the end of the file name. For example, if you compressed a file named **file1**, the compressed data will be stored in **file1.z**. The access modes, access and modified dates, and owner will remain the same as the original file. The compressed file can be restored to its original form using the **pcat** or **unpack** command.

How much a file is compressed depends on two things. They are:

1. The size of the input file
2. The character frequency distribution.

Usually, it is not worthwhile to compress files smaller than three blocks of data because a decoding tree is placed at the beginning of the compressed file. However, if the character frequency distribution is skewed, you may wish to compress the file even if the file is less than three blocks. Some reasons for the character frequency distribution being skewed are printer plots, pictures, or tables in the file.

The **pack** command will not work if:

1. The file appears to be already compressed.
2. The file name has more than 12 characters.

3. The file is linked to another file.
4. The file is a directory.
5. The file cannot be opened.
6. No disk storage blocks will be saved by compression.
7. A file called *name*.**z** already exists.
8. The **.z** file cannot be created.
9. An input/output error occurred during processing.

The **pack** command returns a value that is the number of files that it failed to compress.

Command Format

The general format of the **pack** command is as follows:

pack [-] *name ...*

where:

-	Used to set an internal flag that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of - in place of *name* will cause the internal flag to be set and reset.
name	The name of the file to be compressed.

Sample Command Use

The following command line entries and system responses show the basic operation of the **pack** command. The **ls -l** command is used to show what are the file sizes. The **pack** command is used to show how the command is inputted and the response you will receive. The **ls -l** command is used again to show the size of the files after they are compressed.

```
$ ls -l<CR>
total 109
-rw-------  1 cec   other      29727 July 19 13:14 file1
-rw-------  1 cec   other      24355 July 19 13:15 file2
$ pack - file1 file2<CR>
pack: file1: 36.2% Compression
      from 29727 to 18980 bytes
      Huffman tree has 15 levels below root
      90 distinct bytes in input
      dictionary overhead = 112 bytes
      effective  entropy  = 5.11 bits/byte
      asymptotic entropy  = 5.08 bits/byte
pack: file2: 36.2% Compression
      from 24355 to 15530 bytes
      Huffman tree has 15 levels below root
      85 distinct bytes in input
      dictionary overhead = 107 bytes
      effective  entropy  = 5.10 bits/byte
      asymptotic entropy  = 5.07 bits/byte
$ ls -l<CR>
total 73
-rw-------  1 cec   other      18980 July 19 13:14 file1.z
-rw-------  1 cec   other      15530 July 19 13:15 file2.z
$
```

"paste" — Side-by-Side File Merge

General

The **paste** command is used to combine two or more files of data in a side-by-side fashion. Each file of data is treated like a column of data in a table. The output of the paste command can be displayed on the terminal, redirected to a file, or redirected to another command.

Command Formats

Three general forms of the **paste** command are provided. These forms are as follows:

paste *file(s)*

paste -d *'list' file(s)*

paste -s [**-d** *'list'*] *file(s)*

The *file(s)* argument identifies the names of the files that are to be pasted together. The hyphen (-) can be used as a file name to read a line from the standard input. There is no prompting associated with the use of the hyphen.

The **-s** option is used to merge several lines from each input file as opposed to one line from each input file.

The **-d***list* argument option is used to define the delimiter(s) that is used between the merged lines. The tab character is the default delimiter. The *list* argument identifies what is to replace the tab delimiter. The items (characters) identified in the *list* argument are used in sequence until the end of the list. Then, the listed delimiters are reused in the same sequence. In general, the **list** argument should be in double quotes.

For example, to get one backslash, use **-d'\\'** as the argument; use **-d' '** as the argument to define a space as the delimiter. Special characters are defined by escape sequences. These include the following:

- **\n** for new-line character
- **\t** for tab character
- \\ for the backslash character
- **\0** for an empty string.

Sample Command Use

The following examples are based on the use of two files named **list1** and **list2**. The contents of these two files are as follows:

```
list1:          list2:

list1: item1    list2: item1
list1: item2    list2: item2
list1: item3    list2: item3
list1: item4    list2: item4
list1: item5    list2: item5
list1: item6    list2: item6
list1: item7    list2: item7
```

The following command line entries and system responses show how you can merge the contents of two files using the simplest form of the **paste** command. This form of the command requires no options. The named files are merged in a side-by-side fashion with a tab character as the delimiter between the lines of the files. The output of the **paste** command is redirected to a file named **save**. The **cat** command is used to display the contents of the resulting file.

```
$ paste list1 list2 > save<CR>
$ cat save<CR>
list1: item1    list2: item1
list1: item2    list2: item2
list1: item3    list2: item3
list1: item4    list2: item4
list1: item5    list2: item5
list1: item6    list2: item6
list1: item7    list2: item7
$
```

The following command line entries and system responses show how you can merge the contents of two files using the form of the **paste -d***list* command. The named files are merged in a side-by-side fashion with a slash (/) character as the delimiter between the lines of the file. The output of the **paste** command is redirected to a file named **save**. The **cat** command is used to display the contents of the resulting file.

```
$ paste -d'/' list1 list2 > save<CR>
$ cat save<CR>
list1: item1/list2: item1
list1: item2/list2: item2
list1: item3/list2: item3
list1: item4/list2: item4
list1: item5/list2: item5
list1: item6/list2: item6
list1: item7/list2: item7
$
```

"pcat" — Concatenate and Print Packed Files

General

The **pcat** command is used to concatenate and print files that have been compressed by the **pack** command. The compressed file is expanded and printed on your terminal screen in its original form.

The **pcat** command will not work if:

1. The file name (exclusive of **.z**) has more than 12 characters.
2. The file cannot be opened.
3. The file does not appear to be the output of the **pack** command.

The **pcat** command returns a value that is the number of files that it failed to expand.

Command Format

The general format of the **pcat** command is as follows:

pcat *name ...*

The *name* argument identifies the name of the file that needs to be expanded. The **.z** at the end of the file name does not need to be inputted when specifying *name*.

The standard output of the **pcat** command can be redirected to a file. You will have two files: one that contains the compressed data (*name*.**z**) and one that contains the original data.

The general format when redirecting the output of the **pcat** command follows:

pcat *name* > *new.file*

where:

name	Identifies the name of the file that needs to be expanded.
new.file	Identifies the name of the file that contains the expanded data.

Sample Command Use

The following command line entries and system responses show the basic operation of the **pcat** command. The **ls -l** command is used to display the compressed files before the **pcat** command is given. The **pcat** command is used to show you how the command is input using the redirection method. The **ls -l** command is used again to display the results of executing the **pcat** command.

```
$ ls -l<CR>
total 71
-rw-------  1 cec  other      18980 July 19 13:14 file1.z
-rw-------  1 cec  other      15530 July 19 13:15 file2.z
$ pcat file1 > new.file1<CR>
$ pcat file2 > new.file2<CR>
$ ls -l<CR>
total 181
-rw-------  1 cec  other      18980 July 18 13:14 file1.z
-rw-------  1 cec  other      15530 July 18 13:15 file2.z
-rw-------  1 cec  other      29727 July 19 07:47 new.file1
-rw-------  1 cec  other      24355 July 19 07:48 new.file2
$
```

"pg" — Command Description

General

The **pg** command is a filter that will allow you to view a file one page at a time on a soft-copy terminal screen. A prompt (:) is displayed after every page. If a carriage return is entered after the prompt, another page is displayed. Other options, listed in this section, may be chosen. What makes the **pg** command different from other similar commands is that the **pg** command allows you to back up and review something that has already passed. The **pg** command scans the **terminfo** data base for your terminal type to determine the terminal attributes. The variable **TERM** specifies your terminal type. If **TERM** is not specified, the terminal type **dumb** is assumed. Refer to the *AT&T 3B2 Computer Programmer Reference Manual* for information on the **terminfo** data base.

A pause will occur after each page is displayed and the prompt is given. There are three categories of responses that can be given when the prompt is displayed. The three categories are those that cause further perusal, those that search, and those that change the perusal environment.

Commands that cause further perusal normally take a preceding *address*. The *address* is an optionally signed number that indicates the point from which further text should be displayed. The *address* can be given in pages or lines. A signed *address* specifies a point relative to the current page or line. An unsigned *address* specifies an address relative to the beginning of the file.

The perusal commands are as follows:

<newline> or *<blank>*
: Display the next page. If a signed *address* is used, the **pg** command goes forward (+) or backward (-) the numbered amount of pages specified and displays that page on your terminal screen. If an unsigned *address* is used, the page

number specified will be displayed.

l Scroll one line forward. If a signed *address* is used, the **pg** command simulates scrolling the screen, forward (+) or backward (-), the number of lines specified. If an unsigned *address* is used, the **pg** command prints a screenful beginning at the line number specified.

d or **^D** Simulates scrolling half a screen forward (+**1** *address*) or half a screen backward (-**1** *address*).

The next two perusal commands do not use addresses.

. or **^L** Causes the current page to be redisplayed.

$ Displays the last window (page) in the file. If the input is a pipe, use with caution.

The following commands are available for searching for specific patterns of text. These commands must be ended by a *<newline>*, even if the *-n* option is specified. You may use the regular expressions of the **ed** command. Refer to the *AT&T 3B2 Computer User Reference Manual* for information about the **ed** command.

i/pattern/ Search forward for the *i*th (default is *i*=1) occurrence of *pattern*. Searching will begin after the current page and will continue until the end of the file is reached. If the entire *pattern* is not on the same line, *pattern* will not be found.

i?*pattern*? or *i*^*pattern*^
: Search backward for the *i*th (default is *i*=1) occurrence of *pattern*. Searching will begin before the current page and will continue until the beginning of the file is reached. Use *i*^*pattern*^ if using an Adds 100 terminal.

The line found at the top of the screen will be displayed after the search has ended. By appending **m** or **b** to the search command, you can display the line at the middle of the window or the bottom of the window. The suffix **t** can be used to restore the original file.

You can change the perusal environment with the following commands:

***i*n**	Begin perusing the *i*th next file in the command line. If *i* is not specified, 1 is used.
***i*p**	Begin perusing the *i*th previous file in the command line. If *i* is not specified, 1 is used.
***i*w**	Display another window of text. If *i* is present, set the window size to *i*.
s *filename*	Save the current file that is being perused in *filename*. This command must be ended by a <*newline*>, even if the *-n* option is specified.
h	Help command. An abbreviated summary of available commands is displayed.
q or **Q**	Quit the **pg** command.
!*command*	The *command* is executed by the shell. If the **SHELL** environment variable is set, that shell is used. If the **SHELL** environment variable is not set, the default shell is used. This command must be ended by a <*newline*>, even if the *-n* option is specified.

You can stop sending output to the terminal at any time by depressing the quit key (normally control-\) or the interrupt (break) key. The prompt will appear and you may then enter commands in the normal manner. Unfortunately, some output is lost when you stop the output. This happens because any characters waiting in the terminal output queue are flushed when the quit signal occurs.

The **pg** command acts like the **cat** command if the standard output is not a terminal screen. The only difference is that a header is printed before each file if there is more than one file. Refer to the *AT&T 3B2 Computer User Reference Manual* for information about the **cat** command.

Execution of the **pg** command is stopped if **BREAK**, **DEL**, or is depressed while the **pg** command is waiting for terminal input. If you are between prompts, these signals interrupt the current task and will place you in the prompt mode. Use the interrupt signals with caution when the input is coming from a pipe, since the interrupt is likely to stop the other commands in the pipeline.

There are a couple of bugs that you need to know about. The first one is that the terminal tabs should be set to every eight positions or you may get undesirable results. The second one is that when using the **pg** command as a filter with another command that changes the terminal input/output options, terminal settings may not be restored correctly.

Command Format

The general format of the **pg** command is as follows:

pg [*-number*] [**-p** *string*] [**-cefns**] [*+linenumber*] [*+/pattern/*] [*file(s)*]

where:

-number	The size (number of lines) of the window. If the size is not specified, the default value is one line less

than the total number of lines that can be displayed on your terminal screen.

-p *string* — Causes *string* to be used as the prompt. If **%d** appears in *string*, the first occurrence of **%d** in the prompt is replaced by the current page number when the prompt is issued.

-c — Take cursor to the home position and clear the screen before displaying a page. If **clear_screen** is not defined in the **terminfo** data base, this option will be ignored.

-e — Normally, a pause will occur at the end of each page and at the end of each file. This option eliminates the pause at the end of each file.

-f — Normally, if a line is longer than the terminal screen width, it is split into two lines. However, there are times when some sequences of characters in the text generate undesirable results; such as escape sequences for underlining. Here, you can use the option to inhibit the **pg** command from splitting lines.

-n — Normally, commands must be ended by a <*newline*> character. This option causes the command to end as soon as a command letter is entered.

-s — Causes all messages and prompts to be printed in the standout mode (usually inverse video).

+*linenumber* — Start up at *linenumber*.

+/*pattern*/ — Start up at the first line containing the pattern specified.

file(s) The name of the file to be examined. If *file(s)* is not specified or if a minus sign (-) is specified, the **pg** command reads the standard input.

Sample Command Use

The following command line entry and system response show the basic operation of the **pg** command. The **pg** command along with the **news** command is used in a pipeline to read the system news.

```
$ news | pg -p " (Page %d):" <CR>
Note: The first page of the news will appear next.
(Page 1): Note: This is the prompt. It will appear
        after each page with the number of the page
        you are on. You may now enter a
        command that manipulates the text or enter
        q to quit.
$
```

"sdiff" — Side-By-Side Difference Program

General

The **sdiff** command uses the output of the **diff** command (discussed earlier in this chapter) to produce a side-by-side listing of two files. If the lines are identical, each line of the two files are printed side-by-side with a blank gutter between the two files. If the line exists only in *file1*, a less than (<) symbol is in the gutter. If the line exists only in *file2*, a greater than (>) is in the gutter. If the line exists in both files and they are different, a pipe symbol (|) is in the gutter.

For example:

```
x   |   y
a       a
b   <
c   <
d       d
    >   c
```

Command Format

The general format of the **sdiff** command is as follows:

sdiff [*options ...*] *file1 file2*

The following options exist:

-w *n*	Use the next argument (*n*) as the width of the output line. If *n* is not specified, the line length will be 130 characters.
-l	Only print the left side of any lines that are identical.

-s Do not print identical lines.

-o *output* Use the next argument (*output*) to create a third file that will let you control the merging of *file1* and *file2*. All identical lines of *file1* and *file2* are copied to the *output* file. All different lines of *file1* and *file2* are printed on your terminal screen. After the different lines are printed, you will receive a prompt (**%**). After the prompt (**%**) is received, enter one of the following commands:

l Append the left column to the output file.

r Append the right column to the output file.

s Turn the silent mode on; do not print identical lines.

v Turn the silent mode off.

e l Will let you edit the left column.

e r Will let you edit the right column.

e b Will let you edit the concatenation of the left and right columns.

e Will let you edit a new file.

q Exit from the program.

When you exit from the editor, the resulting file is concatenated on the end of the *output* file.

The arguments *file1* and *file2* are the files that are being compared.

Sample Command Use

The following command line entries and system responses show the basic operation of the **sdiff** command. The **cat** command is used to display the contents of file1 and file2. The **sdiff** command is then used to display a side-by-side comparison of file1 and file2.

```
$ cat file1<CR>
1000
2000
4000
8000
16000
32000
64000
128000
$ cat file2<CR>
500
1000
2000
3000
8000
16000
34000
64000
$ sdiff -w 30 file1 file2<CR>
        >       500
1000            1000
2000            2000
4000    |       3000
8000            8000
16000           16000
32000   |       34000
64000           64000
128000  <
$
```

"split" — Split a File Into Pieces

General

The **split** command is used to read a file and write it in *n* number of lines onto a set of output files. Default is 1000 lines per file. The name of the first output file is *name* with **aa** through **zz** appended. The output file will be appended with **aa**, then **ab**, then **ac**, and so forth until **zz** is reached. A maximum of 676 files can be created using the **split** command. There must not be more than 12 characters in *name*. If no output name is given, **x** is default.

Command Format

The general format of the **split** command is as follows:

split [*-n*] [*file* [*name*]]

where:

-n	The number of lines that are to be written onto each output file.
file	The name of the file to be split.
name	The name of the output file.

If no input file is given or if - is given, the standard input is used.

Sample Command Use

The following command line entries and system responses show the basic operation of the **split** command. The **cat** command is used to display the contents of **file1**. The **split** command is used to split **file1** into 3 lines per output file. The **ls -l** command is used to display the new files that are created. The **cat** command is used again to display the contents of each new file.

```
$ cat file1<CR>
This will be the first line of the first output file.
...
...
This will be the first line of the second output file.
...
...
This will be the first line of the third output file.
...
...
$ split -3 file1 new.file1<CR>
$ ls -l<CR>
total 5
-rw-------  1 cec   other     187 July 19 10:52 file1
-rw-------  1 cec   other      62 July 19 10:53 new.file1aa
-rw-------  1 cec   other      63 July 19 10:53 new.file1ab
-rw-------  1 cec   other      62 July 19 10:53 new.file1ac
$ cat new.file1aa<CR>
This will be the first line of the first output file.
...
...
$ cat new.file1ab<CR>
This will be the first line of the second output file.
...
...
$ cat new.file1ac<CR>
This will be the first line of the third output file.
...
...
$
```

"sum" — Print Check Sum and Block Count of a File

General

The **sum** command is used to calculate and output a 16-bit checksum for a specified file. Typically, the command is used to look for bad data or to validate a file transmitted over a communications interface. The number of blocks in the specified file is also output.

To use the **sum** command to validate transmitted data, the checksum is executed on the file before transmission and the results are sent to the destination. At the destination, the **sum** command is again executed on the received data. The before and after checksums are then compared. Matching checksums show a successful transfer of data and a mismatch shows a problem. Note that you must know whether to use the **-r** option when validating transferred data or not. You must use the same form of the command to calculate the checksum at the source and destination to be able to validate the transmitted data.

Command Format

The general format of the **sum** command is as follows:

sum [**-r**] *file*

The **-r** option causes the command to use a different rationale (algorithm) in computing the checksum. The *file* argument identifies the name of the file to be processed. Note that the file name can be expressed as a complete path name.

Sample Command Use

The following command line entries and system responses show you a typical **sum** command output. The first field output is the checksum, followed by the number of blocks (1), followed by the name of the file (**list1**).

```
$ sum list1<CR>
2496 1 list1
$ sum -r list1<CR>
55792    1 list1
$
```

"tail" — Output End of a File

General

The **tail** command is used to output the last portion of some data. The source data operated on by the command can be from a file, the output of another command, or from the terminal. Options are provided to tell the command at what point from the beginning or end of the input data to start passing data to the output. The start can be expressed in the number of lines, blocks, or characters from the beginning or end of the data.

Command Format

The general format of the **tail** command is as follows:

tail [±[*number*][**lbc**[**f**]]] [*file*]

The *number* argument identifies the number of units from the beginning or from the end of the input where the output is to begin. A plus sign preceding the number means from the beginning of the input data. A minus sign preceding the number means from the end of the input. The units used for the *number* argument are lines (**l**), blocks (**b**), or characters (**c**). The unit identifier immediately follows the *number* argument (no space).

The **-f** option is used to continuously read data from a file. The option provides the ability to monitor the growth of a file that is being written by some other process. The **-f** option is not applicable when data is being piped to the **tail** command.

The *file* argument identifies the name of the source file. Note that the file name can be expressed as a complete path name.

Sample Command Use

The following examples are based on the contents of a file named **sample**. The contents of this file are as follows:

line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
line 11
line 12

The following command line entries and system responses show how you can output the end of a file of data using the simplest form of the **tail** command. This form of the command outputs the last ten lines of the contents of the **sample** file. The default of the *number* argument is **-10l**.

```
$ tail sample<CR>
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
line 11
line 12
$
```

The following sample command lines and system responses show you how to output the contents of the **sample** file starting 45 characters from the beginning of the file:

```
$ tail +45c sample<CR>
ne 7
line 8
line 9
line 10
line 11
line 12
$
```

"tr" — Translate Characters

General

The **tr** command is used as a filter to change data that is passed through it on a character basis. The command functions like a stream editor. Repeated occurrences of a character in succession can be reduced to a single occurrence of the character. Characters can be identified by ASCII letter or octal value. Octal values are preceded by a backslash (\). Ranges of characters are identified by enclosing the range in brackets. For example, [**a-c**] represents the letters a, b, and c. The entire lower case ASCII range is identified by [**a-z**]. Multiple occurrences of a character is represented by an expression [**x*n**], where the **x** is any character and the **n** is the number of repetitions of **x**. The number is treated as an octal number if the most significant digit is a zero. The number is treated as a decimal number if the most significant digit is other than a zero.

Command Format

The general format of the **tr** command is as follows:

tr [**-cds**] [*string1* [*string2*]]

The **-c** option reverses the meaning of *string1*. The *string1* argument identifies the characters that ARE NOT to be translated. The characters identified in the *string1* argument pass unchanged to the output. When the **-c** option is omitted, *string1* characters are translated to *string2* characters.

The **-d** option causes the characters identified by *string1* to be deleted from the output. The *string2* argument is not used with the **-d** option. If the *string2* argument is provided, it will be ignored.

The **-s** option causes multiple occurrences of the characters identified by *string2* to be replaced by a single occurrence of the characters. If only one string argument is given, then *string1* defines the character(s) to be operated on by the command.

The *string1* argument identifies input characters that are to be operated on by the command. When a *string2* argument is also provided, the characters found in *string1* are mapped to the corresponding character in *string2*.

Sample Command Use

The following command line entries and system responses show how you can change all lower case letters in a file named **list1** to upper case letters. The **cat** command is used to display the contents of **list1**. The output of the **tr** command is redirected to a file named **LIST**.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ tr " [a-z]" " [A-Z]" < list1 > LIST<CR>
$ cat LIST<CR>
EGGS
BREAD
MILK
BUTTER
MEAT
$
```

The following command line entries and system responses show how you can use the **tr** command to reduce multiple consecutive occurrences of a space character to a single occurrence throughout a file of data. The **cat** command is used to display the contents of the files.

```
$ cat list3<CR>
This   file  contains  lines
with multiple   spaces  between words.
$ tr -s " " < list3 > newlist<CR>
$ cat newlist<CR>
This file contains lines
with multiple spaces between words.
$
```

The following command line entries and system responses show how you can use the **tr** command to put each word in a file on a separate line. The **cat** command is used to display the contents of the sample file. The output of the **tr** command is displayed on the terminal in this example.

```
$ cat file<CR>
This file contains one line of text.
$ tr -cs "[A-z]" "[\012*]" < file<CR>
This
file
contains
one
line
of
text
$
```

"uniq" — Report Repeated Lines in a File

General

The **uniq** command is used to read an input file while comparing adjacent lines. The second and succeeding copies of repeated lines are removed in the normal output mode and the remaining lines are written on the output file. Repeated lines must be adjacent to be found. The input and output files must have different names.

Command Format

The general format of the **uniq** command is as follows:

uniq [*-udc* [*+n*] [*-n*]] [*input* [*output*]]

where:

-u	The lines that are not repeated in the input file are written on the output file.
-d	Only one copy of just the repeated lines is written on the output file.
-c	The output file is generated in the normal output mode with a count of the number of times each line occurs. This option supersedes *-u* and *-d*.
+n	*n* amount of characters are ignored. Fields are skipped before characters. A field is defined as a string of nonspace, nontab characters separated by tabs and spaces from its neighbors.
-n	The first *n* amount of fields together with any blanks before each field are ignored.

input The name of the input file.

output The name of the output file.

The normal output mode is the union of the *-u* and *-d* options.

Sample Command Use

The following command line entries and system responses show the basic operation of the **uniq** command. The **cat** command is used to display a grocery list. The **sort** command is used to alphabetize the grocery list. To learn more about the **sort** command, refer to your *AT&T 3B2 Computer User Reference Manual*. The **cat** command is used again to display the alphabetized grocery list. The **uniq** command is used to remove the repeated lines and to count the number of times each item was listed. The **cat** command is used again to display the results. This is shown in the next example.

```
$ cat file1<CR>
bread
milk
butter
ice cream
meat
milk
vegetables
potato chips
drinks
orange juice
bread
vegetables
$ sort file1 > file2<CR>
$ cat file2<CR>
bread
bread
butter
drinks
ice cream
meat
milk
milk
orange juice
potato chips
vegetables
vegetables
$ uniq -c file2 file3<CR>
$ cat file3<CR>
   2 bread
   1 butter
   1 drinks
   1 ice cream
   1 meat
   2 milk
   1 orange juice
   1 potato chips
   2 vegetables
$
```

"unpack" — Expand Files

General

The **unpack** command is used to expand files created by the **pack** command. The compressed data is expanded to its original form. The compressed file is removed and the expanded data is placed in a file with the same file name, except that the **.z** is dropped. For example, if you expanded a file named **file1.z**, the expanded data will be placed in **file1**. The access modes, access and modified dates, and owner will remain the same as the compressed file.

The **unpack** command will not work if:

1. The file name (exclusive of **.z**) has more than 12 characters.
2. The file cannot be opened.
3. The file does not appear to be the output of the **pack** command.
4. A file with the "unpacked" name already exists.
5. The unpacked file cannot be created.

The **unpack** command returns a value that is the number of files that it failed to expand.

Command Format

The general format of the **unpack** command is as follows:

unpack *name ...*

The *name* argument identifies the name of the file that needs to be expanded. The **.z** at the end of the file name does not need to be input when specifying *name*.

Sample Command Use

The following command line entries and system responses show the basic operation of the **unpack** command. The **ls -l** command is used to display the character size of the compressed files before they are expanded. The **unpack** command is used to expand the compressed files. The **ls -l** command is used again to display the character size of the expanded files.

```
$ ls -l<CR>
total 71
-rw-------  1 cec   other      18980 July 19 13:14 file1.z
-rw-------  1 cec   other      15530 July 19 13:15 file2.z
$ unpack file1 file2<CR>
unpack: file1: unpacked
unpack: file2: unpacked
$ ls -l<CR>
total 110
-rw-------  1 cec   other      29727 July 19 13:14 file1
-rw-------  1 cec   other      24355 July 19 13:15 file2
$
```

Replace this

page with the

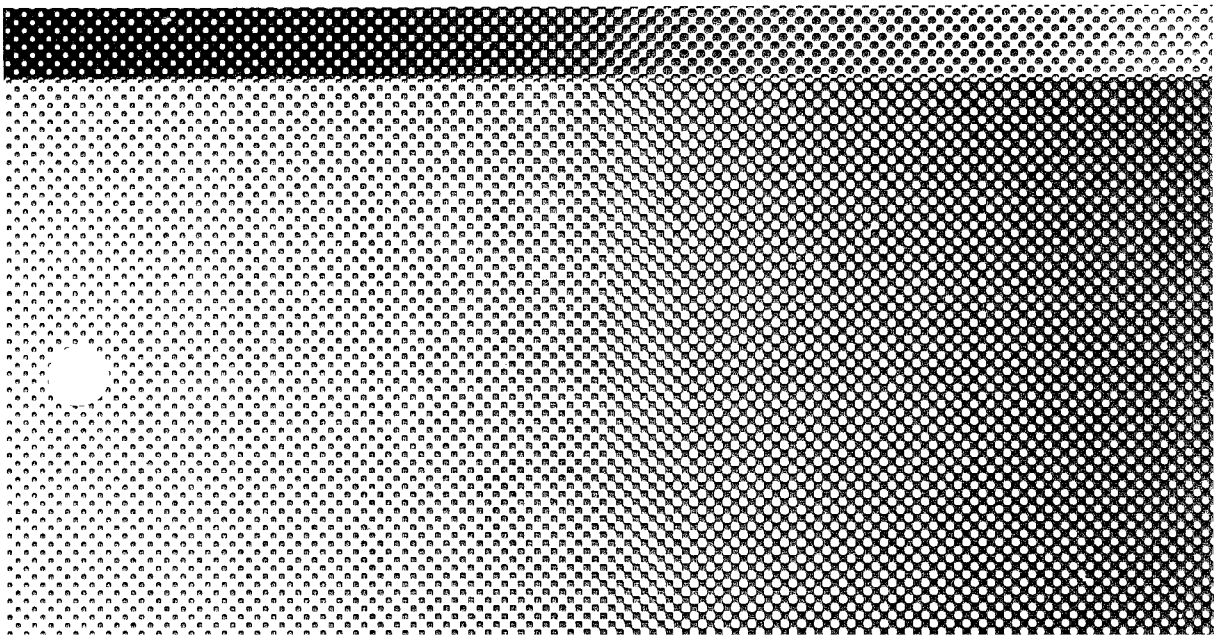
EDITING

tab separator.

AT&T

AT&T 3B2 Computer

UNIX™ System V Release 2.0 Editing Utilities Guide



CONTENTS

Chapter 1

INTRODUCTION

GENERAL

This guide describes the command format and use of the Editing Utilities. The commands and procedures described in this guide are for use by all users.

The **UNIX*** System contains a file system that is used to store user information. Changing files by adding or deleting information can only be done using UNIX System Editor Commands. The editing utilities give the user an easy way to create, read, and change information in these files.

The edit, ex, and vi editors are based on a consistent set of text editor commands. These commands serve as the fundamental building blocks for increasing text editing proficiency.

---

* Trademark of AT&T

The editing utilities allows the user to do two types of editing:

- **Basic editing** allows the casual user to use basic commands to do text editing.
- **Visual editing** allows the user to view several lines of the file at a time and use screen oriented display editing based on basic editor commands.

The editing utilities consists of three text editors designed to meet the needs of the novice user, while allowing the experienced user to use more complex and powerful editing tools. These editors are actually three versions of the ex editor.

The ex editor is an interactive editor that normally accesses only one line of the file at a time. Many of the ex commands are similar to the ed editor commands. The advantage of using the ex editor is the large amount of options available in it.

The edit editor is the simplified version of ex editor and is normally used by novice users. Messages displayed on the screen after an invalid command are more descriptive than with ex or vi. Edit contains fewer commands and most beginners should pick it up quickly. All commands that execute in the edit editor will also execute in the ex editor.

The vi editor is actually the visual mode of editing within the ex editor. Vi is the most complex of the three editors, because there are so many commands that do the same function. However, it is the easiest to use once you understand the basic movement and editing commands. With the vi editor, you can view several lines of the file at one time, and you can move the cursor to any character in the file. Most ex commands can be invoked separately from vi by first entering a ":" and then the ex command. To execute the command, depress the carriage return. Experienced users often mix their use of ex command mode and vi command mode to speed the work they are doing.

RESTRICTIONS

The limits of the editors are as follows:

- 1024 characters per line
- 256 characters per global command list
- 128 characters per file name
- 100 characters per shell escape command
- 63 characters in a string valued option
- 30 characters in a tag name
- 128 characters in the previous inserted or deleted text in (open) or (visual) mode
- 250000 lines in a file.

If you try to use these editors on a file and you receive a message stating that the file is too large, you can either split the file into smaller files or use a different editor. To split the file, you can use the **split** or **csplit** commands contained in the *AT&T 3B2 Computer Directory and File Management Utilities*. If you want to use another editor, you can use the **bfs** editor (big file scanner) contained in the *AT&T 3B2 Computer Directory and File Management Utilities* or the **sed** (stream) editor contained in the *Essential Utilities*.

SPECIAL PURPOSE KEYS

There are several special purpose keys that are used by the vi editor. These keys are important and will be used throughout the document. Their descriptions are as follows:

ESCAPE This key is sometimes labeled *<ESC>* or *<ALT>*. It is normally located in the upper left corner of your keyboard. When you are in the editor, depressing the *<ESC>* key causes the editor to ring the bell indicating that it is in an inactive state. On smart terminals where it is possible, the editor will quietly flash the screen rather than ring the bell. Partially formed commands are canceled with the *<ESC>* key. When you insert text in the file, text insertion is ended with the *<ESC>* key. This is a harmless key to use, so you can depress it whenever you are not certain what state the editor is in.

CR The *<CR>* key refers to the RETURN key and is used to start execution of certain commands. It is normally located on the right side of the keyboard.

DELETE This key is sometimes labeled *<DEL>*, *<RUBOUT>*, or *<BREAK>*. It generates an interrupt that tells the editor to stop what it is doing. This is a forceful way of making the editor return to the inactive state if you do not know or like what is going on.

CONTROL This key is often labeled *<CTRL>*. It is used with other keys to do various functions. It will be represented in this document by the *<CTRL>* symbol. The associated key will be represented by an uppercase letter. To execute a control function, both keys must be depressed at the same time. An example of this will be represented as follows:

<CTRL d>

The function illustrated will cause the screen to scroll down when in the vi editor.

HOW TO INTERPRET COMMANDS

The following conventions are used to show your terminal input and the system output in screens and command lines:

This style of type is used to show system generated responses displayed on your screen.

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <*CR*> carriage return, <*CTRL d*> control d, <*ESC g*> escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this document is organized as follows:

- Chapter 2, "EDIT EDITOR," provides instructions on how to use the edit editor.
- Chapter 3, "EX EDITOR," provides instructions on how to use the ex editor.
- Chapter 4, "VI EDITOR," provides instructions on how to use the visual (vi) editor.

Chapter 2

EDIT EDITOR

PAGE

Chapter 2

EDIT EDITOR

INTRODUCTION

This chapter describes the edit editor used on the 3B2 Computer. Edit is a simplified version of the ex editor, and it is recommended for new or casual users. Messages displayed on the screen after an invalid command are more descriptive than with the other editors.

When using the edit editor, all commands must be entered on a command line. The command line is identified by a colon ":" on a line by itself. Commands entered on the command line can affect the line you are on in the file (current line), a specified set of lines, or the entire file.

Most edit editor command names are English words that can be abbreviated. When an abbreviation conflict is possible, the more commonly used command has the shorter abbreviation. For example, since **substitute** is abbreviated by **s**, **set** is abbreviated by **se**.

The edit editor does not directly change the file being edited. Instead, it works on a copy of the file stored in a temporary memory location called the buffer. The edited file is not changed until you write the changes from the buffer to the edited file.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the edit editor, refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages.

CURRENT LINE DEFINITION

The term "current line" is referred to throughout this chapter. The current line is the line in the file you are now on. Each time you move to a different line in the file, that line becomes the current line. Whenever a command is given, the current line is used as a reference point. Any command that is not directed at any specific line is executed against the current line.

GETTING STARTED

The edit editor can be used to create a new file or to change an existing file. To execute edit you must be logged onto the computer. After the **$** or # prompt is displayed, you can begin working with the edit editor.

Creating a New File

To create a new file, you will need to type **edit** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

```
$ edit filename<CR>
" filename" [New file]
:
```

If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. You will need to re-enter the command correctly.

If you entered the **edit** command without a filename, the editor will still create a new file. However, when you decide to write the file into memory you will be prompted for a filename. See "Writing the Buffer Into the File."

When the edit command is executed, a colon ":" is displayed. The colon identifies the command line and indicates that the edit editor is ready to accept your input commands.

Entering Text

The edit editor commands have two forms: a word that describes what the command does and an abbreviation of the word. You can use either form. Many beginners find the full command name easier to remember, but after some practice use the abbreviation. The command to input text is **append**, that may be abbreviated **a**. Enter **append** after the colon on the command line and then depress the carriage return.

:**append**<*CR*> or :**a**<*CR*>

Edit is now in the *text input mode* (append mode). The colon is no longer displayed on the command line, and this is your signal that you may begin entering lines of text. Anything that you type on your terminal, except a period on a line by itself, is entered into the buffer. If the message:

```
Not an editor command
```

is displayed, check to see what you entered incorrectly and then enter the command again.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

Leaving the Input Mode

To leave the input mode, simply enter a period "." on a line by itself and depress the carriage return. This is the signal that you want to stop inputting text. After receiving a period on a line by itself, edit will re-enter the command mode and display the command line prompt ":".

The text just entered is now stored only in the buffer. If you wish, you can make changes to the text. Making changes is discussed throughout the remainder of this chapter.

Writing the Buffer Into the File

The buffer is only temporary storage for the file. Now that you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

:**write**<*CR*> or :**w**<*CR*>

Edit will then copy the buffer into the file. If the file does not yet exist, a new file will be created and a message will be given indicating that it is a

new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

Edit must have a filename to use before it can write a file. Therefore, if you did not specify the name of the file when you began the editing session, edit will issue the message:

```
No current filename
```

when you give the **write** command. If this happens, simply re-enter the **write** command and specify the filename. Here you would enter:

:write filename<*CR*> or **:w filename**<*CR*>

This will write the buffer to a file named "filename".

Quitting the Editor

When you have finished editing the file and you are ready to return to the UNIX System, enter the **quit** command (abbreviated **q**).

:quit<*CR*> or **:q**<*CR*>

This returns you to the UNIX System unless you forget to write the buffer to the file. The system will issue a message reminding you to write the file. A quick way to **write** and **quit** the edit editor is with the single command:

:wq<*CR*>

If you do not want to save the changes, enter the command:

:q!<*CR*>

This will quit edit and leave the file unchanged from the last write command.

Editing an Existing File

To edit the contents of an existing file named "file1," you begin by issuing the command:

```
$ edit file1<CR>
"file1" 150 lines, 4285 characters
:
```

This places a copy of the file in a buffer, and displays how many lines and characters are in the file. A colon ":" will then be displayed at the command line.

> ***Note:*** If you do not give a filename, edit will create a new file instead of editing the file you want.

After the file description and the colon ":" are displayed, enter a **1** on the command line followed by a carriage return. This will make the first line in the file the current line. The editing process is described throughout the remainder of this chapter.

The procedure for saving changes to the buffer is described in "Writing the Buffer Into the File." The procedure for quitting the editor is described in "Quitting the Editor."

DISPLAYING LINES IN THE FILE

When editing a file, you should always display the current line before making changes. This is important since most commands are executed on the current line. After making any changes, display the lines again to make sure you are happy with the changes. If you do not like the changes, you can use the **undo** command described in "RECOVERING LOST TEXT."

To display a line, all you need to do is depress the carriage return. This will display the current line in the editor. Each time you depress the carriage return, the next line is displayed, and it becomes the current line.

If you want to display the entire contents of the buffer, enter the command:

:1,$print**<*CR*> or **:1,$p<*CR*>

The "1" stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and the "p" is the command to print from line 1 to the end of the buffer. After displaying the buffer, the last line becomes the current line.

Occasionally, characters that do not appear on your terminal screen are contained in a line of text. These characters are normally called "control characters" because the control key was depressed when they were entered. To display all the characters in a line, including control characters, you can use the **list** command instead of the **print** command. For example:

:5,201list<*CR*> or **:5,201**<*CR*>

will display any character contained in that line regardless of what type it is. The **list** command executes exactly the same way the **print** command does.

MOVING AROUND IN THE FILE

Basic Movement Commands

Edit accepts "-" and "+" as movement commands. As you would expect, - moves the current line backwards and + moves the current line forwards. With these commands you can move to adjacent lines in the buffer.

You can move more than one line at a time by using numbers with the + and - commands. For example:

:-5<*CR*>

moves the current line backwards 5 lines from its current position and displays the line. Likewise,

:+25<*CR*>

moves the current line forwards 25 lines from its current position and displays the line. This makes it much easier to move to the line you want to work on. Another useful command is:

:$<*CR*>

that moves the current line to the last line in the buffer and displays the line.

Each line in the file has a line number associated with it, although they are not displayed. Edit allows you to move across large areas of the buffer by entering the line number and a carriage return. For example:

:43<*CR*>

makes 43 the current line and displays the line.

Forward and Backward Search Commands

If you are not sure where a line you want to change is, but you know an exact pattern of characters on the line, you can search for that pattern. **The pattern must be on one line.** The command line interprets the character "/" as meaning "search for this pattern." The *search* command "/" searches from your present position forward through the buffer for the first occurrence of the pattern. For example, if you know the pattern "learning to use edit" is somewhere in the buffer, you can find it by executing the command:

```
:/learning to use edit/p<CR>
```

This will make the line containing this pattern the current line and display the line. If you leave the "p" off the command, edit will still search for the pattern and make it the current line, but will not display the line. Always include the **p** as part of the search command. The pattern may be used more than once in the buffer.

If you execute a search, but edit cannot find the pattern, the message:

```
Pattern not found
```

will be displayed. This means the pattern you searched for is not in the buffer and the current line does not change. Check to see if you correctly entered the search command or if it included any characters with special meanings. (See Special Search Characters.)

The character "**?**" also executes a search when used on the command line. It works the same as the "/" search character, except that it searches backwards from your present position in the buffer.

Repeating Searches

When searching for a pattern, the first occurrence is not the one that you are actually looking for. You could repeat the search command, but there is a much easier way. The editor remembers the last search pattern entered. If you enter the command:

:**//**<*CR*>

a forward search will look for the remembered pattern. The backwards search command **??** will also repeat searches. The repeated search does not have to be the same type as the original search.

Global Searches

The edit editor also allows you to do global searches on the file. A global search is used to find all the occurrences of a specified pattern in a file. This type of search is useful when scanning for a pattern that occurs in several places. The two types of global searches that can be executed use the **g** and **v** commands.

The global search that uses the **g** command locates all the lines that contain a specified pattern. An example would be:

:**g/sample pattern/p**<*CR*>

This will search for and display all lines containing the words "sample pattern". The current line will be the last line displayed.

The global search that uses the **v** command locates all lines that **do not** contain a specified pattern. An example would be:

:**v/sample pattern/p**<*CR*>

This will search for and display all lines that do not contain the words "sample pattern". The current line will be the last line displayed.

Special Search Characters

Several characters have special meaning when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters and their use are explained below:

. The period matches any single character except the newline (carriage return) character. For example, if a line in your file contains the words "edit editor", or a pattern with any other character between "edit edit" and "r", you could find the line by entering the command:

:/f(BB/edit edit.r/<*CR*>

* The asterisk matches any repeated characters except the first ., \, [, or ~ in that group. For example, if a line in your file contains the pattern "the xxxx editor", you could search for the line by entering the command:

:/the x* editor/<*CR*>

[] Brackets are used to enclose a variable set of characters. For example, if a line in your file contains the patterns"file2", "file3", and "file4", you could search for the first occurrence of these patterns by entering the command:

:/file[2-4]/<*CR*>

$ The dollar sign is interpreted by the editor to mean "end of the line". It is used to identify patterns that occur at the end of a line. For example, if a line in your file ends in the pattern "last character" you could find the line by entering the command:

:/last character$/<*CR*>

^ The circumflex (caret) works like "**$**" except it looks for the pattern at the beginning of the line. For example, if a line in your

file begins with the pattern "First character" and you could find the line by entering the command:

:/^First character/<*CR*>

\ The backslash is used to cancel the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example, if a line in your file contains the pattern "This is a $" you could search for it by entering the command:

:/This is a \$/<*CR*>

The character $ will be searched for instead of interpreting it as meaning "end of the line."

To search for the characters ., *, \, [,], $, or ^, you must precede the characters with a backslash. You can also combine these special characters in one search command. For example, .* can be used to search for any string of characters.

MAKING CORRECTIONS TO THE FILE

There are several edit commands you can use to make corrections to a file. These commands are: append, input, delete, substitute, change, move, and copy.

Appending Text

The **append** command (abbreviated **a**) is used to input text in the buffer after the current line. It places edit in the *text input mode*. While in this mode, the colon prompt on the command line is not displayed. Anything you type, except a period on a line by itself, will be entered on lines of text in the buffer. To leave the *text input mode*, simply enter a period "." on a line by itself and depress the carriage return. Edit will then return to the command mode and display the command line prompt ":".

As previously discussed in "GETTING STARTED," the **append** command can be used to input text when the buffer is empty. The **append** command can also be used to input text anywhere in an existing file. The following steps outline how to append text to the current line:

1. Move to the place in the buffer where you want to append text. This can be done using movement commands or a search command. The line you select becomes the current line.

2. Enter the command:

   :**append**<*CR*> or :**a**<*CR*>

   The colon prompt will no longer be displayed on the command line.

3. Enter any text you like using as many lines as you like.

4. To leave the *text input mode* and return to the command mode, enter a period "." on a line by itself and depress the carriage return.

5. The command line prompt ":" will reappear. This indicates that you may enter another edit command.

Inserting Text

The **insert** command (abbreviated **i**) works similarly to the **append** command. The only difference is that text is inserted before instead of after the current line. To insert text in the buffer, enter:

:**insert**<*CR*> or :**i**<*CR*>

on the command line. You may now begin inserting text. To return to the command mode, simply enter a period "." on a line by itself and depress the carriage return. The command prompt will be displayed on the screen.

Changing Text

There may be instances when you want to delete one or more lines and insert new text in their place. This can be done easily with the **change** command (abbreviated **c**). The **change** command instructs edit to delete specified lines and then switch to *text input mode* to accept text to replace the lines. The number of lines you insert does not have to match the number deleted. For example, if you want to change the current line, enter:

:**change**<*CR*> or :**c**<*CR*>

The colon prompt will no longer be displayed. You may begin inserting as many lines of text as you want. To return to the command mode, enter a period "." on a line by itself and depress the carriage return.

If you want to replace lines 25 through 34 with some new text, you would enter:

:**25,34c**<*CR*>

Edit will respond with:

```
10 lines changed
```

The colon prompt will no longer be displayed. The procedure for entering text and for returning to the command mode is the same as for changing

one line. By default, if five or fewer lines are changed, edit will not display the number of lines being changed. (See *report* option given in Chapter 4.)

Deleting Text

The delete command (abbreviated **d**) is an easy command to execute. This command can also be disastrous if you are not careful when using it. To delete the current line, all you have to do is enter:

:delete<*CR*> or **:d**<*CR*>

This will delete the line and display the next line which becomes the current line.

> ***Note:*** You can use the **undo** command "**u**" to retrieve deleted lines as long as you have not executed any other commands that changed the buffer. (See "Recovering Lost Text.")

If you know the line number of a line you want to delete, you can enter the line number followed by **delete** or **d**. For example:

:15d<*CR*>

will delete line 15. You can also delete a range of lines by using commands such as **2,3d** to delete lines 2 and 3, or **2,8d** to delete lines 2 through 8.

When one or more lines are deleted, the numbers of all following lines are changed. When deleting different groups of lines from a file, it is easier to start with the higher line numbers and work toward the lower line numbers.

If you do not know the line number, you can search for the line and then delete it. Searching for text is discussed in "Forward and Backward Search Commands."

Substituting Text

To change any characters on an existing line without replacing the whole line, you can use the **substitute** command (abbreviated **s**). The substitute command searches for a specified pattern and then changes the pattern accordingly. The substitute command normally executes on the current line.

> ***Note:*** The **global** option can be used with the substitute command, but you must be careful. (See "Global Substitutes.")

Using the substitute command can sometimes be confusing to a novice user. However, if you think about the parts of the command, it is really easy. The format of the command is:

:s/*old-pattern***/***new-pattern***/p**

The "s" is the substitute command. The "/old-pattern/" tells edit to search the current line for the pattern. The "new-pattern/" tells edit what to substitute for "old-pattern" and "p" tells edit to display the new form of the current line. For example, if the current line is "Substituting is very confusing." and we want to change it to "Substituting is very easy.", we would use the command:

:s/confusing/easy/p<*CR*>

If you want to delete the word "very" from the new sentence, you could use the substitute command and not put a pattern where the new pattern should be.

:s/very //p<*CR*>

Your new sentence would be "Substituting is easy." Notice that a blank space was also removed because edit considers it a character.

Special Substitution Characters

All the special search characters given in "Special Search Characters" are also special characters in the search portion of substitution commands. However, there are two characters that have special meaning when used in the replacement portion of substitute commands. These characters are **&** and ~.

& The ampersand (**&**) character is used to save you from having to repeat the search portion of the substitute command when you are only adding characters. For example, if a line in your file contains the pattern "The game is tonight" and you wanted to change it to "The game is tonight at eight" you could use the following substitute command:

```
:s/The game is tonight/& at eight/p<CR>
```

~ The tilde (~) character works similar to the ampersand (**&**) character, except that it also repeats previous substitution commands.

To turn off the special meaning of the **&** and the ~ in the substitution command, it must be preceded by a backslash (\). These special characters will work with all types of substitution commands.

Global Substitutes

A global substitute is similar to a regular substitute, except that instead of only working on the current line it works on every line in the buffer. Before trying to understand global substitutes, be sure you understand regular substitutes. (See "Substituting Text.")

You must be careful when using global substitutes. There may be an occasion when you want to use a global substitute, but the pattern you want to search for may not be unique. If you think a line you want left alone might change, first do a global search and display all the lines. You may be able to find a pattern that is unique only to what you want changed. The format of a global substitute is as follows:

:g/*old-pattern***/s/***old-pattern***/***new-pattern***/gp**

In this example, the "**g**/*old-pattern*/" instructs edit to search for every occurrence of "old-pattern". The "**s**/*old-pattern*/*new-pattern*/" instructs edit to substitute "*new-pattern*" for every occurrence of "*old-pattern*". The "**g**" after the substitute command instructs edit to execute the substitution for every occurrence on each line if "*old-pattern*" is on a line more than one time. The "**p**" tells edit to display all the lines where substitutions were made.

> ***Note:*** The "**g**" at the end of the command should be omitted if you only want the first occurrence of the pattern on each line to change.

When using a global substitute command where the pattern you search for is the same as the pattern you want to change, you can use an abbreviated version of the command. For example, the command:

:g/*old-pattern***/s//***new-pattern***/gp**

will execute the same as the previous example. This saves you from having to input the pattern (old-pattern) in twice.

Edit also allows you to execute a global substitute within a range of lines. For example:

:35,75g/*old-pattern/s//new-pattern/gp*<*CR*>

would only do the substitutions from line 35 to line 75. All other lines would not be affected. This option allows you a much greater flexibility when using global substitutes.

If you decide you do not like what happened when you used the global substitute you have two choices. You can either try the **undo** command or you can quit the editor without writing the buffer into the file. (See "RECOVERING LOST TEXT.")

If you are not sure whether you want to keep the changes, you can write the buffer to a new file, and then either use the **undo** command or quit without writing. This way you can review both files before deciding which one to keep. (See "Writing the Buffer to Another File.")

Copying Text

Edit allows you to create a copy of specified lines in the buffer and insert them where you want by using the **copy** command. The original lines will remain unchanged. The **copy** command has the same format as the **move** command. For example:

:14,19copy<*CR*> or **:14,19co**<*CR*>

would create a copy of lines 14 through 19 and place it at the end of the buffer. The original lines 14 through 19 will stay the same. When the command has finished executing, the lines are automatically renumbered.

> ***Note:*** The abbreviation for the copy command is **co**. The **c** command is to change lines of text.

Moving Text

Edit allows you to move lines of text from one location to another in the buffer by using the **move** command (abbreviated **m**). You are allowed to move as many lines as you want. For example,

:2m15<*CR*>

would move line 2 to the position after line 15, and then renumber the lines. If you wanted to move a block of text, you could use the command:

:2,20m25<*CR*>

This would move lines 2 through 20 to the position after line 25.

When using the move command, you can specify the end of the buffer by using the **$** character instead of the line number. This is often much easier than looking to see what is the last line number. Two examples of using the **$** in a move command are:

:15,$m10**<*CR*> and **:1,20m$<*CR*>

The first example would move lines 15 through the end of the buffer to the position after line 10.

The second example would move lines 1 through 20 to the end of the buffer.

FILE MANIPULATION

Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, simply use the write command and the name of the file. For example:

:write filename<*CR*> or **:w filename**<*CR*>

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - use "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example:

:85,$w save<*CR*>

will write lines 85 through the end of the buffer to the file named *save*. The write command does not change the buffer.

Reading Another File Into the Buffer

The **read** command (abbreviated **r**) allows you to input the contents of another file into the buffer without destroying the text already there. To use the read command, first move to the line where you want the file appended. Then enter the read command using the following format:

:read filename<*CR*> or **:r filename**<*CR*>

Edit will append a copy of the file after the current line, and issue a message stating the name of the file, the number of lines, and the number of characters that were inserted.

Obtaining Information About the Buffer

Edit maintains a record of the current information about the buffer. To access this information, enter the **file** command (abbreviated **f**). Edit displays the filename, your current position, and the number of lines in the buffer. If the contents of the buffer have been changed since the last time the file was written, the editor will tell you that the file has been modified. It also displays what per cent of the way you are through the buffer. For example, enter the command:

:**f**<*CR*>

The computer will respond with a message such as:

```
"filename" [Modified] line 15 of 75 --20%--
```

Note: After you save the changes by writing the buffer to the file, the buffer will no longer be considered modified.

ISSUING UNIX SYSTEM COMMANDS

Edit allows you to execute a single UNIX System command by entering a command of the form:

:**!cmd**<*CR*>

where "cmd" represents the command you want to execute. The system will then execute the command. When finished, edit displays an **!** and then reissues the command line prompt ":". You can then continue editing or enter another UNIX System command.

If you need to execute more than one UNIX System command, enter the command:

:**sh**<*CR*>

When you are finished executing UNIX System commands, enter <CTRL **d**>. The editor will then display the message:

```
[Hit return to continue]
```

After depressing the carriage return, the editor will display the command line prompt.

Caution: Be sure to write the buffer into the file before escaping to the UNIX System. The editor will normally save the buffer, but it will issue a message to remind you.

RECOVERING LOST TEXT

Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed that changed the buffer. This enables you to restore the buffer after making an editing mistake. To execute the undo command enter:

:**undo**<*CR*> or :**u**<*CR*>

The undo command only works on commands such as append, insert, delete, change, move, copy, and substitute. You can also undo an undo if you decide to keep the change. Commands that do not affect the buffer such as: write, edit, and print cannot be undone.

Recovering Lost Files

If the system crashes, you can recover the contents of the buffer by using the **recover** command. **The recover command cannot be abbreviated.** You will normally receive mail the next time you log in, giving you the name of the file that was saved for you. You should then change to the directory containing the file being edited when the system crashed. Then access the file by entering:

:**edit filename**<*CR*>

replacing "filename" with the name of the lost file. Once in the editor, enter:

:**recover filename**<*CR*>

Recover is sometimes unable to save the entire contents of the buffer, so always check the contents of the saved buffer before writing it back to the original file.

If something goes wrong with the editor when you are using it, do not leave the editor. You may be able to save your work by using the **preserve** command (abbreviated **pre**). This saves the buffer as if the system had crashed.

If you are writing the buffer into the file and you get the message:

```
Quota exceeded
```

you have tried to use more disk space than you are allotted. When this happens, it is likely that only part of the buffer was written into the file. When this happens you should escape to the UNIX System using the **sh** command and remove some files you do not need. Then, try writing the file again. If this is not possible, enter the command:

:**preserve**<*CR*>

and then get help from the person who is administrating the system. **Do not quit the editor or your buffer will be lost.**

After using the preserve command and then finding the cause of your problem, you can use the **recover** command again.

Chapter 3

EX EDITOR

Chapter 3

EX EDITOR

INTRODUCTION

This chapter describes the ex editor used on the 3B2 Computer. Ex provides the advanced user a wide range of commands and options, but can also be used by new or casual users who only need a simple editor.

When using the ex editor, all commands must be entered on a command line. The command line is identified by a colon ":" on a line by itself. Commands entered on the command line can affect the line you are on in the file (current line), a specified set of lines, or the entire file.

Most ex editor command names are English words, that can be abbreviated. When an abbreviation conflict is possible, the more commonly used command has the shorter abbreviation. For example, since **substitute** is abbreviated by **s**, **set** is abbreviated by **se**.

The ex editor does not directly change the file being edited. Instead, it works on a copy of the file stored in a temporary memory location called the buffer. The edited file is not changed until you write the changes from the buffer to the edited file.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the ex editor, see the manual pages in the *AT&T 3B2 Computer User Reference Manual.*

CURRENT LINE DEFINITION

The term "current line" is referred to throughout this chapter. The current line is the line in the file you are now on. Each time you move to a different line in the file, that line becomes the current line. Whenever a command is given, the current line is used as a reference point. Any command that is not directed at any specific line is executed against the current line. You should always know what line is the current line, or you could mess up the file.

GETTING STARTED

The ex editor can be used to create a new file or to change an existing file. To execute ex, you must be logged onto the computer. After the **$** or # prompt is displayed, you can begin working with the ex editor.

Creating a New File

To create a new file, you will need to type **ex** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

```
$ ex filename<CR>
" filename" [New file]
:
```

If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. You will need to re-enter the command correctly.

If you entered the **ex** command without a filename, the editor will still create a new file. However, when you decide to write the file into memory you will be prompted for a filename. See "Writing the Buffer Into the File."

When the **ex** command is executed, a colon ":" is displayed. The colon identifies the command line and shows that the ex editor is ready to accept your input commands.

Entering Text

Most ex commands have two forms: a word that describes what the command does and an abbreviation of the word. You can use either form. Many beginners find the full command name easier to remember, but after some practice use the abbreviation. The command to input text is **append**, that may be abbreviated **a**. Enter **append** after the colon on the command line and then depress the carriage return.

:append<*CR*> or **:a**<*CR*>

The ex editor is now in the *text input mode* (append mode). The colon is no longer displayed on the command line, and this is your signal that you may begin entering lines of text. Anything that you type on your terminal, except a period on a line by itself, is entered into the buffer. If the error message:

```
Not an editor command
```

is displayed, check to see what you entered incorrectly and then enter the command again.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

Leaving the Input Mode

To leave the input mode, simply enter a period "." on a line by itself and depress the carriage return. This is the signal that you want to stop inputting text. After receiving a period on a line by itself, ex will re-enter the command mode and display the command line prompt ":".

The text just entered is now stored only in the buffer. If you wish, you can make changes to the text. Making changes is discussed throughout the remainder of this chapter.

Writing the Buffer into the File

The buffer is only temporary storage for the file. Now that you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

:write<*CR*> or :w<*CR*>

Ex will then copy the buffer into the file. If the file does not yet exist, a new file will be created and a message will be given indicating that it is a new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename, and give the number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

Ex must have a filename to use before it can write a file. Therefore, if you did not show the name of the file when you began the editing session, ex will issue the message:

No current filename

when you give the **write** command. If this happens, simply re-enter the **write** command and specify the filename. Here you would enter:

:write **filename**<*CR*> or :w **filename**<*CR*>

This will write the buffer to a file named "filename".

Quitting the Editor

When you have finished editing the file and you are ready to return to the UNIX System, enter the **quit** command (abbreviated **q**).

:quit<*CR*> or :q<*CR*>

This returns you to the UNIX System unless you forget to write the buffer

to the file. When this happens, you will receive a message reminding you to write the file. A quick way to **write** and **quit** the ex editor is with the single command:

:wq<*CR*>

If for some reason you do not want to save the changes, enter the command:

:q!<*CR*>

This will quit ex and leave the file unchanged from the last write command.

Editing an Existing File

To edit the contents of an existing file named "file1", you begin by issuing the command:

```
$ ex file1<CR>
"file1" 150 lines, 4285 characters
:
```

This places a copy of the file in a buffer, and displays how many lines and characters are in the file. A colon ":" will then be displayed, this is the command line.

> ***Note:*** If you do not give a filename, ex will create a new file instead of editing the file you want.

After the file description and the colon ":" are displayed, enter a **1** on the command line followed by a carriage return. This will make the first line in the file the current line. The editing process is described throughout the remainder of this chapter.

The procedure for saving changes to the buffer is described in "Writing the Buffer Into the File." The procedure for quitting the editor is described in "Quitting the Editor."

DISPLAYING LINES IN THE FILE

The procedures for displaying lines of a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

MOVING AROUND IN THE FILE

The procedures for moving around in a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

MAKING CORRECTIONS TO THE FILE

The procedures for making corrections to a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

FILE MANIPULATION

Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, use the write command and the name of the file. For example:

:write filename<*CR*> or **:w filename**<*CR*>

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - use "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example:

:85,$w save<*CR*>

will write lines 85 through the end of the buffer to the file named *save*. The write command does not change the buffer.

Reading Another File Into the Buffer

The **read** command (abbreviated **r**) allows you to input the contents of another file into the buffer without destroying the text already there. To use the read command, first move to the line where you want the file appended. Then enter the read command using the following format:

:read filename<*CR*> or **:r filename**<*CR*>

Ex will append a copy of the file after the current line, and issue a message

stating the name of the file, the number of lines, and the number of characters that were inserted.

Obtaining Information About the Buffer

Ex maintains a record of the current information about the buffer. To access this information, enter the **file** command (abbreviated **f**). Ex displays the filename, your current position, and the number of lines in the buffer. If the contents of the buffer have been changed since the last time the file was written, the editor will tell you that the file has been modified. It also displays what per cent of the way you are through the buffer. For example, enter the command:

:f<*CR*>

The computer will respond with a message such as:

```
"filename" [Modified] line 15 of 75 --20%--
```

Note: After you save the changes by writing the buffer to the file, the buffer will no longer be considered modified.

Read-Only Mode

If you want to look at a file you have no intention of changing, you can execute ex in the read-only mode. This mode protects you from accidentally overwriting the file. The read-only option can be set by using the **-R** command line option, by the **view** command line invocation, or by setting the read-only option. It can be cleared by setting the **noreadonly** mode. (See "OPTION DESCRIPTION.") It is possible to write, even while in the read-only mode, by writing to a different file or by using the **:w!** command.

Editing More Than One File

The ex editor is normally used to edit the contents of a single file, whose name is recorded in the current file. However, if you want to access another file without quitting ex, you can use the **e** command. For example:

:e **file2**<*CR*>

where "file2" is the name of the second file. This allows you easy access to both files. The current file is always the one currently being edited. The alternate file is the other file you have access to.

When you want to change to the alternate file, use the **e** command with the filename. Each time you use the **e** command to change files, the file you name becomes the current file and the file you leave becomes the alternate file.

When using the **e** command within the editor, normal shell expansion conventions such as "f*1" for "file1" may be used. In addition, the character **%** can be used in place of the current filename, and the character # in place of the alternate filename. For example:

:e #<*CR*>

will cause the alternate file to become the current file, and the current file will become the alternate file. This makes it easy to deal alternately with two files and eliminates the need for retyping the filename.

Editing Multiple Files and Using Named Buffers

When you have several files that you want to edit without actually leaving and re-entering the ex editor, you can list these files in your ex command. After receiving the command line prompt ":", you can edit file1 as described in this chapter. The remaining arguments are placed with the first file in the argument list. To display the current argument list, enter the **args** command on the command line. To edit the next file in the argument list, enter the **next** command on the command line. The following example shows how to enter three files with the ex command,

how to display the argument list, and how to change to the next file to be edited:

```
$ ex file1 file2 file3<CR>
3 files to edit
" file1"  xxx lines, xxxx characters
:args<CR>
[file1] file2 file3
:next<CR>
" file2"  xxx lines, xxxx characters
:
```

The argument list can be changed by specifying a list of filenames with the **next** command. These names are expanded with the resulting list of names becoming the new argument list, and ex edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, ex has a group of named buffers. These are similar to the normal buffer, except that only a limited amount of operations are available on them. The buffers have names **a** through **z**. It is also possible to refer to **A** through **Z**; the uppercase buffers are the same as the lowercase, but commands append to named buffers rather than replacing if uppercase names are used.

ISSUING UNIX SYSTEM COMMANDS

The procedure for issuing UNIX System commands from the ex editor is exactly the same as for the edit editor. Refer to the procedure given in Chapter 2.

RECOVERING LOST TEXT

Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed that changed the buffer. This enables you to restore the buffer after making an editing mistake. To execute the undo command enter:

:**undo**<*CR*> or :**u**<*CR*>

The undo command only works on commands such as; append, insert, delete, change, move, copy, and substitute. You can also undo an undo if you decide to keep the change. Commands that do not affect the buffer such as: write, edit, and print cannot be undone.

Recovering Lost Files

If the system crashes, you can recover the contents of the buffer by using the **recover** command. **The recover command cannot be abbreviated.** You will normally receive mail the next time you log in giving you the name of the file that was saved for you. You should then change to the directory containing the file being edited when the system crashed. Then, access the file by entering:

:**ex filename**<*CR*>

replacing "filename" with the name of the lost file. Once in the editor, enter:

:**recover filename**<*CR*>

Recover is sometimes unable to save the entire contents of the buffer, so always check the contents of the saved buffer before writing it back to the original file.

If something goes wrong with the editor when you are using it, do not leave the editor. You may be able to save your work by using the **preserve**

command (abbreviated **pre**). This saves the buffer as if the system had crashed.

If you are writing the buffer into the file and you get the message:

```
Quota exceeded
```

you have tried to use more disk space than you are allotted. When this happens, it is likely that only part of the buffer was written into the file. When this happens, you should escape to the UNIX System using the **sh** command and remove some files you do not need. Then, try writing the file again. If this is not possible, enter the command:

:preserve<*CR*>

and then get help from the person who is administrating the system. **Do not quit the editor or your buffer will be lost**.

After using the preserve command and then finding the cause of your problem, you can use the **recover** command again.

Recovering from Hang-ups and Crashes

If a hang-up signal is received and the buffer has been modified since it was last written, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second case) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines. To recover a file you can use the **-r** option. For example: If you were editing the file "filename", you should change to the directory where you were when the crash occurred, and give the command:

:ex -r filename<*CR*>

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

You will normally get mail from the system telling you when a file has been saved. The command **ex -r** will print a list of the files that have been saved for you.

Errors and Interrupts

When errors occur, ex rings the terminal bell (or flashes the terminal screen) and prints an error message. If the primary input is from a file, editor processing will end. If an interrupt signal is received, ex will display the message:

```
Interrupt
```

and returns to its command level. If the primary input is a file, ex will exit.

COMMENT LINES

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

MULTIPLE COMMANDS PER LINE

More than one command may be placed on a line by separating each pair of commands with a | character. However, global commands, comments, and the shell escape (**!**) must be the last command on a line, as they are not ended by a |.

OPTION DESCRIPTION

The options that you can set when using the ex editor are the same as for the vi editor. For a listing and a description of these options, see Chapter 4.

Ex Command Line Options

Instead of just entering the standard ex editor, you can use many options that are sometimes helpful. An example of a command line showing the proper format for using options is shown below.

ex [-][**-v**][**-t** *tag*][**-r**][**-w***n*][**-R**][+*command*] *filename*

These options are given in the following list, along with a short description of their function.

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

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

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

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

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

-R The **-R** option sets the read-only option at the start.

+*command*
An argument of the form +*command* tells the editor to begin by executing the specified command. If +*command* is omitted, ex will make the last line of the first file the current line.

filename

The *filename* arguments show the file to be edited. More than one filename can be given if several files are to be edited. See "FILE MANIPULATION" for further information on editing multiple files.

Chapter 4

VISUAL EDITOR (vi)

PAGE

Chapter 4

VISUAL EDITOR (vi)

INTRODUCTION

This chapter describes the visual editor (vi)* used on the 3B2 Computer. Vi is an interactive text editor that uses the screen of your terminal as a window into the file you are editing. Any changes you make to the file are reflected on the screen.

The vi editor does not directly change the file you are editing. Instead, it makes a copy of the file in a buffer and remembers the file's name. You do not affect the contents of the original file unless you write the changes made back into the original file.

Most vi commands move the cursor around in the buffer. A small set of operators such as **d** for delete and **c** for change alter the text in the buffer. Some of these commands and operators are combined to form operations

* The visual editor (vi) was developed by the Electrical Engineering and Computer Science Department of the University of California, Berkeley Campus.

such as "delete a word" or "change a paragraph." the mnemonic assignment of commands to keys makes the editor command set easy to remember and use.

There are normally several different vi editor commands you can use to get the same results. If you are trying to use vi for the first time, pick a few commands and use them until you no longer have to look them up. Then, gradually try using new commands. You will eventually find more efficient ways of doing the same things. The "CHARACTER FUNCTIONS SUMMARY" at the end of this chapter provides a complete list of vi commands.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the vi editor, see the manual pages in the *AT&T 3B2 Computer User Reference Manual.*

Relations Between vi and ex Editors

The vi editor is actually one mode of editing within the ex editor. When you are running vi, you can escape to the line-oriented editor (ex) by giving the **Q** command. Most ex commands can be invoked separately from vi by first entering a **:** and then the ex command. To execute the command, depress the carriage return.

In rare instances, an internal error may occur in vi. Here, you will get a diagnostic and be left in the command mode of ex. You can then save your work and quit, if you wish, by entering the command:

:x<*CR*>

If you would want to re-enter vi, you can enter the command:

:vi<*CR*>

Experienced users often mix their use of ex command mode and vi command mode to speed the work they are doing. The ex editor is described in Chapter 3.

GETTING STARTED

Defining Your Terminal

To use the vi editor, your 3B2 Computer needs to know what type of terminal you are using. The file **/etc/terminfo** contains the parameters of various terminals. Each type of terminal has a unique code assigned to it. To access the information in /etc/terminfo, you need to set the variable "TERM" to the code for your terminal and then export the variable. For example, to tell the computer you are using a TELETYPE* Model 5620 terminal, you would need to enter the following commands:

```
$ TERM=5620<CR>
$ export TERM<CR>
$
```

Setting Up Your Terminal Configuration

Vi will work on many types of video display terminals, and new terminal types can be added to a terminal description file. Before vi can be used on some terminals, the terminal setup parameters will need to be changed. The changes will vary depending on the terminal. For example, the TELETYPE Model 5410 terminal has a settable parameter called "RCVD'LF" that should be set to "INDEX". For instructions on how to change settable parameters, see the manual supplied with the terminal.

> ***Note:*** For more information on setting up your terminal, see the *AT&T 3B2 Computer Owner/Operator Manual.*

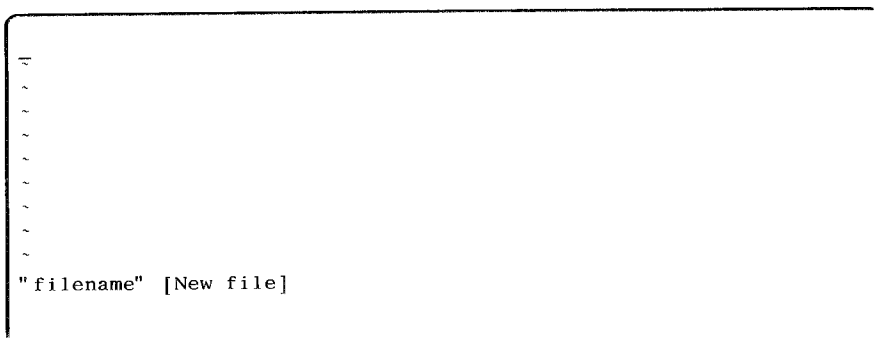
* Trademark of AT&T

Creating a New File

To create a new file, you will need to type **vi** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

$ vi filename<*CR*>

will create a file named "filename", clear the screen, and place the cursor at the top of the screen.



Once the vi command is executed, proceed to "Entering Text." If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. Reenter the command correctly.

Another problem that can occur is if you gave the system an incorrect terminal code (see GETTING STARTED). The editor may mess up your screen because vi sends control codes for one type of terminal to some other type of terminal. Here, enter the command:

:q<*CR*>

This should get you back to the UNIX System shell. Make sure you entered the correct terminal type and then try again.

Entering Text

To begin inputting text in a file, you must enter the *text insertion mode*. To do this you will need to enter either an **a**, an **i**, or an **o** (not followed by a carriage return). Since these are vi commands, they will not be displayed on the screen. After entering the *text insertion mode*, any characters you type are entered into the buffer.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

Leaving the Text Insertion Mode

To leave the *text insertion mode*, simply depress the *<ESC>* key. The computer response should be to backspace one character. This will return you to the command mode.

Returning to the command mode does not destroy the text in the buffer. You must return to the command mode to change any other type of editor command.

Writing the Buffer into the File

The buffer is only temporary storage for the file you are editing. Once you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

`:write`*<CR>* or `:w`*<CR>*

Vi will then copy the buffer into the file. If the file does not yet exist, a new file will be created, and a message will be given indicating that it is a new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename, and give the

number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

> ***Note:*** The **:w** command should be used every few minutes if you are happy with the changes you have made. This will keep you from losing all of your changes if you mess up the file or decide you do not like the changes you have made since the last time you wrote the file.

Vi must have a filename to use before it can write a file. If you did not show the name of the file when you began the editing session, vi will not write the file when you give the **write** command. If this happens, simply reissue the **write** command and specify the filename. Here you would enter:

:write filename<*CR*> or **:w filename**<*CR*>

This will write the buffer to a file named "filename".

Quitting the Editor

When you have finished working in the file and you are ready to return to the UNIX System, there are several methods you can use. If you have already written the buffer to the file, enter the command:

:q<*CR*>

To write the contents of the buffer back into the file you are editing and then quit the editor, enter the command:

:wq<*CR*>, **:x**<*CR*>, or **ZZ** (without depressing <*CR*>)

If for some reason you do not want to save the changes, enter the command:

:q!<*CR*>

This will quit vi and leave the file unchanged from the last write command.

Editing an Existing File

To edit the contents of an existing file named "filename", you begin by issuing the command:

$ vi filename<*CR*>

This places a copy of the file in a buffer. The screen should clear and the text of your file should appear on the screen. For example:

```
This is the first line in the file.
This is the second line in the file.
When a tilde (~) is displayed on a line by
itself, it normally means "end-of-the-file".
~
~
~
~
~
~
"filename" 4 lines, 161 characters
```

If the editor printed a "New file" message, you either gave the wrong filename or you are in the wrong directory. Here, you should enter **:q**<*CR*> to get you out of the editor. Check what directory you are in and try entering the command again.

If the editor makes a mess out of your screen, perhaps you gave the system an incorrect terminal type. Here, enter **:q**<*CR*> to get you back to the command level interpreter. If the editor does not respond, try sending an interrupt to it by depressing the <*DEL*>, <*BREAK*>, or <*RUBOUT*> key on your terminal. Then try entering **:q**<*CR*> again. Figure out what you did wrong and try again.

Once you have executed the vi command and you are in the buffer, you may begin moving the cursor around and change the file. Procedures for saving the changes to the buffer are described in "Writing the Buffer into the File." Procedures for quitting the editor are described in "Quitting the Editor."

Reading an Existing File

If you only want to use the editor to look at a file rather than to make changes, use the command:

```
$ view filename<CR>
```

This will set the read-only option that will prevent you from accidentally overwriting the file. Commands that move the cursor or change the file will execute. However, if you try to use the write command, you will receive the message:

```
"filename" File is read only
```

If you decide that you do want to change the file, you can still write the buffer to the file by entering the command:

```
:w!<CR>
```

MOVING AROUND IN THE FILE

The vi editor has many commands for moving around in a file. These commands allow you to: scroll through the file; search for a string of characters; or move from page to page, line to line, or character to character. Most of these commands can be preceded by a number to make movement in the file easier. A simple example would be to depress the 5 key and then the return key, this will move the cursor down 5 lines in the file.

> ***Note:*** Searching for a string of characters will not work when preceded by a number.

While reading through this chapter, you will notice that commands such as *<CTRL D>*, *<CTRL L>*, or *<CTRL H>* are used. This refers to commands where it is necessary to depress the control key and one other key at the same time. These are referred to as control characters. This may cause some confusion at first, but should not be a problem when you actually start using the vi editor.

> ***Note:*** When using the vi editor, be careful not to leave the *caps lock* key locked down. Capital letter commands are different from lowercase letter commands and you could accidentally mess up your file. If you do execute the wrong command, you can either use the undo command or quit without writing. (See "RECOVERING LOST TEXT.")

Scrolling and Paging Through the Screen

Scrolling and paging are two of the ways to move through a file. The main difference is that it is easier to read through a file while scrolling because the screen rolls up or down one line at a time. Paging causes the screen to be blanked each time a new page is displayed.

Scrolling

Scrolling allows you to continuously read through the file you are editing. *<CTRL D>* allows you to scroll down through the file until you release the keys. You can also scroll up through the file by using the *<CTRL U>* command. Some terminals cannot scroll up at all. Depressing *<CTRL U>* clears the screen and refreshes it with a line farther back in the file at the top.

If you want to see more of the file below where you are, you can depress *<CTRL E>* to expose one more line at the bottom of the screen, leaving the cursor where it is. The *<CTRL Y>* command is similar to the *<CTRL E>* command, except that it exposes one more line at the top of the screen.

Paging

Paging is a way to move forward or backward through a file a page at a time. The *<CTRL F>* command will move forward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file. The *<CTRL B>* command is similar to the *<CTRL F>* command, except that it will move backward a page.

Cursor Movements

Moving Within a Line

Some commands move the cursor one position at a time, and others move the cursor a word at a time. Preceding numbers may be used with all these commands. Keys that move the cursor a word at a time will wrap around the end of the line to the next line.

These commands are described in the following list:

b	Moves the cursor to the beginning of the previous word
e	Moves the cursor to the end of the next word
h	Moves the cursor one position to the left
l	Moves the cursor one position to the right
w	Moves the cursor to the beginning of the next word
B	Moves the cursor to the beginning of the previous word without stopping at punctuation marks
W	Moves the cursor to the beginning of the next word without stopping at punctuation marks
<CTRL H>	Control character that moves the cursor one position to the left
backspace	Moves the cursor one position to the left
spacebar	Moves the cursor one position to the right.

Note: On some terminals, the arrow keys will also move the cursor around on the screen. Most experienced users of vi normally prefer the **h**, **j**, **k**, and **l** keys because they are usually right beneath their fingers.

Moving To Different Lines

There are several commands you can use to move the cursor to a different line on the screen. All these commands except **H**, **L**, and **M** take preceding numbers and act on them. These commands are described in the following list:

j	Moves the cursor down
k	Moves the cursor up
RETURN	Moves the cursor to the first position on the next line
+	Moves the cursor to the first nonwhite position on the next line
–	Moves the cursor to the first nonwhite position on the previous line
H	Moves the cursor to the top line of the screen
M	Moves the cursor to the middle of the screen
L	Moves the cursor to the last line of the screen
<CTRL N>	Control character that moves the cursor down a line in the same column
<CTRL P>	Control character that moves the cursor up a line in the same column.

Moving Through a File

When working with a file containing text, it is often easier to work in terms of sentences, paragraphs, and sections. The following list describes some useful commands for working with text. Preceding numbers may be used with sentence and paragraph commands.

(Moves the cursor to the beginning of the previous sentence.

) Moves the cursor to the beginning of the next sentence.

> ***Note:*** A sentence is defined to end at a ., **!**, or **?**, and is followed by the end of the line or two spaces. Any number of), }, ", and ' closing characters may appear after the ., **!**, or **?**, and before the spaces or end of line.

{ Moves the cursor to the beginning of the previous paragraph.

} Moves the cursor to the beginning of the next paragraph.

> ***Note:*** A paragraph begins after each empty line and also at each paragraph macro specified in the *paragraphs* option. The .bp request is also considered to start a paragraph.

[[Moves the cursor to the beginning of the previous section.

]] Moves the cursor to the beginning of the next section.

> ***Note:*** Sections begin after each macro in the *section* option and each line with a form feed *<CTRL L>* in the first column. Section boundaries are always line and paragraph boundaries.

Searching Through the File

Another way to position yourself in the file is by having the editor search for a specific string of characters on one line. Type the character / followed by a string of characters for which you want to search. To execute the search, depress the carriage return. For example:

/character string<*CR*>

The editor will search from the current position toward the last line in the buffer for the first occurrence of "character string" on one line. The editor will also search backward if you use the **?** character instead of the / character.

If the character string you search for is not present in the file, the editor will display the message:

```
Pattern not found
```

on the last line of the screen and the cursor will return to its initial position.

A search will normally wrap around the end of the file and continue searching until the string is found or the position where the search started is reached. The wrap-around scan feature can be disabled by entering the command:

:set nowrapscan<*CR*> or **:set nows**<*CR*>

You can have the editor ignore whether letters are uppercase or lowercase in searches by entering the command:

:set ignorecase<*CR*> or **:set ic**<*CR*>

The command **:set noic**<*CR*> turns this option off.

Repeating Searches

If the first pattern found by the search command is not the one you were searching for, you can search for the next occurrence of the pattern by entering the command:

n

The **n** command works with forward and backward searches.

Another way to repeat a search without re-entering the entire command is to enter the search command character (/) or (**?**) followed by a carriage return. The direction of the search is determined by the search character you enter.

Special Search Characters

Several characters have special meanings when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters and their uses are explained below:

. The period matches any single character except the newline (carriage return) character. For example, if a line in your file contains the words "vi editor", or a pattern with any other character between "vi edit" and "r", you could find the line by entering the command:

:/vi edit.r/<*CR*>

* The asterisk matches any repeated characters except the first ., \, [, or ~ in that group. For example, if a line in your file contains the pattern "the xxxx editor", you could search for the line by entering the command:

:/the x* editor/<*CR*>

[] Brackets are used to enclose a variable set of characters. For example, if you have a file containing the patterns"file2", "file3", or "file4" you could search for the first occurrence of these patterns by entering the command:

:/file[2-4]/<*CR*>

$ The dollar sign is interpreted by the editor to mean "end of the line". It is used to identify patterns that occur at the end of a line. For example, if a line in your file ends in the pattern"last character", you could find the line by entering the command:

:/last character$/<*CR*>

^ The circumflex (caret) works like "**$**" except it looks for the pattern at the beginning of the line. For example, if a line in your file begins with the pattern "First character", you could find the line by entering the command:

:/^First character/<*CR*>

\ The backslash is used to cancel the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example, if a line in your file contains the pattern "This is a $", you could search for it by entering the command:

:/This is a \$/<*CR*>

The character $ will be searched for instead of being interpreted as meaning "end of the line".

To search for the characters ., *, \, [,], $, or ^, you must precede the characters with a backslash. You can also combine these special characters in one search command. For example, .* can be used to search for any string of characters.

Go To, Find, and Previous Context Commands

The go to (**G**) command allows you to move the cursor to a specific line in the file by using line numbers. For example:

32G

will move the cursor to line 32 in the file. If a line number is not used with the **G** command, the cursor will move to the last line in the file.

The find (**f***x*) command locates the next *x* character to the right of the cursor in the current line. For example, to find the next occurrence of the letter **t** you would enter the command:

ft

The **;** command repeats the last find command for the next instance of the same character. By using the **f** command and then a sequence of **;**'s, you can often get to a particular place in a line much faster than with a sequence of word motions or spaces. There is also an **F** command, that works like **f**, but searches backward. The **;** also repeats the **F** command.

The previous context " (two back quotes) command allows you to move back to the previous position in the file after a motion command, such as /, **?**, or **G**. This command is often more convenient than using the **G** command or performing a search because no advance preparation is required.

> ***Note:*** If you are near the last line of the file, and the last line is not at the bottom of the screen, the editor will place a ~ character on each remaining line to show the end of the file.

MAKING SIMPLE CHANGES

Inputting Text

The vi editor uses append, insert, and open commands to input text into a file. First, use the movement commands described earlier to move the cursor to the position in the file where you want to input text. Then depress the input command you want to use (see list below). Now any characters you type are entered into the buffer. If you are entering more than one line, depress a carriage return whenever you want to start a new line. You can also use the **autowrap** option discussed in "OPTIONS." To stop inputting text, depress the *<ESC>* key. All the commands for inserting text are described in the following list:

- **a** Appends everything you type after the current position of the cursor
- **A** Appends everything you type to the end of the line
- **i** Inserts everything you type before the current position of the cursor
- **I** Inserts everything you type before the first nonblank on the line (inserts before the first character on the line)
- **o** Opens a new line below the position of the cursor
- **O** Opens a new line above the position of the cursor.

Erasing Inserted Text

While inserting text, you can use the *<CTRL H>* or # character to backspace over (erase) the last character typed. To erase the text you have input on the current line, depress the @, *<CTRL X>*, or *<CTRL U>* characters. The *<CTRL W>* will erase a whole word and leave you after the space following the previous word. It is useful for quickly backing up in an insert.

While inserting text, the following conditions should be noted:

- When you backspace during an insertion, the characters you backspace over are not erased. The cursor moves backward and the characters remain on the display. This is often useful if you are planning to type in something similar. The characters disappear when you depress <*ESC*>. If you want to get rid of the characters immediately, depress <*ESC*> and then **a** again.

- You cannot erase characters that you did not insert, and you cannot backspace around the end of a line. If you need to back up to the previous line to correct something, depress the <*ESC*> key, move the cursor back to the previous line, and then make whatever corrections you want.

Continuous Text Input

When you are typing in large amounts of text, it is convenient to have lines broken near the right-hand margin automatically. You can cause this to happen by entering the command:

:set wm=10<*CR*>

This causes all the lines to be broken at a space at least ten columns from the right-hand edge of the screen. The number 10 can be replaced by any number you wish to use.

Joining Lines

If the editor breaks a line and you wish to put it back together, you can tell it to join the lines with the **J** command. You can give the **J** command a count of the amount of lines to be joined (such as **3J** to join 3 lines). The editor supplies white space, if appropriate, at the juncture of the joined lines and leaves the cursor at this white space. If you do not want white space, you can kill it with the **x** command.

Removing Text

The vi editor allows you to remove text from a file with several versions of the delete command. The commands listed below let you remove any object that the editor recognizes (characters, words, lines, sentences, and paragraphs). You do not need to use the *<CR>* or *<ESC>* keys with these commands. To delete more than one object at a time, you can use numbers with these commands. For example, **5dd** removes five lines of text.

dd Delete the current line.

dw Delete the current word.

db Delete the preceding word.

d) Delete the rest of the current sentence.

d(Delete the previous sentence if you are at the beginning of the current sentence, or delete the current sentence up to your present position if you are not at the beginning of the current sentence.

d} Delete the rest of the current paragraph.

d{ Delete the previous paragraph if you are at the beginning of the current paragraph, or delete the current paragraph up to your current position if you are not at the beginning of the current paragraph.

D Delete the rest of the text on the current line and leave the cursor on a blank line.

x Delete the current character.

X Delete the character before the cursor.

Note: To recover text that was accidentally deleted, see "Recovering Lost Text."

Changing Text

The vi editor allows you to use several different commands to change text in a file. With the commands listed below you can change any object that the visual editor recognizes (characters, words, lines, sentences, and paragraphs). All these commands, except **r**, are ended by depressing the *<ESC>* key. Numbers can be used with these commands to determine how many of the objects to change. For example, the command **2cw** removes two words and then changes to the input mode so new words can be inserted.

cc Change a whole line.

cw Change the specified word to the following word.

c) Change the rest of the current sentence.

c(Change the previous sentence if you are at the beginning of the current sentence, or change the current sentence up to your current position if you are not at the beginning of the current sentence.

c} Change the rest of the current paragraph.

c{ Change the previous paragraph if you are at the beginning of the current paragraph, or change the current paragraph up to your present position if you are not at the beginning of the current paragraph.

C Change the rest of the current line.

r Replace a character.

R Replace the following characters.

s Replace a character with a string.

S Replace the current line with a new line.

When you type a change command, the end of the text to be changed is marked with the **$** character to show that a change is now expected up to the **$** character. You are now placed in the insert mode so that anything you type is entered into the buffer. You end the insert mode by depressing *<ESC>*. To summarize, change commands in the visual editor deletes text objects and then places you in the insert mode.

The simplest change that you can make is to change one character. The **r** and the **s** commands can be used for this. If the character is incorrect and is to be replaced by a single character, correct the character by giving the **r***x* command, where *x* is the correct character. If the character is to be replaced by a string of characters, give the **s** (string) *<ESC>* command that substitutes a string of characters for the incorrect character. The **s** command can be preceded with a count of the amount of characters to be replaced.

You can also give a command like **cL** to change all the lines up to and including the last line on the screen, or **c3L** to change through the third line from the bottom line. Using the **c**/*string* command allows you to change characters from the current position to the first occurrence of the search string.

Note: To recover text that was accidentally changed, see "Recovering Lost Text."

COPYING TEXT

The Concept of Yank and Put

Vi provides a method of making a copy of text and placing this copy in another location in the file. This method is called "yank and put." The **y** operator yanks a copy of any specified object (word, line, sentence, or paragraph) into a specially reserved space called a register. The text can then be put back in the file from the register with the commands **p** and **P**; the **p** command puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text you yank forms a part of a line or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before the cursor if you use **P**). If the yanked text forms whole lines, whole lines will be put back without changing the current line.

The **Y** command is used to create a copy of a line. The cursor can then be moved to any character on another line, and the **p** used to place the yanked line following the current line. The **P** command places the copied line above the current line. The **YP** command makes a copy of the current line and places it before the current line. The cursor is placed on the first character of this copy. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also create a copy of the current line and place it after the current line. You can give **Y** a count of lines to yank and thus duplicate several lines.

Vi has a single unnamed register where the last yanked text is saved. Each time a yank command is performed that uses the unnamed register, the previous yank command is lost. To prevent the loss of this text, the editor has a set of named registers [(**a**) through (**z**)] that can be used to save copies of text. The general format of the yank command using named registers is

" *x***y**object

where *x* is the name of the register [(**a**) through (**z**)] into which an object is copied. The following procedure copies a line into a new location in a file.

1. Enter the command:

   " ayy

   This yanks a line from where the cursor is into the named register *a*.

2. Move the cursor to the eventual resting place of this line.

3. Enter the command:

   " ap or **" aP**

   This puts the line at the new location.

Copying Objects

The yank and put commands can be used to copy characters, words, lines, sentences, or paragraphs. All the object commands can be preceded by a number, that allows you to copy more than one object. This is especially useful when copying characters. Each of the following objects should be experimented with so you understand what happens during a yank and put.

Characters can be copied by typing the yank command and then typing the following object commands:

spacebar	Yanks one character in forward direction.
backspace	Yanks one character in backward direction.
h	Yanks one character in backward direction.

l	Yanks one character in forward direction.
f*x*	Yanks all characters from cursor up to *x* in forward direction.
F*x*	Yanks all characters from cursor up to *x* in backward direction.
t*x*	Yanks all characters from cursor up to and including *x* in forward direction.
T*x*	Yanks all characters from cursor up to and including *x* in backward direction.

Words can be copied by typing the yank command and then typing the following objects:

w	Yanks one word in forward direction (punctuation counts as word).
W	Yanks one word in forward direction (punctuation does not count as word).
b	Yanks one word in backward direction (punctuation counts as word).
B	Yanks one word in backward direction (punctuation does not count as word).
e	Yanks one word in forward direction up to last character in word (punctuation counts as word).

Lines can be copied (in addition to **yy** and **Y**) by typing the yank command and then typing the following objects:

$	Yanks one line from cursor to end of line.

<*CR*> Yanks one line plus line cursor is on in forward direction.

j Yanks one line plus line cursor is on in forward direction.

\+ Yanks one line plus line cursor is on in forward direction.

k Yanks one line plus line cursor is on in backward direction.

\- Yanks one line plus line cursor is on in backward direction.

H Yanks line cursor is on through top line on screen.

M Yanks line cursor is on through middle line on screen.

L Yanks line cursor is on through bottom line on screen.

G Yanks line cursor is on through last line in file. If a number precedes **G**, yanks through that line in forward or reverse direction.

/ Yanks from where cursor is up to "searched for" string in forward direction.

? Yanks from where cursor is through "searched for" string in backward direction.

Sentences can be copied by typing the yank command and then typing the following objects:

) Yanks from cursor to end of sentence in forward direction.

(Yanks from cursor to beginning of sentence in reverse direction.

Paragraphs can be copied by typing the yank command and then typing the following objects.

} Yanks from cursor to end of paragraph in forward direction.

{ Yanks from cursor to beginning of paragraph in reverse direction.

MOVING TEXT

The blocks of text that can be moved around in the file are: characters, words, lines, sentences, and paragraphs. To move blocks of text from one location to another, use the following procedure:

1. Delete (or change) the information you need to move with one command. It will be saved in an area and appointed to a register.

2. Move the cursor to the location you wish to insert the text just deleted and put it back in the file with the commands **p** or **P**. The **p** command puts the text after or below the cursor while **P** puts the text before or above the cursor. An example of a delete and put command is:

   xp

   The **x** deletes the character the cursor is on; the cursor moves to the next character to the right. The **p** puts the deleted character back following the character the cursor is on. The result is two characters have swapped positions.

3. If the text you delete forms a part of a line or is an object such as a sentence that partially spans more than one line, then when you put the text back it will be placed after the cursor (or before if you use **P**). If the deleted text forms whole lines, they will be put back as whole lines without changing the current line.

4. You may wish to place the text you are to move into a specific location. The editor has a set of named registers [**(a)** through **(z)**] that you can use to save copies of text. The general format of the delete command using named registers is

   " *x*delete object
   or
   " *x*change object

   where (x) is the name of the register [**(a)** through **(z)**] into which an object is deleted.

The following procedure moves a line to a new location in a file.

1. Enter the command:

   " add

   This deletes the line the cursor is on into the named register **(a)**.

2. Move the cursor to the eventual resting place of this line.

3. Enter the command:

   " ap or **" aP**

   This puts the line at the new location. You can also do the same with a change operation. After the new text is entered and the *<ESC>* key pressed, the deleted text can be "put" at another location in the file.

GLOBAL COMMANDS

Global Searches

When you need to locate all the occurrences of a specific pattern on a line in your file, the global command (**:g**) and a search command (/ or **?**) can be used. The global search command can be used in any of the following formats:

(1) **:[m],[n]g/text**
(2) **:[m],[n]g/text/p**
(3) **:[m],[n]g/text/nu**

The **[m]** represents the line number where the search will start. The **[n]** represents the line number where the search will stop, or **$** that causes the search to continue to the end of your file. If no numbers are entered, all lines in the file will be searched.

- When (1) is entered, the cursor will move to the last occurrence of "**text**".
- When (2) is entered, all the lines containing "**text**" are displayed on the screen.
- When (3) is entered, all the lines containing "**text**" are displayed on the screen. Line numbers will be displayed with each line.

In global searches, a **?** substituted for the / will have the same affect. The special characters described in "Special Search Characters" can be used in global search commands.

Global Substitutes

The global substitute command can be used when the same change needs to be made in several places in the file. The command can be executed against a range of lines or against the whole file. The following formats can be used for global substitutes:

(1) **:[m],[n]g/text/s//newtext**
(2) **:[m],[n]g/text/s//newtext/p**
(3) **:[m],[n]g/text/s//newtext/c**

The **[m]** represents the line number where the search will start. The **[n]** represents the line number where the search will stop. **$** can be used to represent the end of your file. If no numbers are entered, all lines in the file will be searched.

- When (1) is entered, "**newtext**" will be substituted for "**text**" at the first occurrence on each line requested in the command. The cursor will be placed at the last occurrence of the changed "**newtext**".

- When (2) is entered, "**newtext**" will be substituted for "**text**" at the first occurrence on each line requested in the command. The lines containing all occurrences of "**newtext**" substitutions are displayed on the screen.

- When (3) is entered, you are in a "prompt" mode. The "prompt" mode will allow you to decide if you want to make the substitution. The line with the first occurrence of "**text**" is displayed at the bottom of the screen. Each of the characters in "**text**" will be replaced by ^ (caret). If you type a **y** followed by a *<CR>*, "**newtext**" will be substituted for "**text**" in the file. The next line containing "**text**" will then be displayed with ^'*s replacing* "**text**". If you decide not to make the substitution, type a *<CR>* and the next line with "**text**" will be displayed. The line displayed may appear as follows:

The ^^^^ of this sentence needs to be changed.

The special characters described in "Special Search Characters" can be used in the search part of the global substitution command.

REPEATING ACTIONS WITH THE . COMMAND

Vi provides a timesaving command, called the "dot" command. The "dot" command allows you to repeat the last command that changed the buffer by placing the cursor at the location you wish to repeat the command and entering a:

.

The actions that can be repeated using the . command are append, insert, open, delete, change, and put. An example of how to use the dot command would be to insert a line of text in a file and then depress the <*ESC*> key. Then move the cursor to a different location in the file and enter a . "dot". Vi will repeat the previous insert command and insert the line of text here also.

If you want to place text at another location that is in a named register after doing a put, you can save time by using the . command. However, if you executed a put command that is associated with an "unnamed" register, the . command should not be used. This is because the text in the unnamed register may not be the same.

FILE MANIPULATION

Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, simply use the write command and the name of the file. For example:

```
:write filename<CR> or :w filename<CR>
```

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example,

```
:85,$w save<CR>
```

will write lines 85 through the end of the buffer to the file named *save*.

The write command does not change the buffer. The editor will display the name of the file "save" that you have copied into, the number of lines, and the number of characters entered into the file "save". If no numbers are entered, the entire file you are in will be copied to the filename entered.

Sometimes it is necessary to append information onto the end of a file that already exists. For example, if you wanted to append several lines to the file "save", you could use the command:

:12,25w >>**save**<*CR*>

The editor will display the name of the file "save", the number of lines, and the number of characters added to the file.

Reading Another File into the Buffer

While using **vi**, it may be necessary to copy another file into the file you are editing. This can be done using the **:r** command. To copy a file into your file, enter the **:**, a line number that you desire the new text to follow, the **r**, and the name of the file you wish to copy. The format for this command is:

:[n]read filename<*CR*> or **:[n]r filename**<*CR*>

[n] can be any line number in your file. If you enter a **0**, the copied file will be added before line 1 in your file. If you enter a **$**, the copied file will be added to the end of your file.

When the file is added, the editor will display at the bottom of the screen the name of the file you copied, the number of lines in that file, and the number of characters it contains. If you do not enter a number in the above command, the file to be copied will be added following the line your cursor was on when you entered the command. For example, if you wish to write a file named "test" to follow line 10 in your file, enter the command:

:10r test<*CR*>

Reading the Output From UNIX System Commands into the Buffer

There are two commands that you can use to put the output from a UNIX System command into a file. The only difference between the two commands is that one inserts the text between lines and the other replaces the current line with the text.

To insert the output from a UNIX System command between two lines, position the cursor where you want the text and execute the command:

:r !cmd<*CR*>

where "cmd" is the UNIX System command. The inserted text will be displayed on the screen. This command will also allow you to use a line number instead of positioning the cursor where you want the text inserted.

If you want to replace a line in the buffer with the output of a UNIX System command, position the cursor on that line and execute the command:

!!cmd<*CR*>

where "cmd" is the UNIX System command. Only the current line will be replaced by the inserted text. The inserted text will be displayed on the screen.

Changing Files in the Editor

The vi editor is normally used to edit the contents of one file, whose name is recorded as the current file. However, you can edit a different file without leaving the editor by using the command:

:e filename<*CR*>

where "filename" is replaced by the name of the file to which you want to change. This command allows you easy access to both files, because vi does not have to be executed again.

When you are accessing two files, the file you are editing is always considered the current file, and the other file is considered the alternate file. When you want to change to the alternate file, use the **e** command with the filename. Each time you use the **e** command to change files, the file you name becomes the current file and the file you leave becomes the alternate file.

When using the **e** command within the editor, normal shell expansion conventions, such as "f*1" for "file1", may be used. In addition, the character **%** can be used in place of the current filename and the character # in place of the alternate filename. For example:

:e #<*CR*>

will cause the alternate file to become the current file and the current file will become the alternate file. This makes it easy to deal alternately with two files and eliminates the need for retyping the filename.

If you have not written the current file, the editor will display the message:

```
No write since last change (:edit! overrides)
```

and delay editing the other file. You can either give the **:w** command to write the file or **:e! filename** if you want to discard the changes to

the current file and begin editing the next file. To have the editor automatically save the changes, you should include **set autowrite** in your EXINIT and use the **:n** command instead of the **:e** command.

If you want to edit the same file (start over), give the **:e!** command. These commands should be used carefully because once the changes are discarded they cannot be recovered.

Editing Multiple Files and Using Named Buffers

When you have several files that you want to edit without actually leaving and re-entering the vi editor, you can list these files in your vi command. For example, if you enter the command:

vi file1 file2 file3<*CR*>

the computer will respond with a message such as:

```
3 files to edit
"file1" xxx lines, xxxx characters
```

The current file "file1" can now be edited. The remaining arguments are placed with the first file in the argument list. To display the current argument list, enter the command:

:args<*CR*>

The computer will respond with the message:

```
[file1] file2 file3
```

The next file in the argument list may be edited by entering the command:

:next<*CR*> or **:n**<*CR*>

If you have already written the buffer to the file, the computer will respond with a message such as:

```
"file2" xxx lines xxxx characters
```

If you use the **next** command regularly, you may want to set the **autowrite** option.

The argument list can be changed by specifying a list of filenames with the **next** command. These names are expanded with the resulting list of names becoming the new argument list, and vi edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, vi has a group of named buffers. These are similar to the normal buffer, except that only a limited amount of operations are available on them. The buffers have names **a** through **z**. It is also possible to refer to **A** through **Z**; the uppercase buffers are the same as the lowercase, but commands append to named buffers rather than replacing if uppercase names are used.

Read-Only Mode

If you want to look at a file that you have no intention of changing, you can execute vi in the read-only mode. This mode protects you from accidentally overwriting the file. The read-only option can be set by using the **-R** command line option, by the **view** command line invocation, or by setting the read-only option. It can be cleared by setting the **noreadonly** mode. (See "OPTIONS.") It is possible to write, even while in the read-only mode, by writing to a different file or by using the **:w!** command.

Obtaining Information about the Buffer

You can determine the state of the file by using the *<CTRL G>* command. The editor will show you the name of the file, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer that the cursor is located. A sample response would be:

```
"filename" [Modified] line 1048 of 3096 --33%--
```

Note: After you save the changes by writing the buffer to the file, the buffer is no longer considered modified.

ISSUING UNIX SYSTEM COMMANDS

Vi allows you to execute UNIX System commands by entering commands of the form

:!cmd<*CR*>

where "cmd" represents the command you want to execute. Once the command has executed, the computer will issue the message:

```
[Hit return to continue]
```

You can then depress the carriage return to continue editing or enter the **:!** command to issue another UNIX System command.

If you need to execute more than one UNIX System command enter:

:sh<*CR*>

The computer will respond with the shell prompt (**$**). When you have finished executing UNIX System commands, enter a <*CTRL d*>. This will return you to the vi editor.

Caution: Be sure to write the buffer into the file before escaping to the UNIX System. The editor will normally save the buffer, but it will issue a message to remind you.

RECOVERING LOST TEXT

Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed. Undo can often rescue the buffer from a disastrous mistake. To execute the undo command enter:

:undo<*CR*> or **:u**<*CR*>

The undo command only works on commands that change the buffer, such as — append, insert, delete, change, move, copy, and substitute. You can also undo an undo command if you decide to keep the change. Commands such as write, edit, and print cannot be undone.

The **U** command works like the **u** command, except that it returns the current sentence to its original state.

Recovering Lost Lines

You might have a serious problem if you delete text and then regret that it was deleted. The editor saves the last nine deleted blocks of text in a set of numbered registers [1 through 9]. (Text consisting of a few words is not saved in these registers.) You can get the *n*th previous deleted block of text back into your file by the command:

" *n***p**

The " tells that a register name is to follow, *n* is the number of the register you wish to try, and **p** is the put command that puts text in the register after the cursor. If this does not bring back the text you wanted, type **u** to undo this command and repeat the command using a different numbered register. You can repeat this procedure until you find the correct deleted text.

An easier way to search for the correct register can be to use the . (dot) command to repeat the put command. In general, the .

command will repeat the last change. As a special case, when the last command refers to a numbered text register, the . command increments the number of the register before repeating the put command. Thus, a sequence of the form

```
"1pu.u.u
```

will, if repeated long enough, show all the deleted text that was saved. Omit the **u** commands and place all the text in the numbered registers at one location. Stop after any . command to put just the then-recovered text at one location. The command **P** can also be used rather than **p** to put the recovered text before instead of after the cursor.

Recovering Lost Files

If the system crashes, you can recover most of the work you were doing. You will normally receive mail the next time you log in giving you the name of the file that has been saved for you. To recover the file, change to the directory where you were when the system crashed and give a command of the form:

```
$ ex -r filename<CR>
```

replacing "filename" with the name of the file that you were editing. This will recover your work almost at the point where you left off.

You can get a listing of the files that are saved for you by giving the command:

```
$ ex -r<CR>
```

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. Therefore, you can get an older saved copy back by first recovering the newer copies.

For the "recover lost file" command to work, vi must be correctly installed and the mail program must exist to receive mail.

MARKING LINES

The vi editor allows you to mark lines in the file with single letter tags and return to these marks later by naming the tags. For example, mark the current line with an **a** by entering the command:

ma

Then, move the cursor to a different line using any commands you like and enter the command:

'a

The cursor will return to the place you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the marked line. Here, use the form *'x* rather than *`x*. Used without an operator, *'x* will move to the first nonwhite character of the marked line. The '' moves to the first nonwhite character of the line containing the previous context mark ''.

WORD ABBREVIATIONS

Word abbreviation allows you to type a short word and have it expanded into a longer word or words. The commands are:

:abbreviate (or **:ab**)
and
:unabbreviate (or **:una**)

and have the same syntax as **:map**. For example:

:ab ecs Engineering and Computer Sciences<*CR*>

causes the word "ecs" to always be changed into the phrase "Engineering and Computer Sciences." Word abbreviation is different from macros in that only whole words are affected. If "ecs" were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke as it should be with a macro.

ADJUSTING THE SCREEN

If the screen image is messed up because of a transmission error to your terminal or because some program other than the editor wrote to your terminal, use the <*CTRL L*> command to refresh the screen.

If you want to place a certain line on the screen at the top, middle, or bottom of the screen, you can position the cursor to that line and use the **z** command followed by its argument. The following list describes the three possible uses of the **z** command:

zz Places the line at the top of the screen

z. Places the line at the center of the screen

z- Places the line at the bottom of the screen.

LINE REPRESENTATION IN THE DISPLAY

The editor folds long logical lines onto many physical lines in the display. Commands that advance lines, advance logical lines and will skip over all the segments of a line in one motion. The | command moves the cursor to a specific column and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line that is more than 80 columns long.

The editor puts only full lines on the display. If there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, the editor will often clear just the lines to @ to save time (rather than rewriting the rest of the screen). You can always maximize the information on the screen by giving the *<CTRL R>* command.

Line Numbers

Vi allows you to place line numbers before each line on the display. To set the line number option, enter the command:

```
:set nu<CR>
```

To remove the line number option, enter the command:

```
:set nonu<CR>
```

List All Characters on a Line

You can have tabs represented as ^I and the ends of lines shown with $ by entering the command:

```
:set list<CR>
```

To remove the display of tabs and ends of lines enter the command:

```
:set nolist<CR>
```

Lines consisting of only the ~ character are displayed when the last line of the file is in the middle of the screen. These represent physical lines that are past the logical end of the file.

MACROS

The **vi** editor allows you to create macros so that when you enter a single keystroke the editor will act as though you had entered a longer sequence of keystrokes. You can do this if you find yourself typing the same sequence of commands (keystrokes) repeatedly.

There are two types of macros:

- One type, you put the macro body in a buffer register such as *x*. You can then type **@x** to invoke the macro. The @ may be followed by another @ to repeat the last macro.

- You can use the **map** command from the **vi** editor (typically in your **EXINIT**) with a command of the form:

  :map *lhs rhs<CR>*

  mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key). It must be entered within 1 second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and **vi** will wait for you to finish before it echoes anything). The *lhs* can be no longer than ten characters, the *rhs* no longer than 100. To get a space, tab, or newline into *lhs* or *rhs* you should escape them with a *<CTRL v>* (it may be necessary to double the *<CTRL v>* if the map command is given inside **vi** rather than in **ex**). Spaces and tabs inside the *rhs* need not be escaped. To make the **q** key write and exit the editor, enter:

  :map q :wq*<CTRL v><CTRL v><CR> <CR>*

  this means that whenever you type **q**, it will be as though you

had typed **:wq**<*CR*>. A <*CTRL v*> is needed because without it the <*CR*> would end the : command rather than becoming part of the *map* definition. There are two <*CTRL v*>'s because from within **vi**, two <*CTRL v*>'s must be typed to get on. The first <*CR*> is part of the *rhs*, the second ends the : command.

Macros can be deleted with

:unmap *lhs*

If the *lhs* of a macro is **#0** through **#9**, the particular function key is mapped instead of the 2-character # sequence. So that terminals without function keys can access such definitions, the form **#x** will mean function key *x* on all terminals (and need not be typed within 1 second). The character # can be changed by using a macro in the usual way:

:map <*CTRL v*><*CTRL v*><*CTRL i*> #

to use tab, for example. This will not affect the *map* command, that still uses #, but affects the invocation from *visual* mode.

The **undo** command will reverse all the changes made by a macro call as a unit.

Placing an **!** after the word map causes the mapping to apply to *text input* mode rather than *command* mode. Thus, to arrange for <*CTRL t*> to be the same as four spaces, type

:map <*CTRL t*><*CTRL v*>␣␣␣␣

where ␣ is a blank. The <*CTRL v*> is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

OPTIONS

Setting Options

There are three kinds of options: numeric, string, and toggle. Numeric and string options are set by a statement of the form:

:set *option=value<CR>*

Toggle options can be set or not set by statements of the forms:

:set *option<CR>*
and
:set no*option<CR>*

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

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

:set*<CR>*

or the value of a single option with the command:

:set *option* **?***<CR>*

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

:set all*<CR>*

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

:se ai aw nu*<CR>*

Options set by the **set** command last only while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be done by creating a list of **ex** commands that are to be run every time you start **ex**, **edit**, or **vi** (all commands that start with : are **ex** commands). A typical list includes a **set** command and possibly a few **map** commands. Since it is advisable to get these commands on one line, they can be separated with the | character; for example:

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

this establishes the **set** command options *autoindent*, *autowrite*, *terse*, makes @ delete a line (the first **map**), and makes # delete a character (the second **map**). One way to have the commands execute every time you enter the vi editor is to put the line in the file *.exrc* in your home directory. Another way to execute the commands automatically is to place the string in the variable **EXINIT** in your environment. Using the shell, put these lines in the file *.profile* in your home or working directory:

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

Of course, the particulars of the line would depend on the options you want to set.

List of Options

The editor has a set of options that can be useful. Some of these options have been mentioned earlier. They are as follows:

autoindent, ai (default: noautoindent)
Can be used to ease the preparation of structured program text. At the beginning of each **append**, **change**, or **insert** command or when a new line is opened or created by an *append*, *change*, *insert*, or *substitute* operation, the editor looks at the line being appended after, the first line changed, or the line inserted before, and calculates the amount of white space at the start of the line.

Autoindent then aligns the cursor at the level of indentation so determined.

If the user then types in lines of text, the lines will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligning with the first nonwhite character of the previous line. To back the cursor to the preceding tabstop, type *<CTRL d>*. The tabstops (going backwards) are defined as multiples of the *shiftwidth* option. You cannot backspace over the indent except by sending an end-of-file with a *<CTRL d>*.

Specially processed in this mode is a line with no character added to it, that turns into a completely blank line (the white space provided for the *autoindent* is discarded). Also, specially processed in this mode are lines beginning with a ^ and immediately followed by a *<CTRL d>*. This causes the input to be repositioned at the beginning of the line while retaining the previous indent for the next line. Similarly, a **0** followed by a *<CTRL d>* repositions at the beginning without retaining the previous indent.

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

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

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

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

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

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

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

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

flash, fl (default: flash)
Errors or illegal inputs respond by flashing the screen instead of ringing the bell in the terminal. On terminals that do not have flash capability, the bell will still ring.

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

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

list (default: nolist)
All printed lines will be displayed showing hidden characters such as tabs and end-of-lines.

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

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

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

optimize, opt (default: optimize)
Throughput of text is expedited by setting the terminal not to do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

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

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

readonly (default: noreadonly)
Sets the editor so you cannot accidentally change the file.

redraw (default: noredraw)
The editor simulates (using great amounts of output) an intelligent terminal on a dumb terminal (for example, during insertions in *visual*, the characters to the right of the cursor position are refreshed as each input character is typed). This option is useful only at high speeds.

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

report (default: report=5)
Specifies a threshold for feedback from commands. Any command that changes more than the specified amount of lines will provide feedback on the scope of its changes. For commands such as **global**, **open**, **undo**, and **visual**, that have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus, notification is suppressed during a **global** command on the individual commands performed.

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

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

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

shiftwidth, sw (default: shiftwidth=8)
Gives the width a software tabstop used in reverse tabbing with *<CTRL d>* when using *autoindent* to append text and by the shift commands.

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

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

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

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

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

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

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

timeout (default: notimeout)
Set a time limit for the execution of an editor command.

ttytype=
Terminal type defined to system for visual mode. Can be defined before entering visual editor by TERM=type.

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

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

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

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

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

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

CHARACTER FUNCTIONS SUMMARY

This summary shows the uses that the **vi** editor makes of each character. Characters are presented in their order in the ASCII character set: control characters first, most special characters, digits, uppercase characters, and then lowercase characters.

Each character is defined with a meaning it has as a command and any meaning it has during an insert. If it has meaning only as a command, then only this is discussed. Usually, uppercase and lowercase *<CTRL>* characters do the same action.

<CTRL @> Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted; and the insert ends. Only 128 characters are saved from the last insert; if more characters have been inserted, the mechanism is not available. A *<CTRL @>* cannot be part of the file owing to the editor implementation.

<CTRL a> Unused.

<CTRL b> Backward window. A count specifies repetition. Two lines of continuity are kept, if possible.

<CTRL c> Unused.

<CTRL d> As a command, it scrolls down a half window of text. A count gives the amount of (logical) lines to scroll and the count is remembered for future *<CTRL d>* and *<CTRL u>* commands. During an insert, it backtabs over *autoindent* white space at the beginning of a line. This white space cannot be backspaced over.

<CTRL e> Exposes one more line below the current screen in the file, leaving the cursor where it is, if possible.

<CTRL f> Forward window. A count specifies repetition. Two lines of continuity are kept, if possible.

<CTRL g> Equivalent to **:f** *<CR>*, printing the current filename, whether it has been modified, the current line number, the number of lines in the file, and the percent of the way through the file.

<CTRL h> (BS)
Same as **left arrow** (see **h**). During an insert, it eliminates the last input character backing over it but not erasing it. The character remains so you can see what you typed if you wish to type something slightly different.

<CTRL i> (TAB)
Not a command character. When inserted, it prints as some amount of spaces. When the cursor is at a tab character, it rests at the last of the spaces that represent the tab. The spacing of tabstops is controlled by the *tabstop* option.

<CTRL j> (LF)
Same as **Down arrow**. It moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include **j** and *<CTRL n>*.

<CTRL k> Unused.

<CTRL l> The ASCII form feed character that causes the screen to be cleared and redrawn. It is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.

<CTRL m> (*<CR>*)
A carriage return advances to the next line, to the first nonwhite position in the line. Given a count, it advances that many lines. During an insert, a *<CR>* causes the insert to continue onto another line.

<CTRL n> Same as **Down arrow**. It moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include **j** and *<CTRL j>*.

<CTRL o> Unused.

<CTRL p> Same as **Up arrow**. It moves the cursor one line up. A synonym is **k**.

<CTRL q> Not a command character. In *text input* mode, *<CTRL q>* quotes the next character, the same as *<CTRL v>*, except that some TELETYPE drivers will delete the *<CTRL q>* so that the editor never sees it.

<CTRL r> Redraws the current screen eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line.

<CTRL s> Unused. Some TELETYPE drivers use *<CTRL s>* to suspend output until *<CTRL q>* is invoked.

<CTRL t> Not a command character. During an insert with *autoindent* set and at the beginning of the line, it inserts *shiftwidth* white space.

<CTRL u> Scrolls the screen up (inverse of *<CTRL d>*). A count gives the amount of (logical) lines to scroll, and the count is remembered for future *<CTRL d>* and *<CTRL u>* commands. The previous scroll amount is common to both. On a dumb terminal, *<CTRL u>* will often require clearing and redrawing the screen further back in the file.

<CTRL v> Not a command character. In *text input* mode, it quotes the next character so that it is possible to insert nonprinting and special characters into the file.

<CTRL w> Not a command character. During an insert, it backs up as **b** would in *command* mode; the deleted characters remain on the display (see *<CTRL h>*).

<CTRL x> Unused.

<CTRL y> Exposes one more line above the current screen leaving the cursor where it is, if possible. There is no mnemonic value for this key; however, it is next to *<CTRL u>*.

<CTRL z> Unused.

<CTRL [> (*<ESC>*)
Cancels a partially formed command (such as a **z** when no following character has yet been given), ends inputs on the last line (read by commands such as **:**, **/**, and **?**), and ends insertions of new text into the buffer. If an *<ESC>* is given when in the command state, the editor rings the bell or flashes the screen. Therefore, you can press *<ESC>* if you do not know what is happening until the editor rings the bell. If you do not know if you are in *insert* mode, type *<ESC a>* and then the material to be input; the material will be inserted correctly whether or not you were in *insert* mode when you started.

<CTRL e> Unused.

<CTRL]> Searches for the word that is after the cursor as a tag. It is equivalent to typing **:ta**, this word, and then a *<CR>*.

<CTRL ↑> Equivalent to **:e #***<CR>*, returning to the previous position in the last edited file, or editing a file that you specified if you got a "No write since last change" diagnostic and do not want to have to type the file name again. You have to do a **:w** before *<CTRL ↑>* will work in this case. If you do

not wish to write the file, enter **:e!** #<*CR*> instead.

<*CTRL _*> Unused. Reserved as the command character for the TEKTRONIX* 4025 and 4027 terminals.

SPACE Same as **right arrow** (see **l**).

! An operator that processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name ended by <*CR*>. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise, the count is passed on to the object after the **!**. Thus **2!}***fmt*<*CR*> reformats the next two paragraphs by running them through the program *fmt*. To read a file or the output of a command into the buffer use **:r**. To simply execute a command use **:!**.

" Precedes a named buffer specification. There are named buffers (**1** through **9**) used for saving deleted text and named buffers (**a** through **z**) into which you can place text.

The macro character, when followed by a number, will substitute for a function key on terminals without function keys. In *text input* mode, if this is your erase character, it will delete the last character you typed and must be preceded with a \ to insert it since it normally backs over the last input character you gave.

$** Moves to the end of the current line. If the **:se list**<*CR*> command is used, then the end of each line will be shown by printing a **$ after the end of the displayed text in the line. When a count is used, the cursor advances to the end of the line following the count. For example, **2$** advances the cursor to the end of the following line.

---

* Registered Trademark of Tektronix, Inc.

% Moves to the parenthesis (**(**) or brace ({) that precedes or follows the parenthesis or brace at the current cursor position.

& A synonym for **:&**<*CR*>, analogous to the **ex & command.**

' When followed by a ', the cursor returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a nonrelative way. When followed by a letter (**a** through **z**), it returns to the line that was marked with this letter with an **m** command at the first nonwhite character in the line. When used with an operator such as **d**, the operation takes place over complete lines; if you use ', the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence. A sentence ends at a ., **!**, or **?** followed by either the end of a line or by two spaces. Any amount of closing characters (),], " , and ') may appear after the ., **!**, or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[). A count may be used before **(** to advance more than one sentence.

) Advances to the beginning of a sentence. A count repeats the effect. See **(** for the definition of a sentence.

***** Unused.

+ Same as <*CR*> when used as a command.

, Reverse of the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. Especially useful after typing too many **;** characters. A count repeats the search.

- Retreats to the previous line at the first nonwhite character. This is the inverse of + and <*CR*>. If the line moved to is not on the screen, the screen is scrolled or

cleared and redrawn. If a large amount of scrolling would be required, the screen is also cleared and redrawn with the current line at the center.

. Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then type . to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus, after a **2dw**, a **3.** deletes three words.

/ Reads a string from the last line on the screen and scans forward for the next occurrence of this string. The search begins when you press <*CR*>, and the cursor moves to the beginning of the last line to show that the search is in progress. The search may be ended with a <*DEL*> or <*RUB*>, or by backspacing when at the beginning of the bottom line returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can force whole lines to be affected. To do this, give a pattern with a closing / and then an offset +*n* or -*n*.

To include the / character in the search string, you must escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A **$** at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set **nomagic** in your *.exrc* file, you will have to precede the characters ., [, *, and ~ in the search pattern with a \ to get them to work as you would expect.

0 Moves to the first character on the current line. Also used, when forming numbers.

1-9 Used to form numeric arguments to commands.

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and ends with a <*CR*>, and the command is then executed. If you accidentally type **:**, you can return to where you were by typing <*DEL*> or <*RUB*>.

; Repeats the last single "character find" that used **f**, **F**, **t**, or **T**. A count iterates the basic scan.

< An operator that shifts lines left one *shiftwidth*, normally eight spaces. Like all operators, it affects lines when repeated, as in < <. Counts are passed through to the basic object, thus **3**<< shifts three lines.

> An operator that shifts lines right one *shiftwidth*, normally eight spaces. Affects lines when repeated as in >>. Counts repeat the basic object.

? Scans backward, the opposite of /. See the / description for details on scanning.

@ A macro character. Since this is the kill character, you must escape it with a \ to type it in during *text input* mode. It normally backs over the input given on the current line.

A Appends at the end of line, a synonym for **$a**.

B Backs up a word, where words are composed of nonblank sequences, placing the cursor at the beginning of the word. A count repeats the effect.

C Changes the rest of the text on the current line; a synonym for **c$**.

D Deletes the rest of the text on the current line; a synonym for **d$**.

E Moves forward to the end of a word, defined as blanks and nonblanks, like **B** and **W**. A count repeats the effect.

F Finds a single following character, backwards in the current line. A count repeats this search a specified amount of times.

G Goes to the line number given as preceding argument or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center, if necessary.

H Same as **Home arrow**. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count's line on the screen. In any case, the cursor is moved to the first nonwhite character on the line. If used as the target of an operator, full lines are affected.

I Inserts at the beginning of a line.

J Joins lines together, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two.

K Unused.

L Moves the cursor to the first nonwhite character of the last line on the screen. With a line count number, moves the cursor to the first nonwhite character of the indicated line from the bottom. Operators affect whole lines when used with **L**.

M Moves the cursor to the middle line on the screen at the first nonwhite position on the line.

N Scans for the next match of the last pattern given to / or **?**, but in the reverse direction. **N** is the reverse of **n**.

O Opens a new line above the current line and inputs text there up to an <*ESC*>. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete as the *slowopen* option works better.

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines; otherwise, the text is inserted between the characters before and at the cursor. The **P** character may be preceded by a named buffer specification " *x* to retrieve the contents of the buffer. Buffers **1** through **9** contain deleted material, buffers **a** through **z** are available for general use.

Q Quits from **vi** to **ex** *command* mode. In this mode, whole lines form commands and end with a <*CR*>. You can give all the **:** commands; the editor supplies the **:** as a prompt.

R Replaces characters on the screen with characters you type (overlay fashion). End with an <*ESC*>.

S Changes whole lines; a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers and erased on the screen before the substitution begins.

T Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.

U Restores the current line to its state before you started changing it.

V Unused.

W Moves forward to the beginning of a word in the current line where words are defined as sequences of blank/nonblank characters. A count repeats the effect.

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y Yanks a copy of the current line into the unnamed buffer to be put back by a later **p** or **P**; a synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.

ZZ Exits the editor (same as **:x**<*CR*>). If any changes have been made, the buffer is written out to the current file. Then, the editor quits.

[[Backs up the previous section boundary. A section begins at each macro in the *sections* options, normally a .NH or .SH, and also at lines that start with a form feed <*CTRL* /*>*. Lines beginning with { also stop [[; this makes it useful for looking backwards, a function at a time, in C programs.

\ Unused.

]] Forwards to a section boundary. See [[for a definition.

↑ Moves to the first nonwhite position on the current line.

_ Unused.

' When followed by a ', returns to the previous context. The previous context is set whenever the current line is moved in a nonrelative way. When followed by a letter (**a** through **z**), the cursor returns to the position that was marked with this letter. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line. If you use ', the operation takes place over complete lines.

a Appends arbitrary text after the current cursor position; the insert can continue to multiple lines by using <*CR*> within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion ends with an <*ESC*>.

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics or a sequence of special characters. A count repeats the effect.

c An operator that changes the following object, replacing it with the following input text up to an <*ESC*>. If more than part of a single line is affected, the text to be changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences.

d An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w**.

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect.

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find.

g Unused.

h Same as **Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or the synonyms (<*CTRL h*>) has the same effect. A count repeats the effect.

i Inserts text before the cursor; otherwise, like **a**.

j Same as **Down arrow**. Moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include <*CTRL j*> (linefeed) and <*CTRL n*>.

k Same as **Up arrow**. Moves the cursor one line up. <*CTRL p*> is a synonym.

l Same as **Right arrow**. Moves the cursor one character to the right. **SPACE** is a synonym.

m Marks the current position of the cursor in the mark register that is specified by the next character **a** through **z**. Return to this position or use with an operator using ' or '.

n Repeats the last / or **?** scanning commands.

o Opens new lines below the current line; otherwise, like **O**.

p Puts text after/below the cursor; otherwise, like **P**.

q Unused.

r Replaces the single character at the cursor with a single character you type. The new character may be a <*CR*>; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above, this is usually the more useful iteration of **r**.

s Changes the single character under the cursor to the text that follows up to an <*ESC*>; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c**.

t Advances the cursor up to the character before the next character typed. Most useful with operator such as **d** and **c** to delete the characters up to a following character.

You can use . to delete more if this does not delete enough the first time.

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states; thus, is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers.

v Unused.

w Advances to the beginning of the next word, as defined by **b**.

x Deletes the single character under the cursor. With a count, deletes that many characters forward from the cursor position, but only on the current line.

y An operator that yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, " *x*, the text is placed in that buffer also. Text can be recovered by a later **p** or **P**.

z Redraws the screen with the current line placed as specified by the following character:

<*CR*>	Specifies the top of the screen
.	Specifies the center of the screen
-	Specifies the bottom of the screen.

A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the amount of the line to place in the center of the screen instead of the default current line.

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option-- normally .IP, .LP, .PP, .QP and .bp. A paragraph also begins after a completely empty line and at each section boundary (see [[).

| Places the cursor on the character in the column specified by the count.

} Advances to the beginning of the next paragraph. See { for the definition of paragraph.

~ Switches character from lowercase to uppercase and vice versa.

<CTRL ?> (*<DEL>*)
Interrupts the editor returning it to command-accepting state.